Automotive Virtual Platform Specification

Version: 1.0 Beta, rev01

Contents

1 Introduction

This specification covers a collection of virtual device driver APIs. They 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.

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 present in the centralized and connected vehicle cockpit.*

As part of this, GENIVI focuses on interoperability technologies beyond the 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 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 for possible further refinement.

Automotive systems use software stacks with particular needs. Existing standards for virtualization sometimes need to be augmented 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. 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 s*ome* potential for shared implementation across guest **operating systems**.
- There is s*ome* potential for shared implementation across **hypervisors** with different license models.
- It can provide 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 a lot of synergy effects around the most direct effects of standardization. Similarly, there is potential for standardized **hypervisor runtime environments** that 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 future-proofing of designs, as well as lower risks when starting product development.

1.1 Specification outline

This specification, the AVPS, is intended to be immediately usable 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 chapter **General requirements** 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's 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* – execution environments whose virtual machines act like 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 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.

However, there are 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.

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. The platform should ideally be able to execute such CPU instructions without modifying the software.

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

1.3 Hardware pass-through

In a non-virtual system, 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 actual 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 that 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 in 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 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 is offered. 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 standardize also those APIs that give this seemingly “direct” access, but in a different form than the original hardware provides.

1.4 Draft specification state

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

1) 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.
2) 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.
3) 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.
4) 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 anyone!).

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 these 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 and all computer systems inside of vehicles, a.k.a. Electronic Control Units (ECUs).

There are no other assumptions about architecture at this time.

Virtualization implementation architectures

Comparing virtualization solutions can be difficult due to differing internal design. Sometimes these are real and effective differences whereas sometimes only different names are used, or different emphasis is placed on certain aspects when underlying principles may be similar.

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

Some other providers conversely state that their actual *hypervisor* is minimal and highlight especially the fact that the hypervisor purpose is only provides separation and scheduling of individual virtual machines – thus it implements a kind of “separation kernel”. The remaining features of such a design might then be delegated to dedicated VMs, either with unique privileges or considered identical to the “guest” VMs that we are concerned with when discussing system portability. Consequently, it is then those dedicated VMs that implement some of the API of the virtual platform, i.e. the APIs used by the “guest” VMs.

Others may use different nomenclature and highlight that parts of the virtual platform may have multiple privilege levels, “domains*”,* additional levels in addition to the simpler model of user space < OS kernel < Hypervisor.

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

At that point the design of the hypervisor/virtualization platform and the design of the full system (including guest VMs) tend to be inter-dependent, but the intention of the specification is to try to be independent of such design choices. Although other goals have been explained, starting with the obvious goal of portability of guest VMs should bring any concerns to the surface. If any implementation can follow the specification, then the actual technical design is free to vary.

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

Suffice to say that this specification strives to not prefer one approach or limit the design of the virtualization platforms. Some parts of the discussion sections may still speak about the “Hypervisor” implementing a feature, but it should then be interpreted loosely, so that it fits also designs that provide identical feature compatibility but may see this functionality delegated to a dedicated VM.

2.2 Conformance to specification

2.2.1 Optional features vs. optional requirements.

The intention of a platform 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 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 are 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 (unless this is clearly 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 hypervisor implementation (virtual platform) 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.

2.2.2 Virtualization support in hardware

When running on hardware that supports it, then the virtualization interfaces 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.

2.2.3 Hardware emulation

A virtualization platform may deal with the emulation of hardware features for various reasons. Lacking explicit support in the hardware, it may be the 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.

2.3 General system

2.3.1 Booting Guests

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 pointing to a location containing 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 might not realistically implement the EBBR protocol (e.g. some 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, and the virtual platform shall therefore provide it.

For legacy systems we expect:

Option 1) HV exposes the devices in the expected location (and behavior) of an existing legacy system. (Full virtualization)

Option 2) HV decides where to place device features and communicates that to the operating system. *This would be done by device-tree snippets (statically defined).*

In the future this specification could benefit from deciding on a firm standard for how the Hypervisor describes a hardware/memory map to legacy system guests, and associated recommendations for how to port legacy systems.

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. (i.e. the boot code could still be hard-coded, and it does not need to support a lot of runtime configuration) → see requirement above.

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

AVPS Requirements:

- Systems that support a dynamic boot protocol should implement (*the mandatory parts of**) EBBR
- Since EBBR allows either ACPI or Device Tree implementation, this can be chosen according to what fits best for the chosen hardware architecture.
- For systems that do not support a dynamic boot protocol (see discussion), the virtual hardware shall still be described (by the HV vendor) using a device tree format, so that implementers can program custom boot and setup code based on this formal description.

2.4 Common Virtual Device categories

2.4.1 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 establish 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 one can spot three relevant scenarios:

1. 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, one may think of a flash device where the flash translation layer (FTL) needs to be provided by software. This contrasts with, for example, MMC flash device, SD cards and USB thumb drives where the FTL is transparent to the software.
2. 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 use 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.

3. Features which call for an extension of the VIRTIO Block device driver standard. Whereas category 1 and 2 does not need an augmentation of the VIRTIO block device standard, a different story needs to be told, whenever such workarounds do not exist. An example of this is the erasing of blocks. The respective commands can only be emitted by the guest OS. The resulting TRIM commands need to be explicitly implemented in both the front-end as well as in the back-end driven by the hypervisor.

2.4.1.1 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 this together.

Typical automotive persistence requirements are:

1. 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 (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 to the 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 -- details will follow). The key underlying challenge is that there are well known limits to Flash memory technology being “worn out” after a certain number of write cycles.

2. 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 of this may differ and may 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 disappeared). This may allow the system to execute emergency write-through of critical data.

3. 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 inter-related and can be in conflict. They are only solved by limiting the usage of write-through requirement 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 file-systems, and block-level storage drivers, flash-memory controllers which in themselves (in hardware) have a number of 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. Furthermore, 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. We would encourage to continue this discussion and share well thought out design principles that can be reused and adapted, collaboration on the analysis methods and tooling to ensure this and discussing how standards may lock down the compatibility of these solutions.

2.4.1.2 Block Devices

Discussion*:*

While the comments in the introduction remain, 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 should 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 a 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.

One thing was found missing: VIRTIO might not provide the "total blocks written" data that is available in native systems, but it is arguable if guest needs to know this or not. Are there preventative measures / diagnostics that would benefit from this?

NOTE:

UFS devices provides 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 provides 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:

- The platform shall implement virtual block device according to chapters 5.2 in [VIRTIO].
- The platform must implement support for the VIRTIO_BLK_F_FLUSH feature flag.
- The platform must implement support for the VIRTIO_BLK_F_CONFIG_WCE feature flag.
- 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.

2.4.2 Communication Networks

2.4.2.1 Standard networks

Discussion:

Standard networks include those that are not automotive specific (CAN, LIN, FlexRay, CAN-XL, etc.) 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 in practice 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 physical networks will provide a channel with Ethernet-like capabilities within them (APIX□, MOST□ 150, ...). 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 of the usage of virtualization is minimized.

MTU may differ 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 host must be able to set an explicit (possibly also stable across reboots) MAC address in each VM (later, after bootup).

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.

We are unsure of the recommended system design when a MAC address change request is received from a guest VM. Should there be a way to notify other VMs (for example they need to clear ARP caches)?

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

Potential for future work exists here.

General Requirements
(this relates to the kernel API in paravirtualized setups, as opposed to the VM/HV boundary):

- Virtual network interfaces shall be exposed as the operating system's standard network interface concept, i.e. they should show up as a normal network device.

AVPS Requirements:

- If the platform implements virtual networking, it shall also use the VIRTIO-net required interface between drivers and Hypervisor.
- The hypervisor/equivalent shall provide the ability to dedicate and expose any hardware network interface to one virtual machine.
- Implementations of VIRTIO-net shall support the VIRTIO_NET_F_MTU feature flag
- Implementations of VIRTIO-net shall support the VIRTIO_NET_F_MAC feature flag
- Implementations of VIRTIO-net shall support the VIRTIO_NET_F_CTRL_MAC_ADDR feature flag
- (optional) The Hypervisor may implement a whitelist or other way to limit the ability to change MAC address from the VM.

2.4.2.2 VSock and inter-VM networking

Discussion

VSock is a “virtual” (i.e. 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 only (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 does involve writing *some* code that is custom for the virtualization case, as opposed to native, and we recommend system designers 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.

Requirements

- The hypervisor/equivalent *shall* be able to configure virtual inter-VM networking interfaces (either through VSOCK or providing other virtual network interfaces that can be bridged)
- If the platform implements vsock, it shall also use the VIRTIO-vsock required API between drivers and Hypervisor.

2.4.2.3 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 (likely good to 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

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.

2.4.2.4 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 time-stamps 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 platform, 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.

2.4.3 GPU Device

Introduction:

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

The specified *VIRTIO-GPU* is a VIRTIO based graphics adapter. It can operate in 2D mode and in 3D (virgl) mode. The device architecture is based around the concept of resources private to the host, the guest must DMA transfer into these resources. This is a design requirement in order to interface with 3D rendering.

2.4.3.1 GPU Device in 2D Mode

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 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.

REQ-1: The device ID MUST be set according to the requirement in chapter 5.7.1 in [VIRTIO-GPU].

Virtqueues.

REQ-2: The virtqueues MUST be set up according to the requirement in chapter 5.7.2 in [VIRTIO-GPU].

Feature bits.

REQ-3: The VIRTIO_GPU_F_VIRGL flag, described in chapter 5.7.3 in [VIRTIO-GPU], SHALL NOT be set.

REQ-4: The VIRTIO_GPU_F_EDID flag, described in chapter 5.7.3 in [VIRTIO-GPU], MUST be set and supported allow the guest to use display size to calculate the DPI value.

Device configuration layout.

REQ-5: The implementation MUST use the device configuration layout according to chapter 5.7.4 in [VIRTIO-GPU].

REQ-5.1: The implementation SHALL NOT touch the *reserved* structure field as it is used for the 3D mode.

Device Operation.

REQ-6: The implementation MUST support the device operation concept (the command set and the operation flow) according to chapter 5.7.6 in [VIRTIO-GPU].

REQ-6.1: The implementation MUST support scatter-gather operations to fulfil the requirement in chapter 5.7.6.1 in [VIRTIO-GPU].

REQ-6.2: 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.

REQ-7: VGA compatibility, as described in chapter 5.7.7 in [VIRTIO-GPU], is optional.

REQ-TBD: * The device must implement support for the VIRTIO_GPU_CMD_RESOURCE_CREATE_V2 as described in [VIRTIO-FUTURE]

2.4.3.2 GPU Device in 3D Mode

Discussion:

3D mode will offload rendering operations to 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.

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

2.4.4 IOMMU Device

Discussion

A Memory Management Unit (MMU) primarily performs the translation of virtual to physical memory addresses, and in doing so guarantees separation between user-space processes as well as the operating system kernel. An IOMMU provides similarly virtual address spaces to devices other than the main CPU core. Traditionally devices that can do Direct Memory Access (DMA masters) would use bus (physical) addresses, allowing them to access most of the system memory. An IOMMU limits their scope, enforcing address ranges and permissions of DMA transactions, which is required for separation. The separation of memory usage is fundamental to reliability/functional safety, as well as cyber-security considerations. When an additional layer of separation is added by introducing virtual machine scheduling then the IOMMU needs to be capable of mirroring this separation. It requires a hardware design to accommodate this extra layer of translation and/or an implementation of a *virtual* IOMMU to do the same.

The VIRTIO-IOMMU proposed device manages Direct Memory Access (DMA) from one or more physical or virtual devices assigned to a guest VM.

Potential use cases are:

- Limit guest devices' scope to access system memory during DMA (e.g. for a pass-through device).
- Enable scatter-gather accesses due to remapping (DMA buffers do not need to be physically-contiguous).
- 2-stage IOMMU support for systems that don't have relevant hardware.

Additional investigation is needed into capabilities of different hardware platforms. There are useful guidelines that include safety considerations when implementing IOMMU from vendors such as Renesas. In Xen, such implementation is currently needed to implement some Xen features, including buffer sharing, but not as a standard virtual platform API. (The feature is not provided to the guest systems as a feature but only needed internally).

VIRTIO-IOMMU’s primary use case is nested virtualization, which the AVPS group has deemed out of scope, but it also enables some pass-through device cases that are not possible in the case that there are not enough unique DMA Bus IDs. Basically, it enables supporting uses without the silicon having to implement more bus masters. Some SoCs may be lacking the required amount of separate bus masters because it increases the silicon area usage.

AVPS Requirements:

- Since the VIRTIO-IOMMU proposal isn't ratified yet, no requirements are defined at this time.

2.4.5 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 create 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.

REQ-3.5-1: The hypervisor shall provide statically configurable pass-through access to all hardware USB devices

Resource API for USB devices

REQ-3.5-2: 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.

2.5 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.

2.5.1 CAN

Discussion

The working group has found and discussed some work has been done to virtualize 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.

AVPS Requirements:

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

Potential for future work exists here.

2.5.2 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 specialized automotive networks, it seems likely that a system separates the responsibility for this communication into either a dedicated separate core in a multi-core SoC, or a single VM handling the network communication, and then forwarding necessary data to/from other VMs. Special consideration for virtualizing the LIN bus may therefore seem unnecessary, or now not worth the effort.

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.

2.5.3 FlexRay

Discussion

FlexRay™ has not been studied in this working group.

Like many specialized automotive networks, it seems likely that a system separates the responsibility for this communication into either a dedicated separate core in a multi-core SoC, or a single VM handling the network communication, and then forwarding necessary data to/from other VMs. Device virtualization for the FlexRay bus itself may therefore seem unnecessary, or now not worth the effort.

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

AVPS Requirements:

None at this time.

2.5.4 CAN-XL

Discussion

This communication standard 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.

2.5.5 MOST

Discussion

Media Oriented Systems Transport (MOST) has not been studied in this working group.

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

AVPS Requirements:

None at this time.

2.6 Special Virtual Device categories

2.6.1 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 very 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 but regardless of the hardware architecture used in the system.

There do not seem to be a generic, architecture-neutral, watchdog. I don't feel like designing one just for the whitepaper. That would also need to require to include an implementat

AVPS Requirements:

- The platform shall implement a virtual hardware interface to the hardware watchdog, following the generic watchdog described in Server Base System Architecture 6.0 [SBSA]

2.6.2 Power and System Management

Discussion:

Power management is very challenging because it has safety implications and is hardware dependent. For this reason, a standardized solution should:

- Create an abstraction layer (e.g., an API) for applications to issue power management requests in a portable way
- Create a trusted arbiter in the system that is able to take decisions
    - deny or accept guests power management requests depending on the status of the system in general and of the other guests
    - intercept power management events (power failures, user interactions, etc.) and forward them to the relevant guests safely and in the correct order

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

- Power State Coordination Interface (PSCI): Offers interfaces for suspend/resume, enabled/disabling cores, 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 interface for system management, including power domain management, performance management of system components, clock management and reset 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 PSCI and SCMI interfaces to the guest.

For other architectures we recommend following the standard approaches on those architectures.

AVPS Requirements:

- *No requirements written yet*

Potential for future work exists here.

2.6.3 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 blocks 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 hypervisor shall provide other suitable means to implement the required functionality.

AVPS Requirements:

- The hypervisor/equivalent shall support configurable pass-through access to a VM for digital general-purpose I/O hardware
- 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 for better portability).
Potential for future work exists here.

2.6.4 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. The idea would be to define how to speak SCMI using VIRTIO (virtqueues).

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.

Some details remain before a firm requirement is written.

1) Specify how to speak SCMI over VIRTIO.
2) Consider and compare the IIO (Industrial I/O subsystem) driver that is being developed for Linux kernel.

Potential for future work exists here.

Possible future requirement:

- **[PENDING]** For sensors that need to be virtualized the SCMI protocol SHALL be used to expose sensor data from a sensor subsystem to the virtual machines.

2.6.5 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 TBD.

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

2.6.6 Media codec

The proposed VIRTIO standard includes an interface to report and/or negotiate capability, stream format, rate, etc.

It is already possible to do multiple decoding. Current drivers also allow multiple decoding streams, up to the capability of the hardware. The situation will be the same here but multiple VMs share the capabilities. Prioritization of VMs can be handled by Host.

Abstract video streaming device. Input & output buffers and control channel for resolution, frame format. Encoding needs the most parameters whereas for decoding the parameters are already in the source material.

The hardware support 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:

None at this time.

2.7 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 has been added to VIRTIO (crypto accelerator for ciphers, hashes, MACs, AEAD)

RNG and entropy

VIRTIO-entropy (called VIRTIO-rng inside Linux implementation) is preferred because it's a simple and cross-platform interface to deal with.

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 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.

2.7.1 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.

AVPS Requirements:

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

- The Virtual Platform (i.e. the Hypervisor, or in some implementations a domain e.g. Dom0/DomD in Xen) MUST offer at least one good entropy source accessible from the guest.
- The entropy source SHOULD be implemented according to VIRTIO Entropy device, chapter 5.4 [VIRTIO]
- To be specific, it is required that what is received from the implementation of VIRTIO entropy device must contain only entropy and never the result of a pseudo random generator.

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

- 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

2.7.2 Trusted Execution Environments

Discussion:

Access to TrustZone and equivalent Trusted Execution Environments (TEE) should not require modification of the software. This is a feature that is frequently requested from the guest and when legacy systems are ported from native hardware to a virtual platform, there should be minimal impact on 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 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.

AVPS Requirements:

- 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).

2.7.3 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's 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 is 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 blocks, which enables at least two VMs to use the functionality without implementing the complexities of sharing.

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.

A potential future requirement (still pending) might be:

- [PENDING] If a system requires replay protection, it MUST be implemented according to the RPMB requirements specified in [VIRTIO-1.2]

AVPS Requirements:

We need some further refinement of the analysis and waiting for a release of VIRTIO 1.2 in order to lock down the platform requirement in this area.

Potential for future work exists here.

2.7.4 Crypto acceleration

Discussion

VIRTIO-crypto standard seems 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:

- If the virtual platform implements crypto acceleration, then the virtual platform **MAY** implement VIRTIO-crypto as specified in chapter 5.9 in [VIRTIO]

  (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.)

2.8 Supplemental Virtual Device categories

2.8.1 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:

- The virtual interface of the console MUST be implemented according to chapter 5.3 in [VIRTIO]
- 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.
- For security reasons, text consoles MUST be possible to shut off entirely in the configuration of a production system.
  This configuration SHALL NOT be modifiable from within any guest operating system

It is also recommended that technical and/or process related countermeasures which ensure there is no way to forget to disable these consoles, are introduced and documented during the development phase.

2.8.2 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 it is a part of a set of protocols defined by the legacy Plan9 operating system. VIRTIO is very short on details and seem 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 therefore the hypervisor 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:

- REQ FS-1: If filesystem virtualization is implemented then VIRTIO-fs MUST be one of the supported choices.

3 References

- [RFC 2119] https://www.ietf.org/rfc/rfc2119.txt
- [Linux-DeviceTree-Chosen]) How to pass explicit data from firmware to OS, e.g. kaslr-seed
  https://elixir.bootlin.com/linux/latest/source/Documentation/devicetree/bindings/chosen.txt
- [VIRTIO] Virtual I/O Device (VIRTIO) Version 1.1, Committee Specification 01, release 20 December 2018
- [VIRTIO-GPU] Virtual I/O Device (VIRTIO) Version 1.0, Committee Specification 03-VIRTIO-GPU, release 02 August 2015.
- [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
- [VIRTIO-IOMMU] VIRTIO-IOMMU DRAFT 0.8
- [9PFS] VirtFS--A virtualization aware File System pass-through. https://www.researchgate.net/publication/228569314_VirtFS--A_virtualization_aware_File_System_pass-through
- [VIRTIO-FS] https://VIRTIO-fs.gitlab.io/
- [SBSA] Server Base System Architecture 6.0 https://developer.arm.com/docs/den0029/latest
- [SCMI] System Control and Management Interface, v 2.0
- http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0056b/index.html