Automotive Virtual Platform Specification

Version: 2.0

Contents

1 Introduction

This specification covers a collection of virtual device driver APIs and other requirements. The APIs constitute the defined interface between virtual machines and the virtualization layer, i.e. the hypervisor or virtualization "host system". Together, the APIs and related requirements define a virtual platform. This specification, the Automotive Virtual Platform Specification (AVPS) describes the virtual platform such that multiple implementations can be made, compatible with this specification.

A working group within the Hypervisor Project, led by the GENIVI Alliance, prepared this specification initially, and a good deal of information was provided by sources outside the automotive industry. GENIVI develops standard approaches for integrating operating systems and middleware in automotive systems and promotes a common vehicle data model and standard service catalog for use in-vehicle and in the *vehicle cloud*.

As part of this, GENIVI focuses on interoperability technologies beyond user-interactive systems including similarities among all in-vehicle ECUs, and works to collaboratively solve the most current concerns among system implementers. The introduction of virtualization into automotive systems represents one such significant challenge that shows a lot of promise but its complexity and potential for *lock-in* into difficult solutions shall not be underestimated.

The Hypervisor Project Group meetings are open to anyone and does not require membership of the alliance. The Group's work is intended to support in-car systems (ECUs) in the whole automotive industry, which includes support for different operating systems that the industry wants to use, and to create a specification that can be a basis for immediate implementations, while also being open licensed for possible further refinement.

Automotive systems use software stacks with particular needs. Existing standards for virtualization sometimes need to be augmented, partly because their original design was not based on automotive or embedded systems. The industry needs a common initiative to define the basics of virtualization as it pertains to Automotive whereas much of the progress in virtualization has come from IT/server consolidation and a smaller part from the virtualization of workstation/desktop systems. Embedded systems are still at an early stage, but the use of virtualization is increasing, and is starting to appear also in the upstream project initiatives that this specification relies heavily upon, such as VIRTIO.

A shared virtual platform definition in automotive creates many advantages:

- It simplifies moving hypervisor guests between different hypervisor environments.
- It can over time simplify reuse of existing legacy systems in new, virtualized, setups.
- Device drivers for paravirtualization, for operating system kernels (e.g. Linux) do not need to be maintained uniquely for different hypervisors.
- There is some potential for shared implementation across guest operating systems.
- There is some potential for shared implementation across hypervisors with different license models.

A specification can enable industry shared requirements and test suites, a common vocabulary and understanding to reduce the complexity of virtualization.

As a comparison, the OCI Initiative for Linux containers successfully served a similar purpose. There are now several compatible container runtimes, and synergy effects added to the most obvious effects of standardization. Similarly, there is potential for standardized hypervisor runtime environments that

promote portability and allow a standards-compliant virtual (guest) machine to run with significantly less integration efforts.

Hypervisors can fulfill this specification and claim to be compliant with its standard, still leaving opportunity for local optimizations and competitive advantages. Guest virtual machine (VMs) can be engineered to match the specification. In combination, this leads to a better shared industry understanding of how virtualization features are expected to behave, reduced software integration efforts, efficient portability of legacy systems and futureproofing of designs, as well as lower risks when starting product development.

1.1 Specification outline

The AVPS specification is intended to be immediately usable as a set of platform requirements, but also to start the conversation about further standardization and provide guidance for discussions between implementers and users of compatible virtualized systems that follow this specification. Each area is therefore split in a discussion section and a requirement section. The requirement section is the normative part. (See the chapter Chapter 2.2 for further guidance on adherence to the specification).

Each discussion section outlines various non-normative considerations in addition to the firm requirements. It also provides rationale for the requirements that have been written (or occasionally for why requirements were not written), and it often serves to summarize the state-of-the-art situation in each area.

1.2 Hardware considerations

This specification is intended to be hardware selection independent. We welcome all input to further progress towards that goal.

We should recognize, however, that all the software mentioned here (i.e. the hypervisor, and the virtual machine guests that execute kernels and applications) is machine code that is compiled for the real hardware's specific CPU architecture. For anyone who is new to this technology it is worthwhile to point out that it is not an emulation/interpreter layer for individual instructions such as that used to execute CPU-independent "byte-code" in a Java virtual machine. While this understanding is often assumed in similar documents, we point this out because such byte-code interpreters are inconveniently also called "virtual machines". This specification deals with hardware virtualization and hypervisors – i.e. execution environments whose virtual machine environments mimic the native hardware and can execute the same programs. The consequence is therefore that some differences in hardware CPU architectures and System-on-chip (SoC) capabilities must be carefully studied.

Some referenced external specifications are written by a specific hardware technology provider, such as Arm®, and define interface standards for the software/hardware interface in systems based on that hardware architecture. We found that some such interface specifications could be more widely applicable – that is, they are not too strongly hardware architecture dependent. The AVPS may therefore reference parts of those specifications to define the interface to virtual hardware. The intention is that the chosen parts of those specifications, despite being published by a hardware technology provider, should be possible to implement on a virtual platform that executes on *any* hardware architecture.

There are some areas that are challenging, or not feasible, to make hardware independent, such as:

- Access to trusted/secure computing mode (method is hardware dependent).
- Power management (standards are challenging to find).
- Miscellaneous areas, where hardware-features that are designed explicitly to support virtualization are introduced as a unique selling point.

Also, in certain modes, such as access to trusted/secure computing mode for example, note that the software is compiled for a particular CPU architecture as described above. The CPU architecture includes specialized instructions, or values written to special CPU registers, that are used to enter the secure execution mode. Programs in virtual machines should be able to execute such CPU instructions and should not need to be modified to make use of the trusted execution environment when running on a virtual platform.

Continuous work should be done to unify the interfaces in this virtual platform definition to achieve improved hardware portability.

1.3 Hardware pass-through

In a non-virtualization environment, a single operating system kernel accesses the hardware. Whereas, a virtual platform, based on a hypervisor, typically acts as an abstraction layer over the actual hardware. This layer enables multiple virtual machines (VMs), each running their own operating system kernel, to access a single set of hardware resources. The process of enabling simultaneous access to hardware from (multiple) VMs is called virtualization.

If the hardware is not designed for multiple independent systems to access it, then the hypervisor software layer must act as a go-between. It exposes virtual hardware, which has an implementation below its access interface, to sequence, parallelize, or arbitrate between requests to the real hardware resource.

One way to enable access from VMs to hardware is to minimize the required emulation code and instead give VMs access to the real hardware through a process called “hardware pass-through”. Unless the hardware has built-in features for parallel usage, pass-through effectively reserves the hardware for a specific VM and makes it unavailable to other VMs.

One advantage of hardware pass-through is that there is, generally, no performance loss compared to a virtual hardware / emulation alternative. In some systems, there may be other reasons for using hardware pass-through, such as reduced implementation complexity. This specification occasionally recommends handling some features using pass-through, but we have found that currently there is no standard for (and little movement towards standardizing) the exact method to configure hypervisors for hardware pass-through. The AVPS has a few general suggestions but does not currently propose a standard for how to make that portable. This may be an open question for future work.

For the purposes of this specification, hardware pass-through is interpreted as allowing direct hardware access for the code executing in the VM. For example, the code (typically part of the operating system kernel which is the owner of all hardware access) manipulates the actual hardware settings directly through memory-mapped hardware registers, or special CPU instructions.

The obvious case for pass-through is if no arbitration or sharing of the hardware resource between multiple VMs is necessary (or feasible), but in some cases it can be beneficial for the Hypervisor to present an abstraction or other type of API to access the hardware, even if this API does not provide shared access to the hardware. The reason is that direct manipulation of registers by VM guests may have unintended side effects on the whole system and other VMs. In the simplest case, consider that mutually unrelated hardware functions might even be configured by different bits in the same memory-mapped register. This would make it impossible to securely split those functions between two different VMs if normal pass-through access to this register is offered. Instead, the access to this register must be mediated by a Hypervisor, even if it does not support arbitration or sharing of the feature from multiple VMs. If the virtual platform is providing a different interface than direct hardware access on the corresponding native hardware, then it is here still considered virtualization (and it also implies paravirtualization, in fact). In other words, we aim to avoid calling such an implementation pass-through even if it does not enable concurrent access or add any additional features. Like other such virtual interfaces, it is encouraged to also standardize those APIs that give this seemingly “direct” access, but in a different form than the original hardware provides.

1.4 Virtualization implementation designs

Comparing virtualization solutions can be difficult due to differing internal designs. Sometimes these are real and effective differences whereas sometimes only different names are used for mostly equivalent

solutions, or simply different emphasis is placed on certain aspects.

Some hypervisors start from an emulator for a hardware platform and add native CPU-instruction virtualization features to it. Other platforms match a simple model often assumed in descriptions in this document, in which a single component named “hypervisor” is, in effect, the implementation of the entire virtual platform.

Some others conversely state that their actual hypervisor is minimal, and highlight especially the fact that the purpose of the hypervisor is only to provide separation and scheduling of individual virtual machines – thus it implements a kind of “separation kernel”. The remaining features of those designs might then be delegated to dedicated VMs, either with unique privileges or even VMs that are considered almost identical in nature to the “guest” VMs. Consequently, it is then those provided VMs that implement some of the API of the virtual platform, i.e. the APIs available for use by the “guest” VMs.

Some environments use different nomenclature and highlight the fact that parts of the virtual platform may have multiple privilege levels, “domains”. These are additional levels defined beyond the simple model of:
user space < OS kernel < Hypervisor.

Some offerings propose that real-time functions can be run using any Real-Time Operating System (RTOS) that runs as a guest operating system in one VM, as an equivalent peer to a general-purpose VM (running Linux kernel for example). Whereas others put emphasis on their implementation being an RTOS kernel first, that provides direct support for real-time processes/tasks (like an OS kernel), and where the RTOS kernel simultaneously acts as a hypervisor towards foreign/guest kernels. The simple description of this without more deeply defining the actual technical design would be that it is an RTOS kernel that also implements a hypervisor.

The design of the hypervisor/virtualization platform and the design of the full system (including guest VMs) sometimes tend to be interdependent, but the intention of this specification is to try to be independent of such design choices. Although other goals for this specification have also been explained, starting with portability of guest VMs should bring any concerns to the surface. If an implementation can follow the specification and ensure such portability, then the actual technical design of the Hypervisor or virtual platform is free to vary.

These differences of philosophy and design are akin to the discussion of monolithic kernels vs. microkernels in the realm of operating systems but here the discussion is about the “virtual platform kernel” (i.e. hypervisor) instead.

Suffice to say that this specification does not strive to advocate only one approach or limit the design of the virtualization platforms, but still strives to maximize compatibility and shared advancement and therefore focus primarily on defining the APIs between a guest VM (operating system kernel) and the virtual platform it runs on.

Please be aware that certain parts of the discussion sections may still speak about the “Hypervisor” implementing a feature, but it should then be interpreted loosely, i.e. it should fit also designs that provide identical feature compatibility but has the implementation delegated to some type of VM.

1.5 Draft specification state

Some areas of the platform are not ready for a firm definition of a standard virtual platform. The reasons could include:

- There are sometimes proposals made upstream in VIRTIO, for example, that appear to fit the needs of a virtual platform but are not yet approved and are still under heavy discussion. The AVPS working group then considered it better to wait for these areas to stabilize.
- The subject is complex and requires further study. Implementations of virtual hardware devices in this area might not even exist for this reason and we therefore defer to a pass-through solution. As an intermediary, the feature may be implemented with a simple hardware driver that exposes a bespoke interface to the VMs but does not make it available to multiple VMs as virtual-hardware and is neither subject for standardization.
- It could be an area of significant innovation and hypervisor implementers therefore wish to be unencumbered by requirements at this time, to either allow for free investigation into new methods, or for differentiating their product.
- The subject area might not have any of the above issues, but the working group has not had time and resources to cover it yet. *This is an open and collaborative process, so input is welcome from additional contributors.*

For situations described above, there are no requirements yet, but we have sometimes kept the discussion section to introduce the situation, and to prepare for further work. Whereas all parts are open for improvement in later versions, some chapters will have the explicit marker:
Potential for future work exists here.

We invite volunteers to provide input or write those chapters. Joining the working group discussions would be a welcome first step.

2 Automotive Virtual Platform - Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

2.1 Architecture

The intended applicability of this specification is any type of computer systems inside of vehicles, a.k.a. Electronic Control Units (ECUs).

There are no other assumptions about the high-level system architecture at this time.

2.2 Conformance to specification

Optional features vs. optional requirements.

The intention of a standard is to maximize compatibility. For that reason, it is important to discuss how to interpret conformance (compliance) to this specification. The specification might in its current state be used either as guidance or as firm requirements, depending on project agreements. In an actual automotive end-product, there of course also remains the freedom to agree on deviations among partners, so the intention of this chapter is not to prescribe project agreements, but only to define what it means to follow, or be compliant with this specification.

Several chapters essentially say: “if this feature is implemented… then it shall be implemented as follows”. The inclusion of this feature is optional, but adherence to the requirements is not. Note first that this is the feature of the virtualization platform, not the feature of an end-user product. Another way to understand this is that if the mentioned feature exists (in a compliant implementation) then it shall not be implemented in an alternative or incompatible way (an exception should only be made if this is explicitly offered in addition to the specified way).

The specification may also list some features as mandatory. End-user products are, as noted, free to include or omit any features, so the intention here is only to say that if a virtual platform implementation (wants to claim that it follows the specification, then those features must be available. While not every such feature might be used in an end-user product, fulfilling the specification means they exist and are already implemented as part of the platform implementation, and are being offered by the hypervisor implementation / virtual platform, to the buyer/user of that platform.

2.3 Virtualization support in hardware

When running on hardware that supports it, then the virtualization hardware support for things like interrupts and timers, performance monitoring, GPU, etc., are generally encouraged to be used. But it should avoid contradicting the standard APIs as listed here. If in doubt, the virtual platform shall follow the specification (provide the feature as specified) and perhaps provide alternatives in addition to it (see previous discussion Optional features vs. optional requirements). In some cases, there is no conflict

because the optimized hardware support might be used “below the API”, in other words in the implementation of the virtual platform components while still fulfilling the specified API.

Whenever any conflict arises between specific hardware support for virtualization and the standard virtual platform API, then we strongly encourage raising this for community discussion to affect future versions of the specification. It might be possible, through collaboration, to adjust APIs so that these optimizations can be used, when the hardware supports it. And if this is not possible, then a specification like this can still explicitly document alternative acceptable solutions, rather than non-standard optimizations being deployed with undocumented/unknown consequences for portability and other concerns.

2.4 Hardware emulation

A virtualization platform may deal with the emulation of hardware features for various reasons. Lacking explicit support in the hardware, it might be the only way to implement hardware device sharing. In particular, we want to note that for less capable hardware, the hypervisor (or corresponding part of virtual platform) may need to implement emulation of some hardware features that it does not have but which are available on other hardware. It would typically provide much lower performance, but if semantic compatibility can be achieved with the virtual platform as it is specified in this document, then this still supports portability and integration concerns.

2.5 General system

2.5.1 Booting guest virtual machines

Discussion:

A boot protocol is an agreed way to boot an operating system kernel on hardware or virtual hardware. It provides information to OSes about the available hardware, usually by providing a device tree description. Certain hardware specifics are provided in an abstract/independent way by the hypervisor (or, typically, by the initial boot loader on real hardware). These can be base services like real-time clock, wake-up reason, etc.
Outside of the PC world, boot details have traditionally been very hardware/platform specific.

The Embedded Base Boot Requirements (EBBR) specification from ARM® has provided a reasonable outline for a boot protocol that can also be implemented on a virtual platform. The EBBR specification defines the use of UEFI APIs as the way to do this.

A small subset of UEFI APIs is sufficient for this task and it is not too difficult to handle. Some systems running in VMs do not implement the EBBR protocol (e.g. ported legacy AUTOSAR Classic based systems and other RTOS guests). These are typically implemented using a compile-time definition of the hardware platform. It is therefore expected that some of the code needs to be adjusted when porting such systems to a virtual platform. However, requiring EBBR is a stable basis for operating systems that can use it.

We expect two categories of boot information handling:

1) Static setup: The hypervisor exposes the devices at statically defined addresses. This allows the system integrator to incorporate the relevant devices into the guest at compile time and configure the hypervisor indirectly. Thus, an explicit exposed device configuration provides a consistent basis between the compile time configuration of the guest and the environment exposed to the guest by the hypervisor at runtime. This is beneficial when dealing with specialized, small footprint OS which commonly do not process platform specific information at boot-time or when using manual or fully automatized SW configuration frameworks as common for pre-compile configuration of AUTOSAR® SW stacks.

2) Dynamic setup: The hypervisor decides where to place devices and communicates that to the guest operating system at guest boot-time. Once again, this can be done by device-tree dynamically generated and exposed to the guest OS. Dynamic handling of boot information is more flexible as it does not require a potentially err-prone re-configuration of the guest. In turn, the guest needs to be able to process this information at boot-time which is often only found with general-purpose OS such as Linux.

In both cases the specification of platform layout benefits from deciding on a firm standard for how the Hypervisor describes a hardware/memory map to legacy system guests. The consensus seems to be that Device Tree is the most popular and appropriate way, and an independent specification is available at https://devicetree.org. Device tree descriptions are also a human readable specification and can be directly incorporated into any kind of system architecture documentation.

The group found that Android and typical GNU/Linux style systems often have different boot requirements. However, Android could be booted using UEFI and it is therefore assumed that the EBBR requirements can be applicable for running guests.

AVPS Requirements:

{AVPS-v2.0-1}
Virtual platforms that support a dynamic boot protocol MUST implement (the mandatory parts of*) EBBR.

{AVPS-v2.0-2}
Since EBBR allows either ACPI or Device Tree implementation, this can be chosen according to what fits best for the chosen hardware architecture and situation.

{AVPS-v2.0-3}
For systems that do not support a dynamic boot protocol (see discussion), the virtual hardware SHOULD still be described using a device tree format, so that guest-VM and hypervisor implementers can agree on the implementation using that formal description.

2.6 Storage

Discussion:

When using hypervisor technology data on storage devices needs to adhere to high-level security and safety requirements, such as isolation and access restrictions. VIRTIO and its layer for block devices provides the infrastructure for sharing block devices and establishing isolation of storage spaces. This is because actual device access can be controlled by the hypervisor. However, VIRTIO favors generality over using hardware-specific features. This is problematic in case of specific requirements regarding robustness and endurance measures often associated with the use of persistent data storage such as flash devices. In this context we can spot three relevant scenarios:

Features transparent to the guest OS. For these features, the required functionality can be implemented close to the access point, e.g., inside the actual driver. As an example, think of a flash device where the flash translation layer (FTL) needs to be provided by software. This contrasts with, for example, MMC flash devices, SD cards and USB thumb drives where the FTL is transparent to the software.

Features established via driver extensions and workarounds at the level of the guest OS. These are features which can be differentiated at the level of (logical) block devices such that the guest OS uses different block devices and the driver running in the backend enforces a dedicated strategy for each (logical) block device. E.g., guest OS and its application may require different write modes, here reliable vs. normal write.

Features that call for an extension of the VIRTIO Block device driver standard.

Meeting automotive persistence requirements

A typical automotive ECU is often required to meet some unique requirements.
It should be said that the entire “system” (in this case defined as the limits of one ECU) needs to fulfil these and a combination of features in the hardware, virtualization layer, operating system kernel, and user-space applications may fulfil them together.

Typical automotive persistence requirements are:

The ability to configure some specific data items (or storage areas/paths in file system) that are guaranteed to “immediately” get stored in persistent memory (i.e. within a reasonable and bounded time). Although

the exact implementation might differ, it is typically considered as a i.e. "Write Through mode" from the perspective of the user space program in the guest, as opposed to a "Cached mode". In other words, data is written through caches to the final persistent storage. Fulfilling this requires analysis because systems normally use filesystem data caching in RAM for performance reasons (and possibly inside Flash devices as well). The key challenge with this approach is that there are well known limits to Flash memory technology being “worn out” after a certain number of write cycles.

Data integrity in case of sudden power loss

Data must not be “half-way” written or corrupted in any other sense. This could be handled by journaling that can recognize “half-way” written data upon next boot and roll back the data to a previous stable state. The challenge is that rolling back may violate some requirement that trusts that data was “immediately stored” as described in 1). All in all, requirement 2) must be evaluated in combination with requirement 1).

Hardware that loses power cannot execute code to write data from RAM caches to persistent memory. The implementation that balances “write through” with “data integrity upon power loss” may differ. Some systems can include a hardware “warning” signal that power is about to fail (while internal capacitors in the power module might continue to provide power for a very short time after an external power source disappears). This could allow the system to execute emergency write-through of critical data.

Flash lifetime (read/write cycle maximums) must be guaranteed so that hardware does not fail. A car is often required to have a lifetime of hardware beyond 10-15 years.

As we can see, these requirements are interrelated and can be in conflict. They are only solved by enabling a limited use of write-through (1.), and simultaneously finding solutions for the other two.

The persistent storage software stack is already complex, from the definition of APIs that can control different data categories (req 1 is only to be used for some data), storing data using a convenient application programming interface, operating-system kernel implementation of filesystems, and block-level storage drivers, flash-memory controllers which in themselves (in hardware) have several layers implementing the actual storage. Flash memory controllers have a block translation layer, which ensures that only valid and functioning flash cells are being used and that automatically weeds out failing cells, spreads the usage across cells (“wear levelling”), and implements strategies for freeing up cells and reshuffling data into contiguous blocks. When a virtualization layer is added, there can be another level of complexity inserted.

Further design study is needed here, and many companies are forced to do this work on a particular solution on a single system. There is no single answer but there is significant common work that could get done. We would encourage the industry to continue this discussion and to develop common design principles through collaboration on implementation, analysis methods and tooling, and then to discuss how standards may ensure the compatibility of these solutions.

2.6.1 Block Devices

Discussion:

While the conclusion from the introduction remains, that a particular system must be analyzed and designed to meet its specific requirements, the working group concluded that for the AVPS purpose, VIRTIO block device standards could be sufficient in combination with the right requests being made from user space programs (and running appropriate hardware devices below).

With VIRTIO the options available for write cache usage are as below:

Option 1: WCE = on, i.e. device can be set in write-through mode, in VIRTIO block device.

Option 2: WCE = off, i.e. the driver follows BLK_SYNC after BLK_WRITE. The open device call with O_SYNC from user space in Linux ensures fsync after the write operation. The file-system mount can also enable synchronous file operation and it is available through O-option (and only some guarantee to respect it 100%). The Linux API creates a block request with FUA (Forced Unit Access) flag for ensuring that the operation is performed to persistent storage and not through volatile cache.

The conclusion is that VIRTIO does not break the native behavior. Even in the native case write cache can be somewhat uncertain but VIRTIO does not make it worse.

VIRTIO does not provide Total Blocks Written or other parts of S.M.A.R.T. information from the physical disks. We agree that omitting access to host-related information such as disk health data from the virtual platform API is appropriate, because giving virtual machines access to this might have subtle security related effects. Imagine for example the ability for a malicious program to probe for behaviors that cause declining disk health and then exploit those behaviors, or consider generally the implication of VMs analyzing what other VMs are doing.

NOTE:

UFS devices provide LUN configuration (Logical Unit Number) also called as UFS provisioning support such that the devices can have more than one LUNs. A FUA to one LUN does not mean all caches will be flushed. eMMC does not provide such support. SCSI devices like HDD provide the support for multiple LUNs. Similarly, PCIe NVMe SSD can be configured to have more than one namespaces. One could map partitions onto LUNs/namespaces (some optimized for write-through and some for better average performance) and build from there.

AVPS Requirements:

`{AVPS-v2.0-4}` The platform MUST implement virtual block devices according to chapters 5.2 in [VIRTIO].

`{AVPS-v2.0-5}` The platform MUST implement support for the VIRTIO_BLK_F_FLUSH feature flag.

`{AVPS-v2.0-6}` The platform MUST implement support for the VIRTIO_BLK_F_CONFIG_WCE feature flag.

`{AVPS-v2.0-7}` The platform MUST implement support for the VIRTIO_BLK_F_DISCARD feature flag.

The platform MUST implement support for the VIRTIO_BLK_F_WRITE_ZEROS feature flag.

2.7 Communication Networks

2.7.1 Standard networks

Discussion:

Standard networks include those that are not automotive specific and not dedicated for a special purpose like Automotive Audio Bus(R) (A2B). Instead these are frequently used in the computing world, and for our purposes nowadays that means almost always within the Ethernet family (802.xx standards).

These are typically IP based networks, but some of them simulate this level through other means (e.g. vsock, which does not use IP addressing). The physical layer is normally some variation of the Ethernet/Wi-Fi standard(s) (according to standards 802.*) or other transport that transparently exposes a similar network socket interface. Certain alternative networks will provide a channel with Ethernet-like capabilities within them (APIX□, MOST□ 150, ...) but those are automotive-specific. These might be called out specifically where necessary, or just assumed to be exposed as standard Ethernet network interfaces to the rest of the system.

Some existing virtualization standards to consider are:

- VIRTIO-net = Layer 2 (Ethernet / MAC addresses)
- VIRTIO-vsock = Layer 4. Has its own socket type. Optimized by stripping away the IP stack. Possibility to address VMs without using IP addresses. Primary function is Host (HV) to VM communication.

Virtual network interfaces ought to be exposed to user space code in the guest OS as standard network interfaces. This minimizes custom code appearing because the usage of virtualization is minimized.

MTU may differ depending on the actual network being used. There is a feature flag that a network device can state its maximum (advised) MTU and the guest application code might make use of this to avoid segmented messages.

The guest may require a custom MAC address on a network interface. This is important for example when setting up bridge devices which expose the guest's MAC address to the outside network.

To avoid clashes the hypervisor must be able to set an explicit (stable across reboots) MAC address in each VM.

In addition, the guest shall be able to set its own MAC address, although the HV may be set up to deny this request for security reasons.

Offloading and similar features are considered optimizations and therefore not absolutely required.

The virtual platform ought to provide virtual network interfaces using the operating system's normal interface concept (i.e. they should show up as a network device) but the exact details of that may depend on the operating system run in the VM, if the virtual platform includes paravirtualization, which is very

likely. This ought therefore not be written as a requirement in the kernel-to-hypervisor API, but it shall be considered when providing a platform solution.

AVPS Requirements:

{AVPS-v2.0-8} If the platform implements virtual networking, it MUST use the VIRTIO-net required interface between drivers and Hypervisor.

{AVPS-v2.0-9} The hypervisor/equivalent MUST provide the ability to dedicate and expose any hardware network interface to one virtual machine.

{AVPS-v2.0-10} Implementations of VIRTIO-net MUST support the following feature flags:

{AVPS-v2.0-11} VIRTIO_NET_F_MTU

{AVPS-v2.0-12} VIRTIO_NET_F_MAC

{AVPS-v2.0-13} VIRTIO_NET_F_CTRL_MAC_ADDR

{AVPS-v2.0-14} The Hypervisor MAY implement a whitelist or other way to limit the ability to change MAC address from the VM.

2.7.2 VSock and inter-VM networking

Discussion:

VSock is a “virtual” (artificial) socket type in the sense that it does not implement all the layers of a full network stack that would be typical of something running on Ethernet, but instead provides a simpler implementation of network communication directly between virtual machines (or between VMs and the Hypervisor). The idea is to shortcut anything that is unnecessary in this local communication case while still providing the socket abstraction. Higher level network protocols should be possible to implement without change.

When using the VSock (VIRTIO-vsock) standard, each VM has a logical ID but the VM normally does not know about it. Example usage: Running an agent in the VM that does something on behalf of the HV.

For the user-space programs the usage of vsock is very close to transparent, but programs still need to open the special socket type (AF_VSOCK). In other words, it involves writing some code that is custom for the virtualization case, as opposed to native, and we recommend system designers to consider this with caution for maximum portability.

Whereas vsock defines the application API, multiple different named transport variations exist in different hypervisors, which means the driver implementation differs depending on chosen hypervisor. VIRTIO-vsock however locks this down to one chosen method.

AVPS Requirements:

{AVPS-v2.0-15} The virtual platform MUST be able to configure virtual inter-VM networking interfaces (either through VSOCK or providing other virtual network interfaces that can be bridged)

{AVPS-v2.0-16} If the platform implements VSOCK, it MUST use the VIRTIO-vsock required API between drivers and Hypervisor.

2.7.3 Wi-Fi

Discussion:

Wi-Fi adds some additional characteristics not used in wired networks: SSID, passwords/authentication, signal strength, preferred frequency…

There are many potential systems designs possible and no single way forward for virtualizing Wi-Fi hardware. More discussion is needed to converge on the most typical designs, as well as the capability (for concurrent usage) of typical Wi-Fi hardware. Together this may determine how much it would be worth to create a standard for virtualized Wi-Fi hardware.

Examples of system designs could include:

Exposing Wi-Fi to only one VM and let that act as an explicit gateway/router for the other VMs.

Let the Wi-Fi interface be shared on the Ethernet level, similar to how other networks can be set up to be bridged in the HV. In this case some of the network setup such as connecting to an access point, handling of SSID and authentication would need to be done by the Hypervisor, or at least one part of the system (e.g. delegate this task to a specific VM).

Virtualizing the Wi-Fi hardware, possibly using capabilities in some Wi-Fi hardware that allow connecting to multiple access points at the same time.

To do true sharing it would be useful to have a Wi-Fi controller that can connect to more than one endpoint (Broadcom, Qualcomm, and others, reportedly have such hardware solutions.)

A related proposal is MAC-VTAB to control pass-through of the MAC address from host to VM. Ref: https://github.com/ra7narajm/VIRTIO-mac80211

AVPS Requirements:

This chapter sets no requirements currently since the capability of typical Wi-Fi hardware, the preferred system designs, and defining standards for a virtual platform interface needs more investigation.

Potential for future work exists here.

2.7.4 Time-sensitive Networking (TSN)

Discussion:

TSN adds the ability for ethernet networks to handle time-sensitive communication including reserving guaranteed bandwidth, evaluating maximum latency through a network of switches, and adding fine-grained timestamps to network packets. It is a refinement of the previous Audio-Video Bridging (AVB) standards, in order to serve other time-sensitive networking.

It is not yet clear to us how TSN affects networking standards. Many parts are implemented at a very low level, such as time-stamping packets being done in some parts of the Ethernet hardware itself to achieve the necessary precision. For those parts it might be reasonable to believe that nothing changes in the

usage, compared to non-virtual platforms, or that little need to change in the Virtual Platform API definitions compared to standard networks.

Other parts are however on a higher protocol level, such as negotiating guaranteed bandwidth, and other monitoring and negotiation protocols. These may or may not be affected by a virtual design and further study is needed.

A future specification may include that the hypervisor shall provide a virtual ethernet switch and implement the TSN negotiation protocols, as well as the virtual low-level mechanisms (e.g. Qbv Gate Control Lists). This requirement would be necessary only if TSN features are to be used from the VM.

AVPS Requirements:

To be added in a later version.

Potential for future work exists here.

2.8 Graphics

Introduction:

The Graphics Processing Unit is one of the first and most commonly considered shared functionality when placing multiple VMs on a single hardware, and yet it is likely the most challenging. Standard programming APIs are relatively stable for 2D, but significant progress and change happens in the 3D programming standards, as well as feature growth of GPUs, especially for built-in virtualization support.

2.8.1 GPU Device in 2D Mode

VIRTIO-GPU is appropriate and applicable for 2D graphics but using the 3D mode only is more common these days.

In the unaccelerated 2D mode there is no support for DMA transfers from resources, just to them. Resources are initially simple 2D resources, consisting of a width, height and format along with an identifier. The guest must then attach a backing store to the resources for DMA transfers to work.

When attaching buffers use pixel format, size, and other metadata for registering the stride. With uncommon screen resolutions, this might be unaligned, and some custom strides might be needed to match.

AVPS Requirements: for 2D Graphics

Device ID.

`{AVPS-v2.0-17}` in [VIRTIO-GPU].	The device ID MUST be set according to the requirement in chapter 5.7.1

Virtqueues.

{AVPS-v2.0-18} The virtqueues MUST be set up according to the requirement in chapter 5.7.2 in [VIRTIO-GPU].

Feature bits.

{AVPS-v2.0-19} The VIRTIO_GPU_F_VIRGL flag, described in chapter 5.7.3 in [VIRTIO-GPU], MUST NOT be set.

{AVPS-v2.0-20} The VIRTIO_GPU_F_EDID flag, described in chapter 5.7.3 in [VIRTIO-GPU], MUST be set and supported to allow the guest to use display size to calculate the DPI value.

Device configuration layout.

{AVPS-v2.0-21} The implementation MUST use the device configuration layout according to chapter 5.7.4 in [VIRTIO-GPU].

- The implementation MUST NOT touch the reserved structure field as it is used for the 3D mode.

Device Operation.

{AVPS-v2.0-22} The implementation MUST support the device operation concept (the command set and the operation flow) according to chapter 5.7.6 in [VIRTIO-GPU].

- The implementation MUST support scatter-gather operations to fulfil the requirement in chapter 5.7.6.1 in [VIRTIO-GPU].
- The implementation MUST be capable to perform DMA operations to client's attached resources to fulfil the requirement in chapter 5.7.6.1 in [VIRTIO-GPU].

VGA Compatibility.

{AVPS-v2.0-23} VGA compatibility, as described in chapter 5.7.7 in [VIRTIO-GPU], is optional.

2.8.2 GPU Device in 3D Mode

Discussion:

There is ongoing development in 3D APIs and this impacts setting a standard for virtualization of 3D graphics. In addition, it is desirable to make use of hardware support for virtualization in modern SoCs to get further improved isolation/separation and higher performance, compared to a more abstract virtual graphics API.

Input requirements for GPU Virtualization

Since this is a particular challenging area we go through a list of system design considerations:

- Security
- Safety
- Policy / QoS / Performance isolation / Resource reservation
- Power Management
- Boot flow

- Hypervisor implementation
- Portability (Upgrading of HW and SW shall be easy/efficient. The base architecture shall be common enough such that components can be upgraded independently.
- Licensing of code involved
- Hardware blocks:
- (GP)GPU
- Framebuffer Composition
- IOMMU / GIC ITS (isolated device memory, isolated device MSI handling)
- Security
- Each queue-pair needs to have an IOMMU unit attached
- Partitioning vs. covert channels
- Memory bus separation

If safety-relevant use-case included

- Targets all HW blocks (GPU, Composition, CPU/GIC)
- Priority of request handling
- Watchdog for progress and/or output results of graphics?
- Generic hardware virtualization requirements / issues
- Multiple queues to pass to clients
- At least one device interrupt pre client
- Partitioning of HW (e.g. for GPU, DSP, …)
- Policy / configuration of queue handling
- QoS / queue scheduling, covert channels vs. partitioning
- IOMMU per client
- Secure interrupts (MSI/ITS)

Strategies

There are 3 primary approaches to virtualization support:

1) API layering (e.g. VIRTIO/VirGL. Less performance, compatible everywhere)
2) Mediated hardware access
3) Direct hardware access (using HW support for virtualization) (typically ARM, Imagination, NVIDIA, and others). This is usually the most performant solution.

1) API layering (VIRTIO-GPU with VirGL)

Even using general APIs like VIRTIO, rendering operations will be executed on the host GPU and therefore requires a GPU with 3D support on the host machine.

The guest side requires additional software in order to convert OpenGL commands to the raw graphics stack state (Gallium state) and channel them through VIRTIO-GPU to the host. Currently the 'mesa' library is used for this purpose. The backend then receives the raw graphics stack state and interprets it using the *virglrenderer* library from the raw state into an OpenGL form, which can be executed as entirely normal OpenGL on the host machine. The host also translates shaders from the TGSI format used by Gallium into the GLSL format used by OpenGL. Currently TGSI is implemented in Mesa and supported primarily in Linux.

The API is however stable and could potentially be used by other OSes in a different implementation. It might be difficult to require this as a cross-platform standard for virtualization.

The VIRTIO based solution should become more flexible and independent from third party libraries on the guest side as soon as Vulkan support within *virtio-gpu* is introduced. It will be achieved by the fact that Vulkan uses Standard Portable Intermediate Representation as an intermediate device-independent language and the language is already a part of the standard itself, so no additional translation between the guest and the host are required. It is still a work in progress [VIRTIO-VULKAN].

Details of the VirGL approach:

- Run HW driver in a subsystem (i.e. VM)
- Transport for API commands (e.g. OpenGL), e.g. virtio-gpu
- Use generic drivers in client-VMs
- Slower than hardware-provided virtualization but allows to have hardware-independent software components in the client guests.

2) Mediated hardware access

Mediated hardware access is as it sounds a strategy where there is some hardware support but the Hypervisor must take a fairly large responsibility to implement the virtualization features. This has been seen on some platform, primarily Intel-based. It is a less popular design at this point so there are no further details here for now.

3) Hardware support for virtualization

Modern automotive SoCs have special features built in to support GPU virtualization, specifically to allow more than one system (VM) to access the graphics capabilities while ensuring separation. Such features shall guarantee that critical work tasks (in one VM) can use GPU capabilities unaffected by less critical tasks (other VMs).

Typical hardware features for supporting virtualization include:

- Tagging memory accesses with a number (VM ID or OS ID) that is used as part of the address when doing address translation in the MMU. Thus, it guarantees that a GPU operation belonging to a particular VM can only access the memory belonging to that VM.
- Handling interrupts from GPU operations:
- Multiple separate interrupts that can be assigned to specific VMs
- Single interrupt line – hypervisor must direct to the correct VM

Hardware support for separate command queues/input/pipelines for GPU operations (to be used by separate VMs)

Assignment strategies for GPU calculation cores:

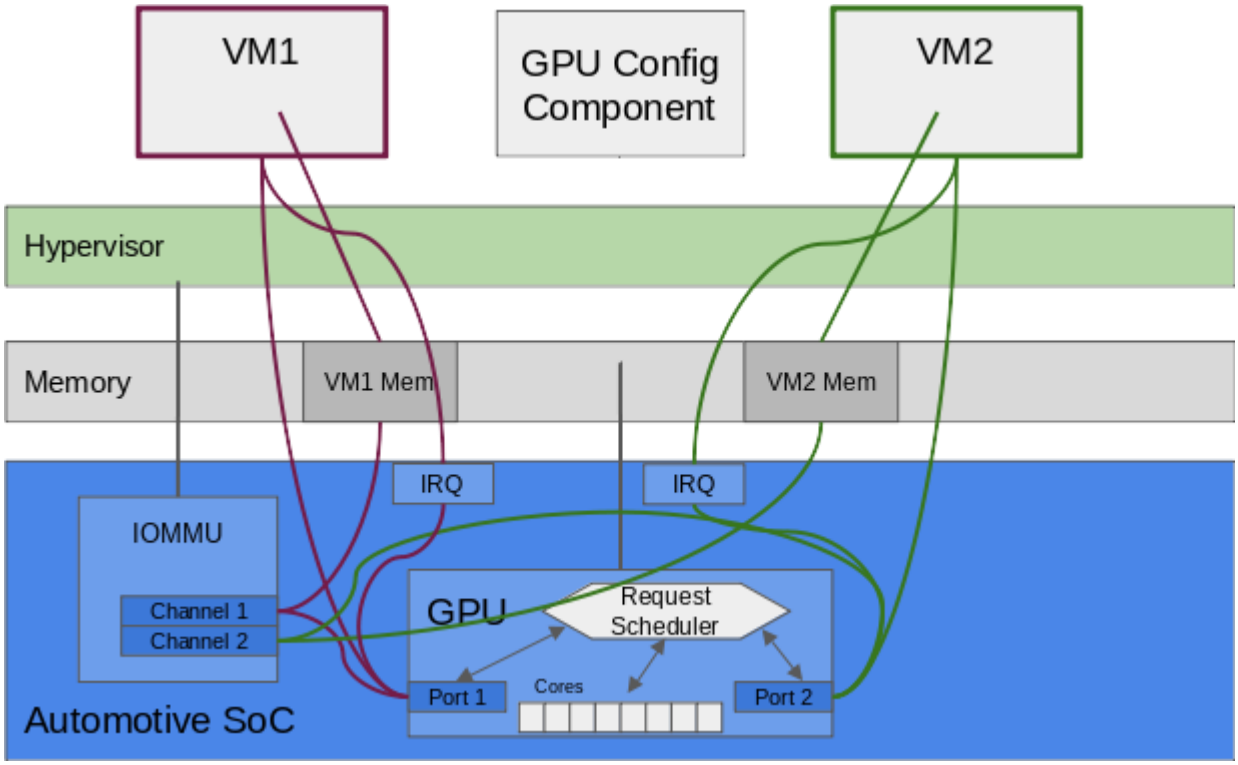
Priority based

This feature can interrupt the processing of a lower priority job if a job appears in a higher priority queue. It provides a flexible model that is easy to understand, but context switching between jobs may use some resources.

Partition based

This feature enables dedicating certain parts of the GPU calculation cores to specific VMs, thus guaranteeing their availability for critical tasks.

Generalized design of virtualization capable GPUs



The intention with the general model is to note the similarities among multiple hardware implementations of virtualization, even if the details differ slightly:

There is generally one or several job queues for VMs to define the GPU processing needs.

There is one interrupt queue per VM to notify the VMs when calculations are completed, error conditions, etc. If the hardware cannot separate the interrupts per VM, the Hypervisor needs to be involved in routing the interrupt to the right VM. It is also possible that all VMs are notified, and they need to look at their queues to know if this is a relevant interrupt. There are security downsides to this of course.

In all solutions there is some method of allocating jobs to GPU cores according to VM importance/priority/status.

Ultimately, these features are similar enough that initial system planning can be done independent of hardware, and some basic level of VM portability can be planned for over time. As is often the case, the product requirements (number of VMs, GPU capability and performance) still needs to be compared to the exact hardware capability, but that is true also for non-virtualized systems. Ultimately, this specification aims primarily to find common solutions for Hypervisors and cannot fully address portability across hardware.

There appears however here to be an opportunity to make a formal abstraction over these similarities, which we would like to encourage, but we are not aware of such a fully unified API at this point.

Therefore, after this analysis the platform requirements for GPU virtualizations are still short and there may still be room for improvement. Since graphical automotive systems are most likely to select a hardware platform with virtualization support, we also conclude by requiring the fully portable VIRTIO-GPU method as optional.

A short summary of what the Hypervisor / VP must ensure while implementing a system with hardware virtualization support:

- Set up the GPU to provide a set of queues with interrupts
- Pass queues to VMs through hypervisor configuration
- Pass (at least) one device interrupt to the guest
- Setup-component to run in HV (or dedicated VM)
- Configuration of queue handling policies (QoS, etc.), and/or configuration of partitioning of GPU resources (performance isolation)
- Minimum set of required policies? (E.g. should work defined by different VMs be scheduled using equal share or with different priorities?)

AVPS Requirements: for 3D Graphics

`{AVPS-v2.0-24}` VIRTIO-GPU is an optional feature.

`{AVPS-v2.0-25}` If the hardware supports graphics virtualization (according to the common model described above) then the VP shall implement it with APIs that promote future portability. In particular, the full driver stack shall aim for that graphical software can be written as independently as possible. In effect, each VM can then behave as if it had its own dedicated GPU.

2.8.3 Virtualization of framebuffer / composition

Discussion:

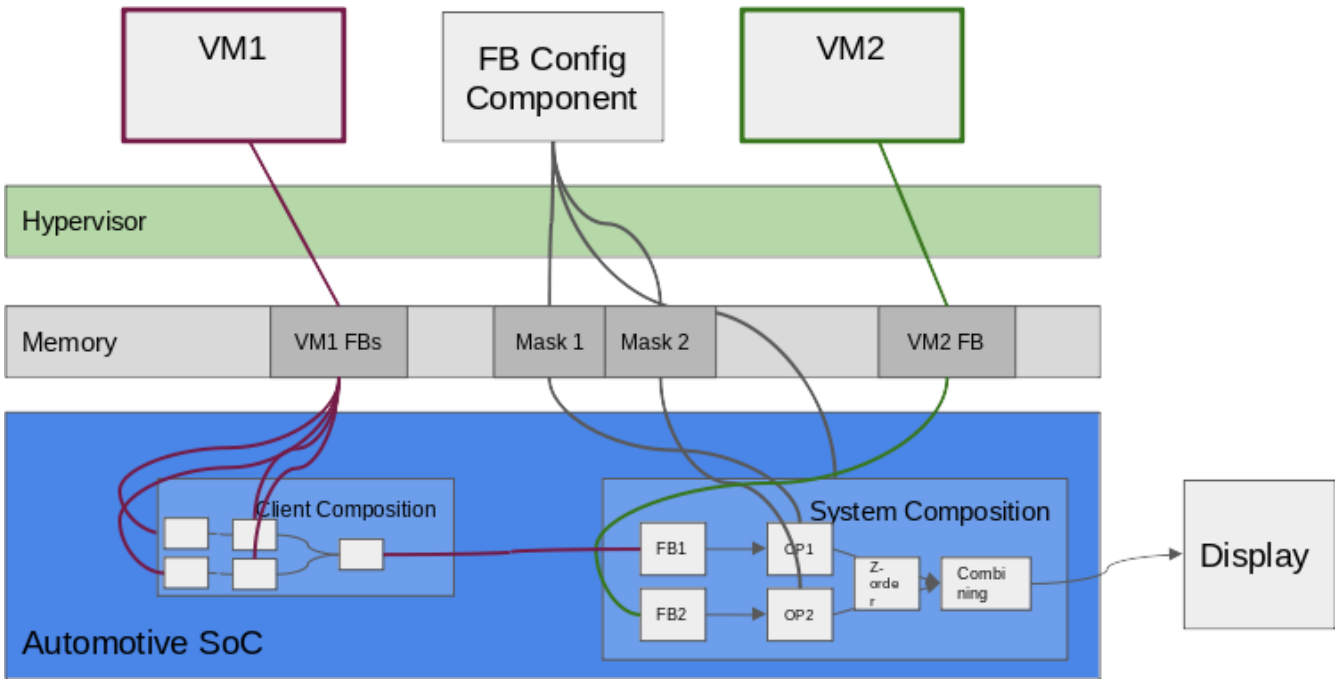
Virtualization approaches for framebuffer handling (for the composition of the final display picture) may exist independently from the 3D object calculations.

These are the basic considerations:

- Need to be hardware-provided (software-based composition is too slow)
- It is generally safe to assume that modern hardware has this
- GPU and Display Composer hardware could also be used to implement composition work when there isn't dedicated support for composition.
- It requires one (logical) composition engine per (logical) display
- Composition requirements:
- At least one framebuffer for each client/VM
- Separation of framebuffer and mask description
- Mask description must be independently controllable by HV

- Requirement for mask? 8-bit alpha channel? 1-bit?
- Nested composition?
- This would allow a guest to use a composition engine as well, e.g. video playing, overlay

Basic design of framebuffer composition in virtualized systems:



AVPS Requirements:

{AVPS-v2.0-26} The virtual platform shall support separating graphics output from VMs into different outputs, including a guarantee that VMs cannot view or modify the other VM's output (security/safety property):

- The requirement is only applicable if the hardware has support for multiple display outputs.

{AVPS-v2.0-27} If there is a single output, it shall still be possible to assign one out of several VMs to this display output, and the guarantee that other VMs cannot read or modify the output shall still apply.

{AVPS-v2.0-28} The virtual platform shall have the ability to compose graphics from several VMs into the final output:

- The requirement is only applicable if the SoC has the corresponding support for hardware composition.

{AVPS-v2.0-29} The Virtual Platform shall implement support for defining the policy for combining the graphics, including masking, considering the capabilities provided by the hardware.

- The capabilities of the image combination (simple masking / alpha blending / other) shall follow the hardware capabilities.

2.8.4 Additional output and safe-rendering features

Discussion:

Some hardware platforms include additional framebuffer and display output features.

These are often tailored towards so-called safe rendering and are intended to guarantee that the safety-critical graphics (could be tell-tales in cluster display) is guaranteed to be displayed correctly (and/or it is always noticed and reported if for some reason the graphics cannot be displayed).

These features may guarantee the graphics output by letting a checksum follow the bitmap data and check it at the end of the pipeline as near to the display as possible, or it may have more advanced features that can compare actual output to the intended output with allowance for minor detail changes.

It is strongly recommended for the Virtual Platform implementation to support the underlying hardware features for safety, but since the capabilities are often hardware-dependent we do not include more detailed requirements on the virtual platform implementation.

AVPS Requirements:

No platform requirements at this time. Each product implementation ought to carefully plan its own requirements in this area.

2.9 Audio

Discussion:

There is a pending proposal to the next VIRTIO specification for defining the audio interface. It includes how the Hypervisor can report audio capabilities to the guest, such as input/output (microphone/speaker) capabilities and what data formats are supported when sending audio streams.

Sampled data is expected to be in PCM format, but the details are defined such as resolution (number of bits), sampling rate (frame rate) and the number of available audio channels and so on.

Most such capabilities are defined independently for each stream. One VM can open multiple audio streams towards the Hypervisor. A stream can include more than one channel (interleaved data, according to previous agreement of format).

Noteworthy is that this virtual audio card definition does not support any controls (yet). For example, there is no volume control in the VIRTIO interface, so each guest basically does nothing with volume and mixing/priority is somehow implemented by Hypervisor layer (or companion VM, or external amplifier, or…) or software control (scaling) of volume would have to be done in the guest VM through user-space code doing this.

It might be a good idea to define a Control API to set volume/mixing/other on the hypervisor side. In a typical ECU, the volume mixing/control might be implemented on a separate chip, so the actual solutions vary.

Challenges include the real-time behavior, keeping low latency in the transfer, avoiding buffer underruns, etc. Determined reliability may also be required by some safety-critical audio functions and the separation of audio with varying criticality is required, although sometimes this is handled by preloading chimes/sounds into some media hardware and triggered through another event interface.

State transitions can be fed into the stream. Start, Stop, Pause, Unpause. These transitions can trigger actions. For example, when navigation starts playing, you can lower the volume of media.

Start means start playing the samples from the buffer (which was earlier filled with data) (and opposite for input case). Pause means stop at the current location, do not reset internal state, so that unpause can continue playing at that location.
There are no events from the virtual hardware to the guest because it does not control anything. It is also not possible to be informed about buffer underrun, etc.

A driver proof of concept exists in the OpenSynergy GitHub, and an example implementation in QEMU already. Previously QEMU played audio by hardware emulation of a sound card, whereas this new approach is using VIRTIO.

Potential future requirements:

[PENDING] If virtualized audio is implemented it MUST implement the VIRTIO-sound standard according to [VIRTIO-SND].

There are several settings / feature flags that should be evaluated to see which ones shall be mandatory required on an automotive platform.

2.10 IOMMU Device

Discussion:

A Memory Management Unit (MMU) provides virtual memory and allows an operating system to assign specific memory chunks to processes and thus allows building memory protection between processes.

An IOMMU provides similar means for hardware devices other than the CPU cores. A device which can do Direct Memory Accesses (DMA) can access and modify memory on its own. If the device is driven by an untrusted software component, e.g. an untrusted VM, or the device itself is not trusted, the hypervisor needs hardware means to guard the memory accesses of the device such that it can only access the memory areas which it is supposed to access. Guarding device memory accesses is also useful for potentially malfunctioning or misbehaving devices, e.g., due to bugs in their firmware or electrical glitches, such that the system is protected and can react accordingly. In this regard an IOMMU acts as an additional line of protection in a system, especially valuable in safety-conscious environments.

Overall, an IOMMU provides the kind of protection and separation for devices required for reliability, functional safety and cyber-security.

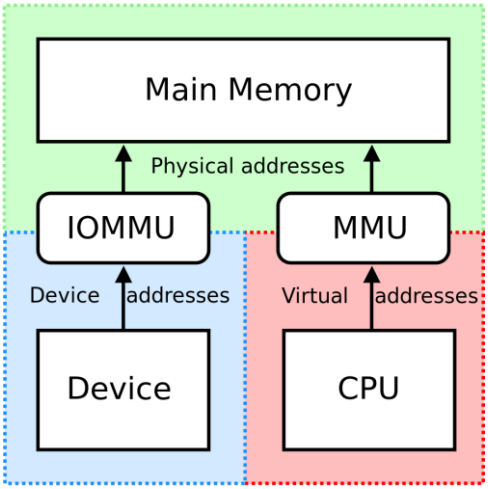


Image from Wikimedia Commons, License: Public Domain (info)

In practice this means that each DMA-capable device needs to be guarded by an IOMMU-unit that filters memory accesses done by this device. This also includes bus systems, such as PCIe, that are integrated in the system and requires that each such device on this bus has its own IOMMU-unit so that each of those devices can be potentially given to different untrusted VMs.

Devices that are virtualization-aware, i.e. that can provide separate distinct interfaces for VMs such as virtual functions (VFs) in SR-IOV, need an IOMMU-unit for each VF. Devices that are not virtualization-aware but shall be used among distrusting VMs must be multiplexed by the hypervisor such that the hypervisor ensures that only one VM can access the device at a time. This includes reprogramming the IOMMU settings for these devices on each switch of a VM and reloading the device configuration for the VM to be switched to. Also this likely implies adjustments to the VM scheduler in order not to switch to a new VM until the device really finishes work (completes memory accesses) for the current VM. Examples can be devices such as accelerators and DSPs. Due to the complex implementation of this switching it is recommended to only do this for coarse grained switching use-cases, or to implement a multiplexing of the device on a higher level.

Since there are different implementations of hardware mechanisms to protect the platform, this specification requires generically that the platform and hypervisor provide a solution for memory protection for co-processors, PCIe-devices, DMA-capable hardware blocks and other similar devices, such that all devices that can be configured, controlled, or programmed by untrusted VMs running on the general-purpose CPUs, are also limited to only the memory that the controlling VM shall have access to.

Generally, platforms need to be built such that IOMMUs are placed in front of each memory-accessing device in a granularity allowing to pass devices to different VMs. This includes interconnects such as PCIe where multiple independent devices can be hosted. The DMA-capable devices which are physically connected to the same IOMMU-unit (a situation where the same IOMMU context is shared between multiple devices) must not be assigned to different VMs. If such devices are present in the platform and need to be used for hardware pass-through they can only be assigned to the same VM to avoid security issues.

2-stage IOMMUs, or controlling IOMMU-contexts from the guest

In a basic implementation, an IOMMU is transparent for the guest VM, as the IOMMU is solely controlled by the hypervisor and configured for all devices a VM has access to. Any misconfiguration of devices, or misbehavior, or malicious behavior will be detected by the IOMMU and handled by the hypervisor accordingly.

However, an IOMMU can be useful for an operating system kernel itself, and in virtualization contexts this would require that an IOMMU could also be used and programmed in the guest VM context. Some IOMMUs on Arm can provide two separate stages of address translation (e.g. Renesas IPMMU-VMSA and some versions of Arm SMMU). This makes it possible to configure the IOMMU in way where the first context (page table) is used for stage-1 address translation in the guest (VA-IPA) and the second context (page table) is used for stage-2 address translation (IPA-PA) in the hypervisor. In effect, both the hypervisor and the guest VM can program the IOMMU.

This requires some emulation by the hypervisor if the IOMMU control registers for both stages are located within the same memory page (4K). While the page-table can be managed by the guest exclusively, the access to control registers must be trapped by the hypervisor and properly emulated. When control registers are shared we don’t want to let guests control IOMMU without validating the operations or an untrusted guest would be able to disable the IOMMU in order to bypass it, or re-program it to use for example the context of another VM.

Finally, in the case the IOMMU does not offer two stages or there is no corresponding support in the hypervisor to emulate guest accesses to the IOMMU control registers, then the hypervisor can offer the guest a fully emulated (virtual) IOMMU and let the VM program that via the VIRTIO-IOMMU protocol. This is usually recommended over a 2-stage emulation.

Some uses of a 2-stage IOMMU are:

The guest can limit the access of a device to an even smaller part of its own memory as part of making its software more resistant and detecting bugs in the guest system software.

Provide devices contiguous memory that is built out of physically scattered memory pages of VM memory.

Access devices that are limited in address range, such as 32bit devices on 64bit hosts. With a 2-stage IOMMU the guest can selectively assign those memory pages to the 32bit device that shall be used and the guest can adapt this at runtime according to its buffer allocation strategy. This is generally useful but other

options exist when the hypervisor can give the guest multiple areas of physical memory where one of those areas is within the reach of the device and the guest allocates buffer this device out of this area.

Left out of scope of this specification are use-cases where guest-programmable IOMMUs are used for other hybrid implementations. One example is an implementation that adds virtualization to Linux containers, such as the Kata Containers project.

It was also decided that nested virtualization is not a required feature in the context of this specification.

In conclusion, AVPS therefore only states an optional requirement for reprogramming the IOMMU from the guest because the uses for second-stage IOMMU summarized above are primarily for development support, may have other workarounds, or is needed for nested virtualization which we do not require. An optional requirement applies some conditions if this optional feature is included.

Some SoC solutions have additional memory-related features to support memory and I/O protection in security and or safety contexts. Bus-masters and bus-slaves (for example connected to AXI and peripheral buses) can be assigned to security and safety groups. Examples of bus-masters are System/Application CPU, Realtime CPU, GPU. Access limitations are set up to control memory access between groups. For example you could allow only the Realtime CPU to access CAN. These controls may be further described in a future version of the specification.

AVPS Requirements:

{AVPS-v2.0-30} The virtualization system must protect the system from untrusted, malfunctioning or buggy devices or devices driven by untrusted VMs.

{AVPS-v2.0-31} Support for a guest-programmable IOMMU is an optional feature with regards to this specification.

{AVPS-v2.0-32} If the virtual platform implements guest-control of IOMMU then:

- It shall use [VIRTIO-IOMMU]

or:

- If the hardware has support for two stage IOMMU the Virtual Platform may provide guest programming of a separate stage instead if the hypervisor emulates access to IOMMU control registers, if this is required to ensure full separation and avoiding interference between VMs.

Note that a chapter on DSP / co-processors may have additional requirements related to IOMMU.

2.11 USB

Discussion:

The AVPS working group and industry consensus seems to be that it is difficult to give concurrent access to USB hardware from more than one operating system instance, but we elaborate on the possibilities and needs in this chapter. It turns out that some research and implementation has been done in this area, but at this point it is not certain how it would affect a standard platform definition. In any case, the discussion section provides a lot of thinking about both needs and challenges.

[VIRTIO] does not in its current version mention USB.

The USB protocol has explicit host (master) and device (slave) roles, and communication is peer-to-peer only and never one-to-many. We must therefore always be clear on which role we are discussing when speaking about a potential virtual device:

Virtualizing USB Host Role

Concurrent access to a host controller would require creating multiple virtual USB devices (here in the meaning of virtual hardware device, not the USB device role), that are mapped onto a single hardware implemented host role, which i.e. a single USB host port. To make sharing interesting we first assume that a USB Hub is connected so that multiple devices can be attached to this shared host. Presumably, partitioning/filtering of the tree of attached devices could be done so that different virtual hosts are seeing only a subset of the devices. The host/device design of the USB protocol makes it very challenging to have more than one software stack playing the host role. When devices connect, there is an enumeration and identification procedure implemented in the host software stack. This procedure cannot have multiple masters. At this time, considering how USB host can be virtualized is an interesting theoretical exercise but value trade-off does not seem to be there, despite some potential ways it might be used if it were possible (see use-case section). We don't rule out the possibility of research into this changing the perception, however.

Virtualizing USB Devices

This could possibly mean two things. First, consider a piece of hardware that implements the USB-device role, and that hardware runs multiple VMs. Such virtualization seems next to nonsensical. A USB-device tend to be a very dedicated hardware device with a single purpose (yes, potentially more than one role is possible, but they tend to be related). Implementing the function of the USB-device would be best served by one system (a single Virtual Machine in a consolidated system). Thus, at most it seems that pass-through is the realistic solution.

The second interpretation is the idea of allowing multiple (USB host role) VMs to concurrently use a single actual USB-device hardware. This is difficult due to the single-master needs for enumerating and identifying that device. It is rather the higher-level function of the device (e.g. file storage, networking, etc.) that may need to be shared but not the low-level hardware interaction. Presumably, therefore a single VMs must in practice reserve the USB device hardware during its use and no concurrency is expected to be supported. Also here, research may show interesting results, but we saw little need to delve into it at this time.

Use cases and solutions

There are cases to be made for more than one VM needing access to a single USB device. For example, a single mass-storage device (USB memory) may be required to provide files to more than one subsystem (VM). There are many potential use cases but just as an example, consider software/data update files that need to be applied to more than one VM/guest, or media files being played by one guest system whereas navigation data is needed in another.

During the writing of this specification we found that some research had into USB virtualization but there was not time to move that into a standard.

After deliberation we have decided for the moment to assume that hypervisors will provide only pass-through access to USB hardware (both host and device roles)

USB On-The-Go(tm) is also left out of scope, since most automotive systems implement the USB host role only, and in the case a system ever needs to have the device role it would surely have a dedicated port and a single operating system instance handling it.

A general discussion for any function in the virtual platform is whether pass-through and dedicated access is to be fixed (at system definition/compile time, or at boot time), or possible to request through an API during runtime.

The ability for one VM to request dedicated access to the USB device during runtime is a potential improvement and it ought to be considered when choosing a hypervisor. With such a feature, VMs could even alternate their access to the USB port with a simple acquire/release protocol. It should be noted of course that it raises many considerations about reliability and one system starving the other of access. Such a solution would only apply if policies, security and other considerations are met for the system.

The most likely remaining solution to our example of exposing different parts of a file collection to multiple VMs is then that one VM is assigned to be the USB master and provide access to the filesystem (or part of it) by means of VM-to-VM communication. For example, a network file system such as NFS or any equivalent solution could be used.

Special hardware support for virtualization

As noted, it seems likely implementing protocols to split a single host port between multiple guests is complicated. This applies also if the hardware implements not only host controllers but also a USB-hub. In other words, when considering the design of SoCs to promote or support USB virtualization, it seems a more straightforward solution to simply provide more separate USB hardware devices on the SoC (that can be assigned to VMs using pass-through), than to build in special virtualization features into the hardware. That does not solve the use case of concurrent access to a device but as we could see there are likely software solutions that are better.

AVPS Requirements:

The following requirements are limited and expected to be increased in the future, due to the challenges mentioned in the Discussion section and that more investigation of already performed work (research papers, etc.) needs to be done.

Configurable pass-through access to USB devices.

{AVPS-v2.0-33} The hypervisor MUST provide statically configurable pass-through access to each USB host controller.

Resource API for USB devices

{AVPS-v2.0-34} The hypervisor MAY optionally provide an API/protocol to request USB access from the virtual machine, during normal runtime.

⚠ The configuration of pass-through for USB is yet not standardized and for the moment considered a proprietary API. This is a potential for future improvement.

Potential for future work exists here.

2.12 Automotive networks

This chapter covers some traditional in-car networks and buses, such as CAN, FlexRay, LIN, MOST, etc., which are not Ethernet TCP/IP style networks treated in the Standard Networks chapter.

2.12.1 CAN

Discussion:

The AVPS working group has found and discussed some work related to virtualizing CAN. A proposal exists named VIRTIO-can, but this is not in the VIRTIO standard: https://github.com/ork/VIRTIO-can and other research has been published as papers. For further insight, refer to the GENIVI Hypervisor Project group home page.

Like many automotive networks, it seems likely that a system may separate the responsibility for this communication into either a dedicated separate core in a multi-core SoC, or to a single VM, and then forward the associated data to/from other VMs from that single point.

CAN might be worth special consideration due to that some virtualization work has been presented.

2021/April: A formal VIRTIO-CAN proposal was sent to the VIRTIO development mailing list, which is worth tracking for the future.

AVPS Requirements:

We do not specify any requirements at this time since there is no obviously well adopted standard, nothing has been accepted into upstream specifications, and we have not yet had enough stakeholders to agree that the AVPS should put forward a single defined standard.

However, this requirement may be updated if VIRTIO-CAN gets traction.

Potential for future work exists here.

2.12.2 Local Interconnect Network (LIN)

Discussion:

LIN is a serial protocol implemented on standard UART hardware. For that reason, we assume that the standard way to handle serial hardware in virtualization is adequate.

Like many automotive networks, it also seems likely that a system may separate the responsibility for this communication into either a dedicated separate core in a multi-core SoC, or to a single VM, and then forward the associated data to/from other VMs from that single point.

Special consideration for virtualizing the LIN bus may therefore seem unnecessary, or not worth the effort now.

Reports of design proposals or practical use of LIN in a virtualized environment are welcome to refine this chapter.

AVPS Requirements:

-> Refer to any requirements given on serial devices.

2.12.3 FlexRay

Discussion:

FlexRay™ has not been studied in this working group.

Like many automotive networks, it seems likely that a system may separate the responsibility for this communication into either a dedicated separate core in a multi-core SoC, or to a single VM, and then forward the associated data to/from other VMs from that single point.

Device virtualization for the FlexRay bus itself may therefore seem unnecessary, or not worth the effort now.

Reports of design proposals or practical use of FlexRay in a virtualized environment are welcome, in order to refine this chapter.

AVPS Requirements:

None at this time.

2.12.4 CAN-XL

Discussion:

CAN-XL is still in development. We welcome a discussion with the designers on how or if virtualization design should play a part in this, and how the Automotive Virtual Platform definition can support it.

AVPS Requirements:

None at this time.

2.12.5 MOST

Discussion:

Media Oriented Systems Transport (MOST) has not been studied by the AVPS working group.

Reports of design proposals or practical use of MOST in a virtualized environment are welcome, in order to refine this chapter.

AVPS Requirements:

None at this time.

2.13 Watchdog

Discussion:

A watchdog is a device that supervises that a system is running by using a counter that periodically needs to be reset by software. If the software fails to reset the counter, the watchdog assumes that the system is not working anymore and takes measures to restore system functionality, e.g., by rebooting the system. Watchdogs are a crucial part of safety-concerned systems as they detect misbehavior and stop a possibly harming system.

In a virtualized environment, the hypervisor shall be able to supervise that the guest works as expected. By providing a VM a virtual watchdog device, the hypervisor can observe whether the guest regularly updates its watchdog device, and if the guest fails to update its watchdog, the hypervisor can take appropriate measures to ensure a possible misbehavior and to restore proper service, e.g., by restarting the VM.

While a hypervisor might have non-cooperating means to supervise a guest, being in full control over it, using a watchdog is a straight-forward and easy way to implement a supervision functionality. An implementation is split in two parts, one being the in the hypervisor, the device, and another in the guest operating system, a driver for the device offered by the hypervisor. As modifying and adding additional drivers to an operating system might be troublesome because of the effort required, it is desirable to use a watchdog driver that is already available in guest operating systems.

Fortunately, there are standard devices also for watchdogs. The Server Base System Architecture [SBSA] published by ARM defines a generic watchdog for ARM systems, which also has a driver available in the popular Linux kernel and thus only requires hypervisors to provide a virtual generic watchdog device according to SBSA's definition (device compatible: "arm,sbsa-gwdt"). The specification offers appropriate actions in case the guest fails to update the watchdog.

We therefore recommend hypervisors to implement the watchdog according to the generic watchdog described in SBSA, not only on ARM systems but regardless of the hardware architecture used in the system.

AVPS Requirements:

`{AVPS-v2.0-35}` The platform MUST implement a virtual hardware interface to the hardware watchdog, following the generic watchdog described in Server Base System Architecture 6.0 [SBSA]

2.14 Power and System Management

Discussion:

Power and system management can be an important part of the system, and takes care of management of the virtual platform as well as passing information to the hypervisor if the VM accesses devices directly. Tasks include:

- Shutdown and reset of the virtual platform
- Enabling and disabling of cores for multi-core VMs
- Informing the hypervisor on performance requirements
- Suspend-to-RAM of the virtual platform
- Inform the hypervisor on secondary/indirect peripheral use, such as peripheral clocks and peripheral power-domains

Since the hypervisor is responsible to keep the overall system in a secure and safe state, it must offer an arbitration service that is able to take decisions:

- Deny or accept a guest requests depending on the status the system in general and of other guests
- Intercept power management events (power failures, user interactions, etc.) and forward them to the relevant guests safely and in the correct order

To facilitate portability of VMs between hardware platforms, the VMs shall use a platform-independent API as much as possible.

On Arm systems, there exist standardized interface for platform and power management topics:

Power State Coordination Interface (PSCI): Offers interfaces for suspend/resume, enabling/disabling cores, secondary core boot, system reset and power-down as well as core affinity and status information.

System Control and Management Interface (SCMI): offers a set of operating-system independent interfaces for system management, including power domain management, performance management of system components, clock management, sensor management and reset domain management.

The interfaces are built in a way allowing a hypervisor to implement those interfaces for guests, and possibly arbitrating and managing requests for multiple guests.

When a hypervisor offers features regarding power and system management to virtualization guests on the Arm platform, the hypervisor shall offer virtual PSCI and SCMI interfaces to the guest.

For other architectures, the landscape is diverse, such that this specification can only recommend using the standard mechanisms used on those architectures and implement appropriate support in the hypervisor environment to support guests.

For Intel based systems, it is typical for the virtual platform to expose a virtual interface that follows ACPI. There are similar concepts in ACPI, PSCI and SCMI but creating a unified interface is not realistic at this point. Therefore the approach in this specification is to provide different requirements for different hardware platforms. So far, the specification has firm requirements on Arm-based systems only, and more analysis would be needed to write firm requirements for x86-64 (Intel/AMD), RISC-V, MIPS, PowerPC, SH (e.g. V850 microcontrollers) and other architectures.

A definition of SCMI over VIRTIO was recently merged into the master branch of the specification development, see [VIRTIO-SCMI]. Future requirements are likely to reference this specification as required support.

AVPS Requirements:

{AVPS-v2.0-36} AVPS requires that compatible hypervisors on the Arm architecture implement functionality using PSCI and SCMI (ref: [SCMI]) when such a feature is offered by the hypervisor.

- If these features are included, they shall fulfil the specifications as written:

Required:

- VM-Core on/off: Arm: PSCI
- VM-reset, VM-poweroff: Arm: PSCI
- Idle/sleeping states: Arm: PSCI
- Suspend-to-ram: Arm: PSCI

Optional:

- CPU Performance Management: Arm: SCMI Clock, Power domains, Performance domains, Reset domains: Arm: SCMI

Required PSCI Interface: v1.1

Required SCMI Interface: v2.0 or later

Potential for future work exists (for additional CPU architectures) here.

2.15 GPIO

Discussion:

GPIOs are typically simple devices that consist of a set of pins that can be operated in input or output mode. Each pin can be either on or off, sensing a state in input mode, or driving a state in output mode. For example, GPIOs can be used for sensing buttons and switching, or driving LEDs or even communication protocols. Hardware-wise a set of pins forms a GPIO block that is handled by a GPIO controller.

In a virtualization setup, a guest might want to control a whole GPIO block or just single pins. For a GPIO block that is provided by a GPIO controller, the hypervisor can pass-through the controller so that the guest can directly use the device with its appropriate drivers. If pins on a single GPIO block shall be shared across multiple guests, or a guest shall not have access to all pins of a block, the hypervisor must multiplex access to this block. Since GPIO blocks are rather simple devices, the platform specification recommends emulating a widely used GPIO block and use the unmodified drivers already existing in common operating systems.

Usage of GPIOs for time-sensitive use, such as “bit-banging”, is not recommended because it requires a particular scheduling of the guest. For such cases, the virtual platform should provide other suitable means to implement a driver for the functionality that is being emulated by bit-banging.

Future Outlook:

There is a proposal described in the ACRN project for VIRTIO transport [REF] that implementations may consider. If the support becomes officially proposed in the VIRTIO specification and Linux mainline drivers appear then this may be considered as a platform requirement in a later version. There is an independently proposed Linux driver to consider [REF].

AVPS Requirements:

{AVPS-v2.0-37} The hypervisor/equivalent shall support configurable pass-through access to a VM for digital general-purpose I/O hardware

{AVPS-v2.0-38} The platform may provide emulation of a widely used GPIO block which already has drivers in Linux and other kernels

A future specification version may require a specific emulation API (e.g. VIRTIO, when it exists) for better portability. Potential for future work exists here.

2.16 Sensors

Discussion:

Most of what are considered sensors in a car are deeply integrated with electronics or associated with dedicated ECUs and accessing their data may already be defined by the protocols that the ECUs or electronics provide and as such the protocol is unrelated to any virtual platform standardization.

However, as SoCs become more integrated there are often a variety of sensors implemented on the same silicon and directly addressable. As such they may be candidates for a device sharing setup. Sensors such as ambient light, temperature, pressure, acceleration, IMU Inertial Measurement Unit (rotation), tend to be built into SoCs because they share similarities with mobile phone SoCs that require these.

The Systems Control Management Interface (SCMI) specification, ref: [SCMI], defines a kind of protocol to access peripheral hardware. It is usually spoken from general CPU cores to the system controller (M3 core responsible for clock tree, power regulation, etc.) via a hardware mailbox.

Since this protocol is already defined and suitable for communication between, it would be possible to reuse it for accessing sensor data quite independently of where the sensor is located.

The Systems Control Management Interface (SCMI) specification does not specify the transport, suggesting hardware mailboxes but acknowledging that this can be different.

Access to the actual sensor hardware can be handled by a dedicated co-processor or the hypervisor implementation and provide the sensor data through a communication protocol.

For sensors that are not appropriate to virtualize we instead consider hardware pass-through.

The SCMI specified protocol was not originally defined for the virtual-sensor purpose but describes a flexible and an appropriate abstraction for sensors. It is also appropriate for controlling power-management and related things. The actual hardware access implementation is according to ARM

offloaded to a "Systems Control Processor", but this is an abstract definition. It could be a dedicated core in some cases and in others not.

An IIO (Industrial I/O subsystem) driver has been merged into the Linux Kernel (5.13) and this is based on SCMI (version 3). [SCMI-IIO]

The 3.0 version of SCMI requires timestamps which we think is critical and therefore SCMI 3.0 is required here for sensor data, instead of SCMI 2.0.

AVPS Requirements:

`{AVPS-v2.0-39}` For sensors that need to be virtualized the SCMI protocol MUST be used to expose sensor data from a sensor subsystem to the virtual machines.

Required SCMI version for sensor interfaces: 3.0 [SCMI3]

2.17 Cameras

Potential in-car architectures:

- Separate Camera-ECU communicating to head unit via some network
- Head unit is connected directly to the camera sensor
- Cameras used entirely for object recognition (no need to connect to an ECU that drives a user-facing display)

Also integrated image processors and other camera-dedicated silicon is likely to be increasing (also built into generic SoCs), to simplify advanced calculation such as intelligent object/obstacle recognition, and stitching multiple cameras into "bird's eye view", etc.

The existing [VIRTIO-video] specification draft defines only encoder and decoder capabilities. The camera capability has been proposed to be defined by an additional feature flag, but the community has not agreed on this.

Different pieces of camera hardware may have different capabilities and limitations, such as how many settings can be set individually for each sensor. Some settings are thus forced to be identical for all, or for groups of cameras. Some of the complexities include, but are not limited to:

Camera sensors might have different characteristics: the serial interface bandwidth can differ for different connected sensors to take potential higher and lower resolution capture requirements into account, but in other cases, hardware control (resolution, frame rate, ...) are set in groups, so you can only set the same setting for all, or a group of cameras.

Since all the camera sensors are usually routed to a single DMA engine, it is then up to the paravirtualized solution to provide buffer sharing mechanism to receive buffers from the driver and fill those with data the same way it is done for the video codec sharing.

Image processing / stitching is often built into the camera device (or ECU), so it is unlikely to be modified by a virtual hardware / hypervisor layer.

Some stateless cameras allow reconfiguration all the way down to individual frames, which would theoretically allow different VMs to have different configurations while sharing the same camera. Whenever possible, the hypervisor should utilize this capability to provide seamless access to the camera sensors and their settings in the most flexible way that is possible. When the camera device does not support the stateless operation mode, the hypervisor could emulate this mode but restrict access to the camera to only one or more clients at a time according to the limitations that the real hardware has regarding the individual sensor configuration.

For future reference and study, Xen has camera virtualization support. Webcam use on a laptop/workstation is the likely driving use case, as opposed to the multiple cameras of an automotive system. It is defined using a Xen specific protocol which is not specified by VIRTIO and has been accepted by the Xen and Linux kernel community. At the time of writing, the front-end driver is created but not yet merged in mainline Linux.

Above, we argued against advanced emulation of camera capabilities in the hypervisor layer, and thus the hypervisor is expected to put limits on, or negotiate, and ultimately allow only the configurability that the hardware allows for.

AVPS Requirements:

Since we are not yet aware that a proposal similar to “VIRTIO-camera” has started yet, no requirements are defined at this time.

Potential for future work exists here.

2.18 Media codecs.

This chapter covers hardware support devices for encoding and decoding of compressed media formats (audio and video) and the potential of VMs to share these hardware capabilities.

Typically there are two interface types defined for multimedia codecs. A stateful interface does not require the user to maintain additional information, like the amount of the encoded data to be processed at the next step and corresponding format dependent metadata, to perform buffer processing, it is done internally by the hardware and by the driver. A stateless interface in turn implies that on each processing step the user is expected to maintain the current state of operation and provide it to the device on a per frame basis to advance the processing pipeline.

For example, for the decoder use case the stateless interface requires a lot of stream processing steps (like metadata parsing) to be done by the user in software. This means, in case the actual hardware is stateful, there would be not enough data to perform any real operation on the multimedia stream on the hypervisor side. On the other hand, if the paravirtualized multimedia device implements the stateful interface, it should not be a problem to handle the data and do any required parsing on the hypervisor side if the real hardware has either stateful or stateless interface.
Therefore the virtual device interface should be stateful to be possible to implement on all hardware variants.

The proposed [VIRTIO-video] standard is designed with the stateful interface in mind. It defines a command-response set to report and/or negotiate capabilities, stream formats, and various codec-specific settings.

The [VIRTIO-video] specification draft defines encoder and decoder capabilities. The actual codec hardware performs the encoding/decoding operation. The virtual platform provides concurrent or arbitration between multiple guests. The HV can also enforce some resource constraints in order to better share the capabilities between VMs.

AVPS Requirements:

[VIRTIO-video] is designed to define a virtual interface to video encoding and decoding devices.
Since the VIRTIO-video proposal isn't ratified yet, no requirements are defined at this time but it is likely in a later release of the document.

- [PENDING] If a system requires video codec sharing, it MUST be implemented according to the VIRTIO-video requirements specified in [VIRTIO-X.X]

Potential for future work exists here.

2.19 Cryptography and Security Features

Sharing of crypto accelerators.

On ARM, crypto accelerator hardware is often only accessible from TrustZone, and stateful as opposed to stateless. Both things make sharing difficult.

A cryptography device exists in VIRTIO (intended to model crypto accelerator for ciphers, hashes, MACs, AEAD) [VIRTIO-CRYPTO]

RNG and entropy

VIRTIO-entropy (called virtio-rng inside Linux implementation) is preferred because it is a simple and cross-platform interface.

Some hardware implements only one RNG in the system and it is in TrustZone. It is inconvenient to call APIs into TrustZone in order to get a value that could just be read from the hardware but on those platforms, it is the only choice. While it would be possible to implement VIRTIO-entropy via this workaround, it is more convenient to make direct calls to TrustZone.

The virtual platform is generally expected to provide access to a hardware-assisted high-quality random number generator through the operating system's preferred interface (/dev/random device on Linux)

The virtual platform implementation should describe a security analysis of how to avoid any type of side-band analysis of the random number generation.

2.19.1 Random Number Generation

Discussion:

Random number generation is typically created by a combination of a true-random and pseudo-random implementations. A pseudo-random generation algorithm is implemented in software. "True" random values may be acquired by an external hardware device, or a built-in hardware (noise) device may be used to acquire a random seed which is then further used by a pseudo-random algorithm. VIRTIO specifies an entropy device usable from the guest to acquire random numbers.

In order to support randomization of physical memory layout (as Linux does) the kernel also needs a good quality random value very early in the boot process, before any VIRTIO implementations can be running. The device tree describes the location to find this random value or specifies the value itself. The kernel uses this as a seed for pseudo-random calculations that decide the physical memory layout.

Traditionally a requirement for a “true” random generator is required but there is a lot of debate of whether this truly improves the situation compared to pseudo-random generators. In particular, it is harmful if too many processes “deplete” the true random generator to the extent that PRNGs cannot be reliably seeded, and applications may then effectively receive statistically less random numbers.

AVPS Requirements:

To support high-quality random numbers to the kernel and user-space programs:

`{AVPS-v2.0-40}` The Virtual Platform MUST offer at least one good entropy source accessible from the guest.

`{AVPS-v2.0-41}` The entropy source SHOULD be implemented according to VIRTIO Entropy device, chapter 5.4 [VIRTIO]

- To be specific, it is required that what h received from the implementation of the VIRTIO entropy device SHOULD contain only entropy.

To support memory layout randomization in operating systems that support it (Linux specifically):

`{AVPS-v2.0-42}` The virtual platform MUST provide a high-quality random number seed immediately available during the boot process and described in the hardware device tree using the name kaslr-seed (Ref: [Linux-DeviceTree-Chosen]) https://elixir.bootlin.com/linux/latest/source/Documentation/devicetree/bindings/chosen.txt

2.19.2 Trusted Execution Environments

Discussion:

Access to TrustZone and equivalent Trusted Execution Environments (TEE) is a feature that is frequently requested from the guest, so when legacy systems are ported from native hardware to a virtual platform, should not require significant modification of the software. Accessing the trusted execution environment should work in the exact same way as for a native system. This means it can be accessed using the standard access methods that are typically involved executing a privileged CPU instruction (e.g. SMC calls on ARM, equivalent on Intel). Another option used on some Intel systems is to run OPTEE instances, one per guest. The rationale for this is that implementations that have been carefully crafted for security (e.g.

Multimedia DRM) are unlikely to be rewritten only to support virtualization.

While it is possible to run a TEE as a guest VM on ARM, GlobalPlatform discourages this, because in this case the hypervisor will have access to TEE's memory, which is undesirable.

AVPS Requirements:

{AVPS-v2.0-43} Access to TrustZone and equivalent functions MUST work in the exact same way as for a native system using the standard access methods (SMC calls on ARM, equivalent on Intel).

2.19.3 Replay Protected Memory Block (RPMB)

Discussion:

The Replay Protected Memory Block (RPMB) provides a means for the system to store data to a specific memory area in an authenticated and replay protected manner. An authentication key information is first programmed to the device. The authentication key is used to sign the read and write made to the replay protected memory area with a Message Authentication Code (MAC). The feature is provided by several storage devices like eMMC, UFS, NVMe SSD, by having for one or more RPMB area.

Different guests need their own unique Strictly Monotonic Counters. It is not expected for counters to increase by more than one, which could happen if more than one guest shares the same mechanism. The RPMB key must not be shared with multiple guests and another concern is that RPMB devices may define maximum write block sizes, so it would require multiple writes if the data chunk were large, making the process no longer atomic from one VM. Implementing a secure setup for this is for now beyond the scope of this description, but it seems to require establishing a trusted entity that implements an access "server", which in turn accesses the shared RPMB partition.

Notably, most hardware includes two independent RPMB slots, which enables at least two VMs to use the functionality without implementing the complexities of sharing.

Some platforms offer virtualized RPMB support whereas some platforms instead use hardware passthrough to offer RPMB functionality to the TEE.

A proposal for Virtual RPMB was recently proposed on VIRTIO mailing list and is planned to be included in VIRTIO version 1.2, but it is unclear to this working group whether the proposal addresses all the implementation complexity, or leaves the solution open as unknown implementation details.

AVPS Requirements:

{AVPS-v2.0-44} If the platform provides virtualized replay protection, the device MUST be implemented according to the RPMB requirements specified in [VIRTIO-RPMB]

2.19.4 Crypto acceleration

Discussion:

VIRTIO-crypto standard seems to have been started primarily for PCI extension cards implementing crypto acceleration, although specification seems generic enough to support future (SoC) embedded hardware.

The purpose of acceleration can be pure acceleration (client has the key) or rather HSM purpose (such as key is hidden within hardware).

The implementation consideration is analogous to the discussion on RNGs. On some ARM hardware these are offered only within TrustZone and in addition the hardware implementation is stateful. It ought to be possible to implement VIRTIO-crypto also by delegation into TrustZone and therefore we require it also on such platforms however it should be understood that parallel access to this feature may not be possible, meaning that this device can be occupied when a guest requests it. This must be considered in the complete system design.

AVPS Requirements:

`{AVPS-v2.0-45}` If the virtual platform implements crypto acceleration, then the virtual platform MAY implement VIRTIO-crypto as specified in chapter 5.9 in [VIRTIO]. (ref: **[VIRTIO-CRYPTO])**

(NB This requirement might be a MUST later, if the hardware is appropriate, and optional for hardware platform platforms that are limited to single-threaded usage or other limitations. At that point a more exact list of required feature bits from VIRTIO should be specified.)

2.20 Supplemental Virtual Device categories

2.20.1 Text Console

Discussion:

While they may be rarely an appropriate interface for the normal operation of the automotive system, text consoles are expected to be present for development purposes. The virtual interface of the console is adequately defined by [VIRTIO]

Text consoles are often connected to a shell capable of running commands. For security reasons, text consoles need to be possible to shut off entirely in the configuration of a production system.

AVPS Requirements:

`{AVPS-v2.0-46}` The virtual interface of the console MUST be implemented according to chapter 5.3 in [VIRTIO]

`{AVPS-v2.0-47}` To not impede efficient development, text consoles shall further be integrated according to the operating systems' normal standards so that they can be connected to any normal development flow.

`{AVPS-v2.0-48}` For security reasons, text consoles MUST be possible to shut off entirely in the configuration of a production system.
This configuration MUST NOT be modifiable from within any guest operating system.

`{AVPS-v2.0-49}` It is also recommended that technical and/or process related countermeasures are introduced and documented during the development phase, to ensure there is no way to forget to disable these consoles.

2.20.2 Filesystem virtualization

Discussion:

This chapter discusses two different features, one being host-to-vm filesystem sharing and the other being VM-to-VM sharing, which might be facilitated by Hypervisor functionality.

The function of providing disk access in the form of a "shared folder" or full disk pass-through is a function that seems more used for desktop virtualization than in the embedded systems that this document is for. In desktop virtualization, for example the user wants to run Microsoft Windows in combination with a MacOS host, or to run Linux in a virtual machine on a Windows-based corporate workstation, or to try out custom Linux systems in KVM/QEMU on a Linux host, for development of new (e.g. embedded) systems. Host-to-VM filesystem sharing might also serve some purpose also in certain server virtualization setups.

The working group found little need for this host-to-vm disk sharing in the final product in most automotive systems, but we summarize the opportunities here if the need arises for some product.

[VIRTIO] mentions, very briefly, one network disk protocol for the purpose of hypervisor-to-vm storage sharing, which is 9pfs. 9pfs is a part of a set of protocols defined by the legacy Plan9 operating system. VIRTIO is very short on details and seems to be lacking even references to a canonical formal definition. The VIRTIO specification is thus complemented only by scattered information found on the web regarding the specific implementations (Xen, KVM, QEMU, ...). However, a research paper on VirtFS however has a more explicit proposal which is also 9P based -- see ref [9PFS]. This could be used as an agreed common basis.

9pfs is a minimalistic network file-system protocol that could be used for simple HV-to-VM exposure of file system, where performance is not critical.

9pfs has known performance problems however but running 9pfs over vsock could be an optimization option. 9pfs seems to lack a flexible and reliable security model which seems somewhat glossed over in the 9pfs description: It briefly references only "fixed user" or "pass-through" for mapping ownership on files in guest/host.

A more advanced network disk protocol such as NFS, SMB/SAMBA would be too heavy to implement in the HV-VM boundary, but larger guest systems (like a full Linux system) can implement them within the normal operating system environment that is running in the VM. Thus, the combined system could likely use this to share storage between VMs over the (virtual) network and in that case the hypervisor/virtual platform does not need an explicit implementation.

A recently proposed VIRTIO-fs [VIRTIO-FS] aims to “provide local file system semantics between multiple virtual machines sharing a directory tree”. It uses the FUSE protocol over VIRTIO, which means reusing a proven and stable interface and guarantees the expected POSIX filesystem semantics also when multiple VMs operate on the same file system.

Optimizations of VM-to-VM sharing will be possible by using shared memory as being defined in VIRTIO 1.2. File system operations on data that is cached in memory will then be very fast also between VMs.

As stated, it is uncertain if fundamental host-to-vm file system sharing is a needed feature in typical automotive end products, but the new capabilities might open up a desire to use this to solve use-cases that were previously not considering shared filesystem as the mechanism. We can envision something like software update use-cases that have the Hypervisor in charge of modifying the actual storage areas for code. For this use case, download of software might still happen in a VM which has advanced capabilities for networking and other operations, but once the data is shared with the HV, it could take over the responsibility to check software authenticity (after locking VM access to the data of course) and performing the actual update.

In the end, using filesystem sharing is an open design choice since the data exchange between VM and HV could alternatively be handled by a different dedicated protocol.

References:
VIRTIO 1.0 spec : {PCI-9P, 9P device type}.
Kernel support: Xen/Linux 4.12+ FE driver Xen implementation details
[VIRTIO-FS] (see reference section)
The VirtFS paper [9PFS]

Some other 9pfs-related references include:
(This is mostly to indicate the scattered nature of 9P specification. Links are not provided since we cannot now evaluate the completeness, or if these should be considered official specification or not).

A set of man pages that seem to be the definition of P9.

QEMU instructions how to set up a VirtFS (P9).

Example/info how to natively mount a 9P network filesystem.

Source code for 9pfs FUSE driver

The VirtFS paper [9PFS]

AVPS Requirements:

`{AVPS-v2.0-50}` If filesystem virtualization is implemented, then [VIRTIO-FS] MUST be one of the supported choices.

3 References

[Linux-DeviceTree-Chosen] How to pass explicit data from firmware to OS, e.g. kaslr-seed
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/devicetree/bindings/chosen.txt?h=v5.12

[RFC 2119] https://www.ietf.org/rfc/rfc2119.txt

[SBSA] Server Base System Architecture 7.0
https://developer.arm.com/docs/den0029/latest

[SCMI] System Control and Management Interface, v 2.0 or later if not otherwise specified

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0056b/index.html
(NOTE: this link now offers v3.0)

[SCMI3] System Control and Management Interface, v 3.0

http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0056b/index.html
or https://developer.arm.com/documentation/den0056/c/?lang=en

[SCMI-IIO] (Linux kernel driver)
https://lwn.net/Articles/844752/

[VIRTIO] Virtual I/O Device (VIRTIO) Version 1.1, Committee Specification 01, released 11 April 2019
https://docs.oasis-open.org/VIRTIO/VIRTIO/v1.1/VIRTIO-v1.1.html

[VIRTIO-CRYPTO]
https://docs.oasis-open.org/VIRTIO/VIRTIO/v1.1/cs01/VIRTIO-v1.1-cs01.html#x1-3500009

[VIRTIO-FS] https://VIRTIO-fs.gitlab.io/

[VIRTIO-GPU] Chapter 5.7 GPU-Device in VIRTIO 1.1

[VIRTIO-IOMMU] Chapter 5.13 (proposed in the following rendering of VIRTIO:)
http://jpbrucker.net/VIRTIO-iommu/spec/VIRTIO-v1.1+VIRTIO-iommu.pdf#1a8 with the separate description here: https://jpbrucker.net/VIRTIO-iommu/spec/VIRTIO-iommu-v0.13.pdf

[VIRTIO-SCMI] VIRTIO device for SCMI interface (future VIRTIO release)
https://github.com/oasis-tcs/VIRTIO-spec/commit/80b54cfd10a3e3d40eef532e7741ec50e63618b8
https://github.com/oasis-tcs/VIRTIO-spec/blob/master/VIRTIO-scmi.tex

[VIRTIO-SND] Likely to be part of VIRTIO 1.2 - currently merged to the master branch:
https://github.com/oasis-tcs/VIRTIO-spec/blob/master/VIRTIO-sound.tex

[VIRTIO-VIRGL]
https://github.com/Keenuts/VIRTIO-GPU-documentation/blob/master/src/VIRTIO-GPU.md

[VIRTIO-VULKAN]
https://gitlab.freedesktop.org/virgl/virglrenderer/-/milestones/2

Information on RPMB (using forwarding to hardware) https://xen-troops.github.io/papers/optee-virt-rpmb.pdf
https://lists.linaro.org/pipermail/tee-dev/2020-January/001413.html

[9PFS] VirtFS--A virtualization aware File System pass-through.
https://www.researchgate.net/publication/228569314_VirtFS--A_virtualization_aware_File_System_pass-through