A new Hardware Abstraction Layer for non-Hypervisor environments based on virtio

Author: Michele Paolino, Alvise Rigo, Timos Ampelikiotis
Approver: Daniel Raho
Updated: on 2022-06-22
Contact: m.paolino@virtualopensystems.com

Table of Contents

1 About this document **3**
1.1 Document scope 3
1.2 Revision history 3
1.3 References 3
1.4 Vocabulary 3
1.5 Glossary 5
2 Introduction **6**
3 Requirements **8**
4 State of the art (SoA) - Standard vhost-user implementation **9**
5 Design **10**
6 Description of the design component by component **11**
6.1 virtio-loopback 12
6.2 virtio-loopback-adapter 14
7 Data Plane & Control Plane **15**
7.1 Control Plane 15
7.2 Data Plane 18
8 Vring Memory Management **20**
8.1 Brief description of the Vring 20
8.2 Sharing Vrings – Qemu case 21
8.3 Sharing Vrings – Virtio-loopback case 22
8.4 Architectural approaches for sharing the Vrings 23
9 Next steps **24**
10 Conclusion **25**
Annex I – Touchscreen sensitivity support **26**

1 About this document

1.1 Document scope

This document is introducing a new design of a Hardware Abstraction Layer (HAL) based on virtio architecture and targeting non virtualized environments.

1.2 Revision history

Date	Revision	Change description	Section	Author
2022-06-22	V1.1	Final Review	8, 9, 1, 2	Michele Paolino
2022-06-21	V1.0.7	Proposed technics to share the Vring	8.4, 9	Timos Ampelikiotis
2022-06-15	V1.0.5	Vrings' Memory Management	8	Timos Ampelikiotis
2022-05-11	V 1.0	First version for discussion with EG-VIRT	All	Michele Paolino
2022-05-09	V 0.8	Final review	All	Michele Paolino and Timos Ampeliokitis
2022-05-06	V 0.7	Update Description of the Control Plane	7.2	Timos Ampelikiotis
2022-05-06	V 0.6.5	Review and Requirements	3	Michele Paolino
2022-05-06	V 0.6	Transport description updated	6	Alvise Rigo
2022-05-04	V 0.5	Description of the Control Plane and Data Plane	7.1, 7.2	Timos Ampelikiotis
2022-04-28	V 0.3	Description of the Design's component	6	Timos Ampelikiotis
2022-04-28	V 0.2	Introduction	2	Michele Paolino
2022-04-15	V 0.1	First draft design	All	Timos Ampelikiotis

1.3 References

Document name	Revision
AGL_virtio_On_Native_2021_RFQ.pdf	2022-02-22
EG-VIRT Device Virtualization AGL Confluence	2022-05-11

1.4 Vocabulary

The key words **MUST**, **SHALL**, **"REQUIRED"** are equivalent and they denote an absolute requirement. If they are combined with a NOT they indicate an absolute prohibition of the specification.

The key words **SHOULD**, **RECOMMENDED** are equivalent and they denote the fact that may exist valid reason in particular circumstances to ignore a particular item, but the full implications must be understood and carefully investigated. When they are used with the word **NOT** there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful.

The key words **MAY**, **OPTIONAL** are equivalent and they mean that an item is truly optional.

1.5 Glossary

AGL	Automotive Grade Linux
DMA	Direct Memory Access
EG-VIRT	Virtualization Expert Group
HAL	Hardware Abstraction Layer
GPA	Guest Physical Address
GVA	Guest Virtual Address
HPA	Host Physical Address
HVA	Host Virtual Address
HW	Hardware
IOCTL	Input/output control device system call
IRQ	Interrupt Request
MMIO	Memory-mapped I/O
SYS_CALL	System calls
SW	Software
PA	Physical Address
PCI	Peripheral Component Interconnect
VA	Virtual Address

2 Introduction

This document details the design of a virtio HAL solution for non-virtualized environments drafted by the Automotive Grade Linux (AGL) Virtualization Expert Group (EG-VIRT).

The objective of EG-VIRT is to abstract hardware dependencies for the AGL framework/applications in a way that user space programs can be run unmodified on real hardware, virtualized systems and in the cloud. virtio is notably the main abstraction solution used in virtualized systems, and being an open standard is particularly of interest for its application also in the native case. Several virtio drivers are already available in the Linux kernel, and they provide a widely tested hardware/software interface for applications that has been selected for this design.

At the same time, an increasing number of user space devices/accelerators drivers (open source as well as proprietary) is today available. For instance, open source solutions based on vhost-user are available for input, block, random number generator, net devices. Such devices typology is a good solution to abstract the hardware access abstracting it from the kernel, from the device type and from its license.

For this reason the two endpoints considered by this design are on one side the virtio driver in the kernel, and on the other a user space device based on vhost-user (Figure 1).

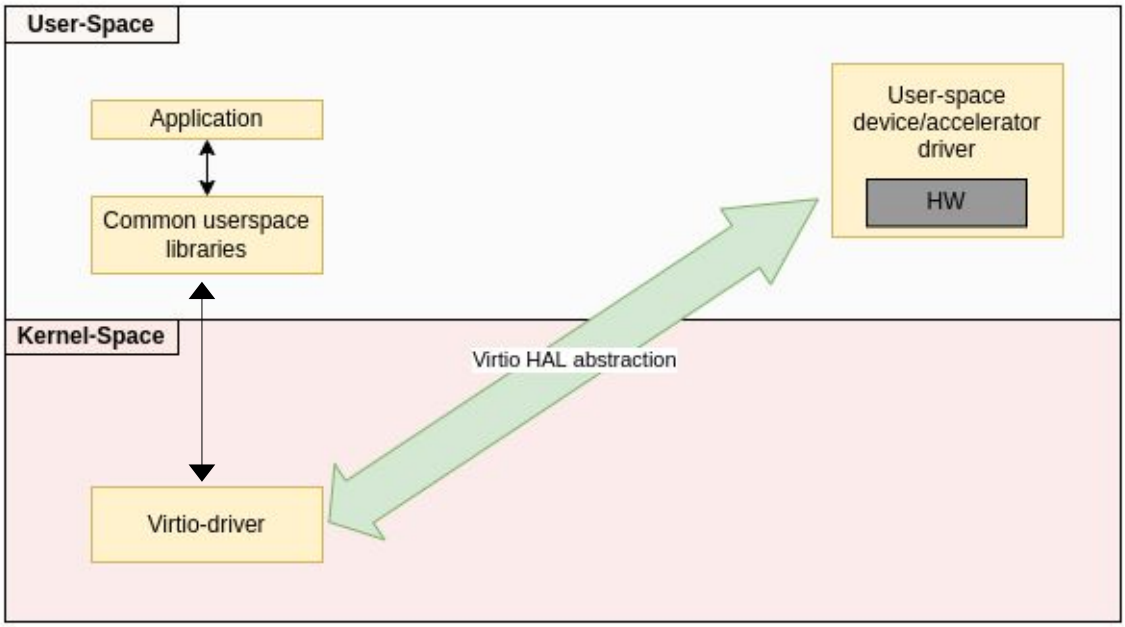

Figure 1: Objective of the virtio HAL abstraction design

The target of this activity is to develop a working virtio HAL prototypes and evaluate their performance via a set of benchmarks.

This document is organized as follows: Section 3 details the requirements that led to this design document, while Section 4 describes vhost-user technology as a key starting point for the solution proposed. Difference between the two are also highlighted. The design and its component is shown in more detail in Sections 5 and 6, where details on the implementation are given together with a step by step description of control and data planes (Section 7). Follows a closer look into the *vring* data structure and how this is handled in the presented architecture (Section 8). After, next steps and milestones are detailed in Section 9. Lastly, status and plans of the touchscreen activity are detailed together with conclusions respectively in Annex I and Section 10.

This document is intended to be continuously updated, with additional information about each of the components and with a track record of the decisions taken.

3 Requirements

Here below we recap the requirements included in the initial RFQ with an additional column that refer to the section of this document where the requirement has been mentioned.

Requirement	***Description***	***Details in Section***
REQ1	Run unmodified existing user-space applications	5
REQ2	Run existing vhost-user devices	5
REQ3	Target virtio (spec v1.1) on the kernel side	5
REQ4	AGL version LL+ and kernel version 5.4+	5
REQ5	Touchscreen device with sensitivity support and open source driver available. Possibility to connect it to the Renesas R-Car H3.	Annex I
REQ6	Follow zero-copy principle (tentative)	7
REQ7	Benchmark performance: • Measure latency and throughput overhead of virtio common device I/F implementation comparing to the case of common kernel subsystem API as a HAL without virtio frontend/backend. • For latency measurements use virtio-input ◦ Generate input event using software input device (uinput) ◦ Receive generated event via virtio-input. Measure time past. • For throughput measurements use virtio-blk.	9
REQ8	Vhost-user -input to be extended with sensitivity support	Annex I

4 State of the art (SoA) - Standard vhost-user implementation

One way to decouple the IO device from the driver in virtualized environments is today represented by vhost-user. Thanks to vhost-user, it is possible to load a standardized (virtio) driver in the guest, and then load a specific device driver in user space. Changing the device driver in user space, does not affect the guest environment. Such abstraction capability is of interest for this project, which wants to extend this functionality also to non-virtualized (native) applications.

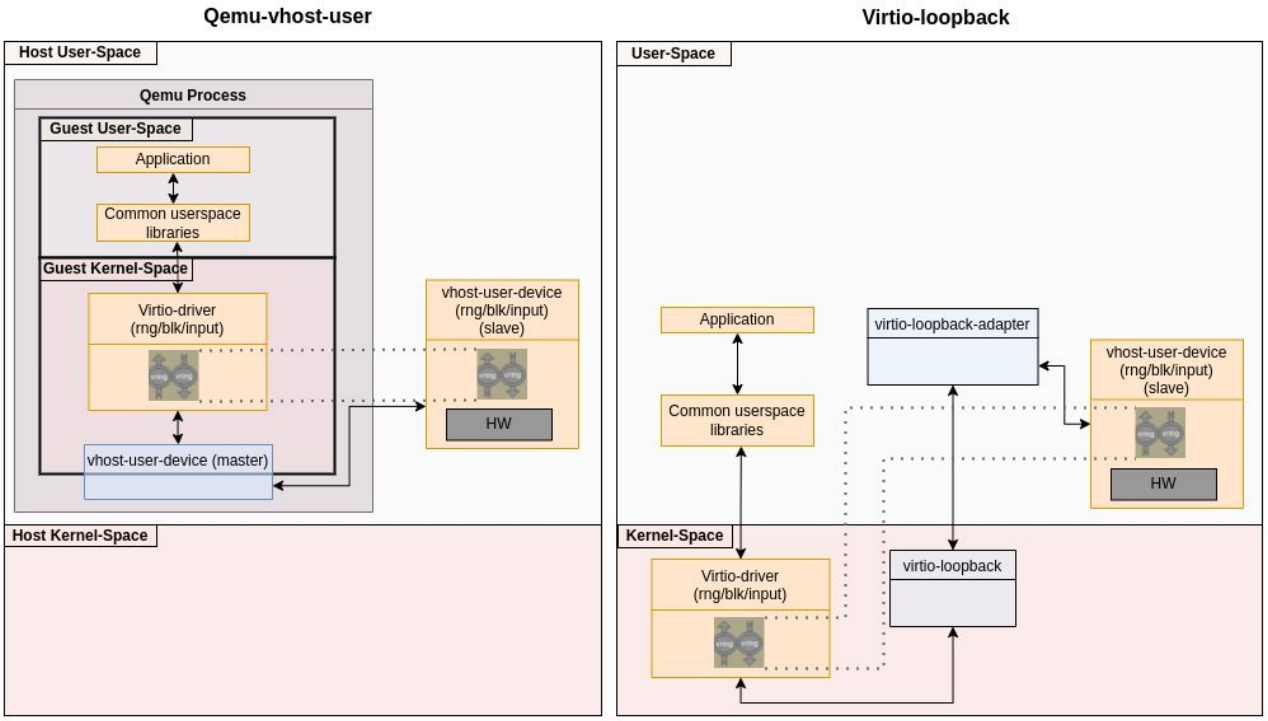

Figure 2: QEMU vhost-user architecture on the left / virtio-loopback architecture on the right side

In general, the QEMU vhost-user architecture is composed by a userspace application (vhost-user-device in Figure 2 on the left) that communicates with the QEMU process using a standardized protocol, vhost-user. Inside QEMU the communication master (vhost-user.c) is installed and interacts on one side with the guest kernel via the virtio transport and on the other side with the device driver implementation in user space.

The proposed design (rightmost side of Figure 2) keeps unchanged both the protocol used (vhost-user) and the device implementation. On the other hand, being in a non-virtualized environment, the communication master features (vhost-user.c) will be moved from QEMU to a different user space application that we call *virtio-loopback-adapter*. This component will then interact with a new transport device, that now relies in the host kernel space, instead of the host user space.

5 Design

The proposed design is derived from the requirements detailed in Section 3. One of the key points of the architecture is to keep the virtio drivers as well as the vhost-user user space devices unmodified to benefit from the existing community efforts in these areas (REQ1 and REQ2).

As a consequence, the driver to be used will be taken directly from the Linux kernel sources. For what concerns the user space devices, the table below sums up the possible targets:

Device	***Description***	***Sources***
vhost-user-input	User space implementation of input device. To be used with keyboards, mouse, etc.	https://github.com/QEMU/QEMU/blob/master/contrib/vhost-user-input/main.c
vhost-user-blk	Userspace implementation of Block (BLK) (Disk) device.	https://github.com/QEMU/QEMU/blob/master/contrib/vhost-user-blk/vhost-user-blk.c
vhost-user-rng	Userspace implementation of Random Number Generator (RNG) device. It is developed in rust programming language.	https://github.com/rust-vmm/vhost-device/tree/main/rng
vhost-user-net (tentative)	Userspace implementation of networking driver.	To be discussed with EG-VIRT

Two new components, one in kernel space and the other in user space are proposed. The kernel space component is a new virtio transport that forwards driver calls in user space where the device is implemented.
The second component is an application in user space (virtio-loopback-adapter) that is particularly important for the set-up of the system configuration, but that does not impact the data plane path to avoid overhead.

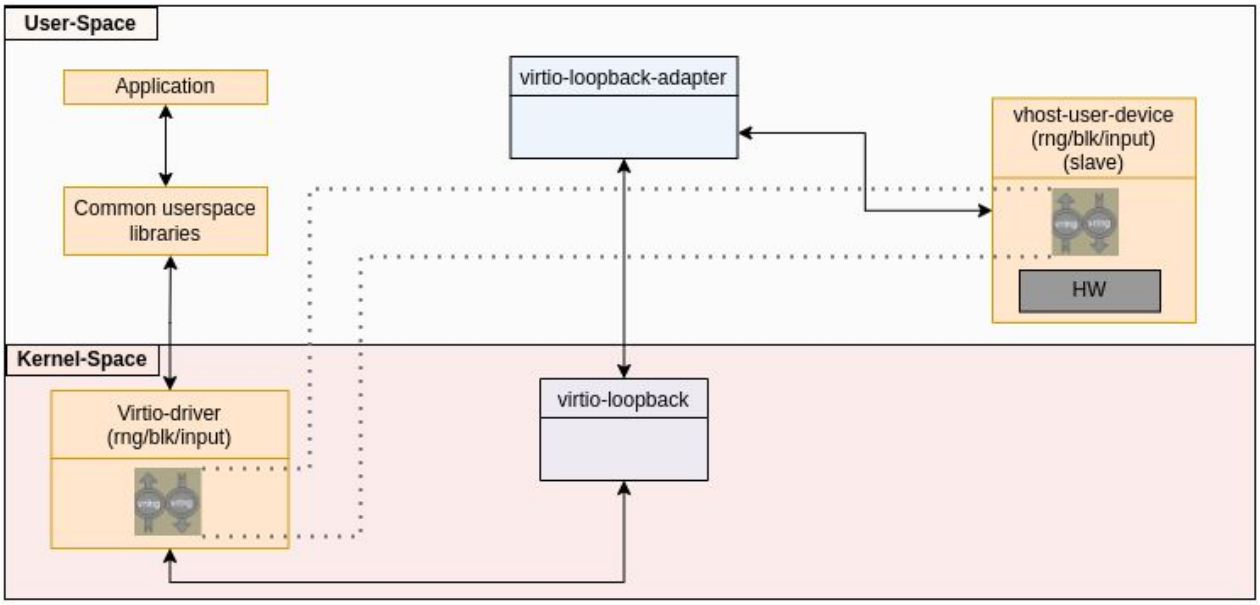

Figure 3: virtio-loopback high level design overview

The proposed architecture is not particularly bound to a given version of the Linux kernel, AGL or virtio specs. As a result, during the implementation phase, REQ3 (virtio 1.1) and REQ4 (AGL LL+ and kernel 5.4+) will be addressed.

The developed results will target meta-agl-devel/meta-egvirt and will be enabled agl-demo-platform-virtio-native image in the AGL community. The kernel space virtio transport will be proposed to the Linux kernel community, with the virtio-loopback-adapter application can be considered part of a different repository hosted by AGL. The main components of the proposed architecture are detailed in the next section.

6 Description of the design component by component

In this section each component of the design is detailed, including information about the implementation. The idea behind this design is to keep the same logic as in QEMU, to not diverge too much from its highly tested design and logic as well as to keep existing vhost-user devices unchanged.

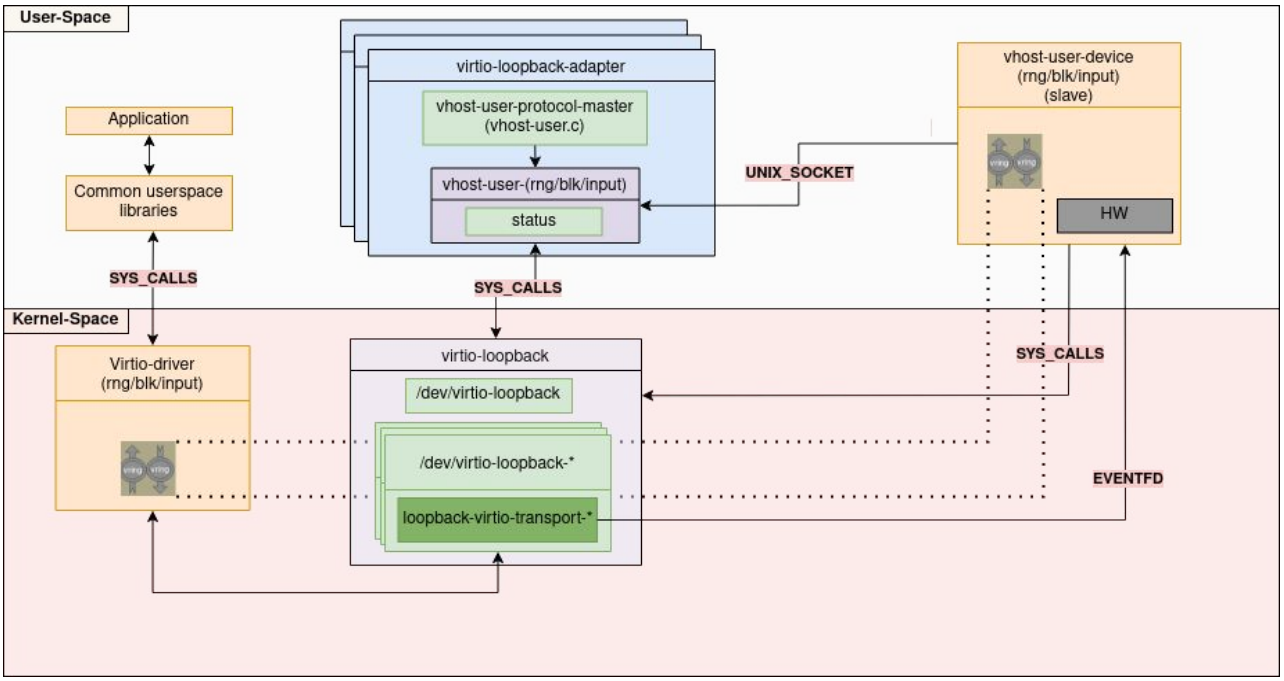

Figure 4: virtio-loopback kernel and user space components

6.1 virtio-loopback

Virtio-loopback resides in the Linux kernel and is acting as virtio transport that communicates with the virtio driver, as well as a standard character device (/dev/virtio-loopback) that provides memory to be mapped to the vhost-user-device.

Compared with other existing virtio transport drivers (such as virtio-mmio or virtio-pci), virtio-loopback transport driver is designed to be used in an environment with no emulation, virtualization or para-virtualization. Its first objective is allowing the virtio driver to correctly setup the virtio communication with the rest of the system.

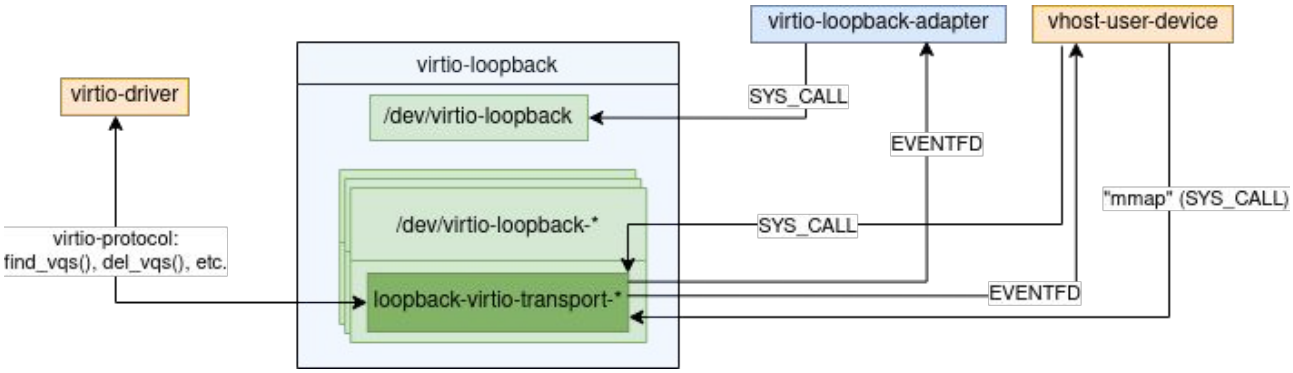

Figure 5: virtio-loopback kernel components overview

More in details, existing virtio-transport drivers interact with the QEMU emulated device (called *virtio-(input/rng/blk)-device in QEMU*) via Memory Mapped IO (MMIO) “read/write” operations. The proposed *virtio-loopback-transport* instead of MMIO read/write will trigger notifications (eventfd) to the “virtio-loopback-adapter”, either to notify for new written data (write operation) or to request new data (read operation).

Additionally, the key role of the transport layer is to implement hooks for the virtio driver, that in the Linux kernel driver are represented by the *virtio_config_ops* structures. As a result, each one of the expected function needs to be re-implemented to be run in a non-virtualized environment:

- **Functions that interact with the device status** (e.g., get(), set(), get_status(), set_status(), get_features(), finalize_features() and reset()):
  - **Implementation in a virtualized environment:** these call are making MMIO read and write accesses to the virtio device registers. As a result of these accesses, the proper QEMU IO callbacks (part of the emulation of the virtio device) are returning/setting the desired values. For instance, to get the device status with the virtio-mmio transport, the virtio driver would MMIO read 4 bytes at offset VIRTIO_MMIO_STATUS, causing a trap in the host system and making QEMU resolve the request.
  - **virtio-loopback implementation**: This will be implemented as a combination of eventfds and system calls. The above device status read example in this scenario would be accomplished by writing the VIRTIO_LOOPBACK_STATUS value in a shared structure between the device and the transport; then, the transport driver would forward the request to the user space adapter notifying it via an eventfd and wait for a response, which will be written back in a different field of the shared structure. The notification of such a response will be done with an ioctl() or write() to the virtio-loopback kernel driver.

- **Functions that set-up virtqueues** (e.g., find_vqs(), del_vqs()):
  - **Implementation in a virtualized environment**: find_vqs() and del_vqs() are necessary to setup the virtqueues for the device. Although this job seems to be transport layer-independent, it is actually not as the virtqueues require to have a dedicated IRQ which depends on the system/type of transport layer used. In addition, the configuration of the virtqueue requires several pieces of information from the device (like maximum number of queues supported, alignment requirements, etc.) which are retrieved via specific register (PCI or MMIO) accesses. At the end of this calls, the driver must send to the host (or, in our case, the vitio-loopback-adapter) the coordinates of the queues (in virtio MMIO driver this is done via the registers VIRTIO_MMIO_QUEUE_DESC_LOW, VIRTIO_MMIO_QUEUE_AVAIL_LOW and VIRTIO_MMIO_QUEUE_USED_LOW). It is worth noting that the IRQ allows to have notifications from the slave (the actual vhost-user driver) to the master (the virtio kernel driver). In QEMU, this notifications are done via a guest notifier, as opposed to the notification path that goes from the guest to the host, which is called host notifier.
  - **virtio-loopback implementation**: In the loopback case, we can not have a dedicated IRQ as we are not in a virtualized environment. For this reason, the virtio-loopback kernel driver must provide some means of notification for the user space, in order to signal certain events (analogously to what done in the virtualized scenario by the host notifier which relies on IRQFD to trigger an IRQ inside the guest). This can be done by a dedicated file, */dev/virtio-loopback-**, which supports a SYS_CALL as notification and which will be opened in user space by the *virtio-loopback-adapter*. Thanks to a prior exchange of file descriptors between the *vhost-user-(input/blk/rng)-device* and the *virtio-loopback-adapter*, the former will be able to directly invoke the SYS_CALL on the file descriptor (it could be a simple write(); in this case, the notification mechanism will be identical to the virtualized scenario with just a different file-based abstraction i.e., character file vs. eventfd file).
- **Functions to access virtqueues** (e.g., get_shm_region())
  - **Implementation in a virtualized environment**: In the virtualized scenario this is done indirectly, by exploiting QEMU (owner of the whole guest memory and thus of the virtqueues). In fact, QEMU can easily share the file descriptor of the guest memory and send it to the user space process willing to read and write to it (to do this, however, the process must be privileged). Simplifying a bit, the destination process can *mmap()* the whole guest memory and, thanks to the fact that it has the address of each virtqueue, it can precisely reach them.
  - **virtio-loopback implementation**: In the loopback case, to enable a user space process (*vhost-user-(input/blk/rng)-device*) to directly access the *virtqueues,* the *virtio-loopback-transport* driver should create "/dev/virtio-loopback" folder in which specialized character files should be created that refer, each of them, to the underlying virtio kernel driver. For instance, /dev/virtio-loopback-0 can be created for the first input device, and so on. By mmap()-ing this file, the user space process will have access with a contiguous virtual memory area to all the queues owned by the corresponding virtio kernel driver.

6.2 virtio-loopback-adapter

The *virtio-loopback-adapter* application has the important role to establish the control plane between the userspace device implementation and the virtio driver in the kernel space. As mentioned in Section 4, this component implements part of the features that are today implemented by QEMU (vhost-user-protocol-master in Figure 6) and keeps track of the status of each user space device present on the system. One instance of virtio-loopback adapter is run per device. This mimics the behaviour that today QEMU has, and provides best compatibility with existing devices implementation.

As detailed below in the figure, on the one hand *virtio-loopback-adapter* communicates with *virtio-loopback-transport* via the /dev/virtio-loopback device node using SYS_CALLS and on the other hand it communicates with the *vhost-user-device* over UNIX socket using the *vhost-user protocol.*

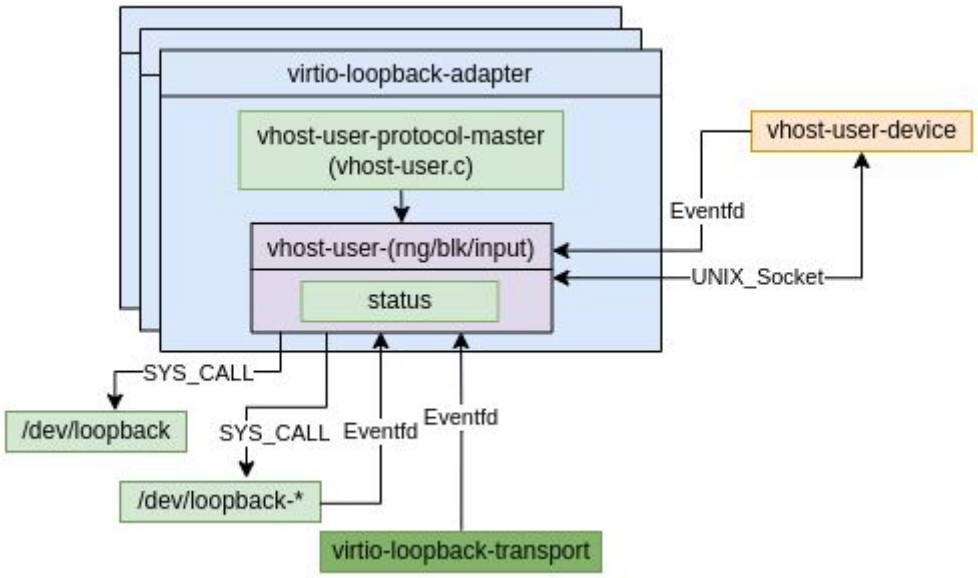

Figure 6: virtio-loopback-adapter component description

Every time that a new device is started, it waits for vhost-user messages to be received over the shared socket. As a result, the *virtio-loopback-adapter* will implement the necessary logic to establish the communication with the *vhost-user(input/rng/blk)-device* and to retrieve information from the *vhost-user(input/rng/blk)-device* needed from both virtio and vhost-user communication.

The device status information is queried from both the driver and the device implementation itself. In vhost-user virtualized environments, QEMU keeps track of such information and provides means to get-set it to the virtio driver, as well as to the userspace device via vhost-user messages. In this design, to maintain the same implementation of both the driver and the user space device, the device status will be kept and updated in the adapter component.

As a result, what in the virtualized scenario was done by means of MMIO read and write, in the *virtio-loopback-adapter* will be done via:

- eventfd(s) used to notify the adapter about new requests
- IOCTL (SYS_CALL) calls to notify the kernel components of a new request

7 Data Plane & Control Plane

This section provides further details of how the data and control planes of the solution will work. Each execution flow is shown detailing the different execution steps. Some of these execution steps might slightly change during the implementation phase (e.g., because a performance optimization has been found and applied, because of comments from open source communities, etc.).

7.1 Control Plane

The control plane execution is shown in the following Figures in four steps. It is normal for the control plane to be more complex than data plane. The target of this phase is to set-up the environment to facilitate as much as possible (and with best performance) the data plane.

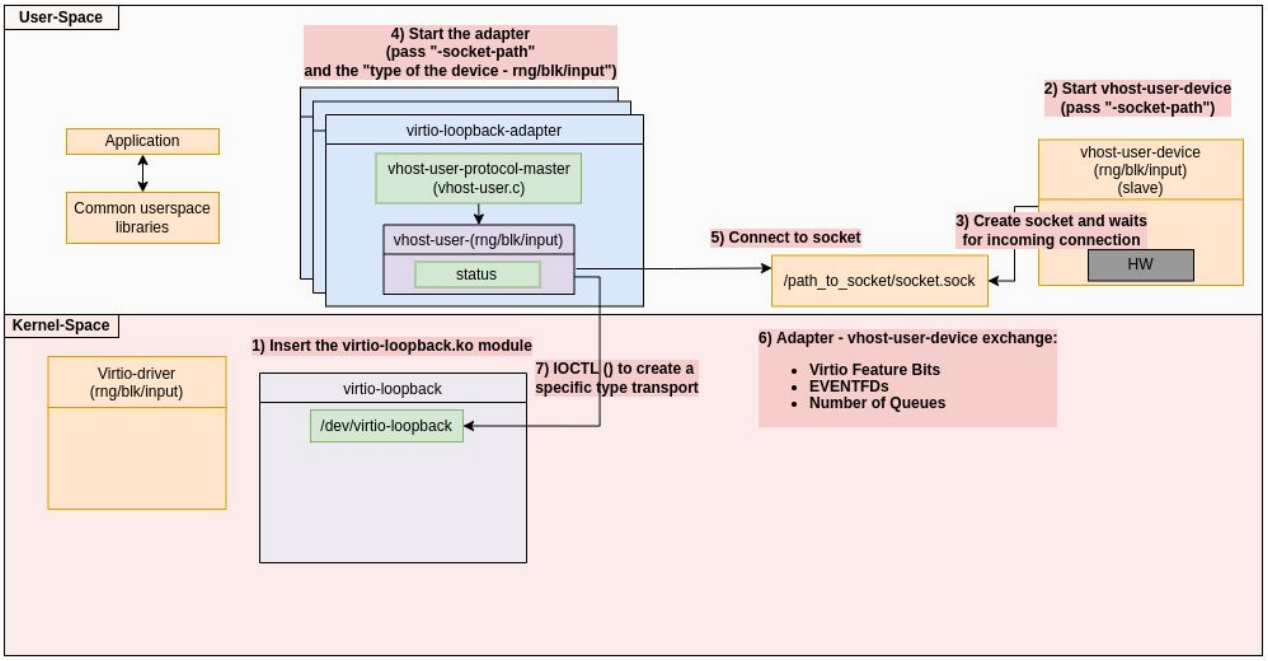

Figure 7: **Components initialization**: The control plane set up starts with the execution of the components presented in this design document. Initially, the *virtio-loopback.ko* is added to the system (e.g., running *insmod virtio-loopback.ko*). The module inserted will create a new device node */dev/virtio-loopback* and wait for requests from the virtio-loopback-adapter (step 1). Later, the device is started, providing as an argument the path to the UNIX socket and the other device dependent parameters (e.g., shared memory object, block file, etc.) used by the device. After, virtio-loopback-adapter is started with socket and device type parameters (steps 2 to 5). Initial message exchanges will happen already between device and virtio-loopback-adapter exchanging information related to virtio and vhost-user features. At the end of this exchange, the virtio-loopback-adapter knows (almost) all the information that later will be requested by the virtio-driver (step 6). Once all components are up and running, the virtio-loopback-adapter calls an IOCTL SYS_CALL to the */dev/virtio-loopback* device providing the virtio type (e.g., blk, rng, input) (Step 7).

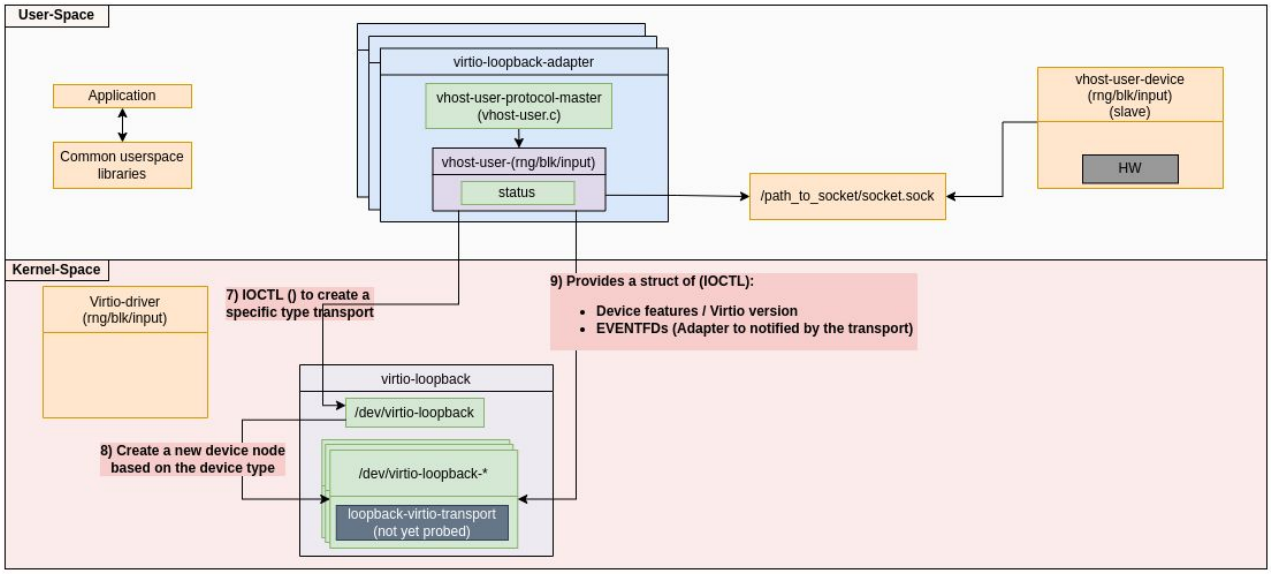

Figure 8: **virtio-loopback-adapter configuration**. As a result of the previously described IOCTL SYS_CALL, a new device node */dev/virtio-loopback-(rng/blk/input)* will be created (step 8). After, the virtio-loopback-adapter calls an IOCTL to the */dev/virtio-loopback-** and provides virtio specific information (Step 9) that has earlier obtained from the *vhost-user-(rng/blk/input)-device* (e.g., feature bits, max number of VirtQueues supported, etc.).

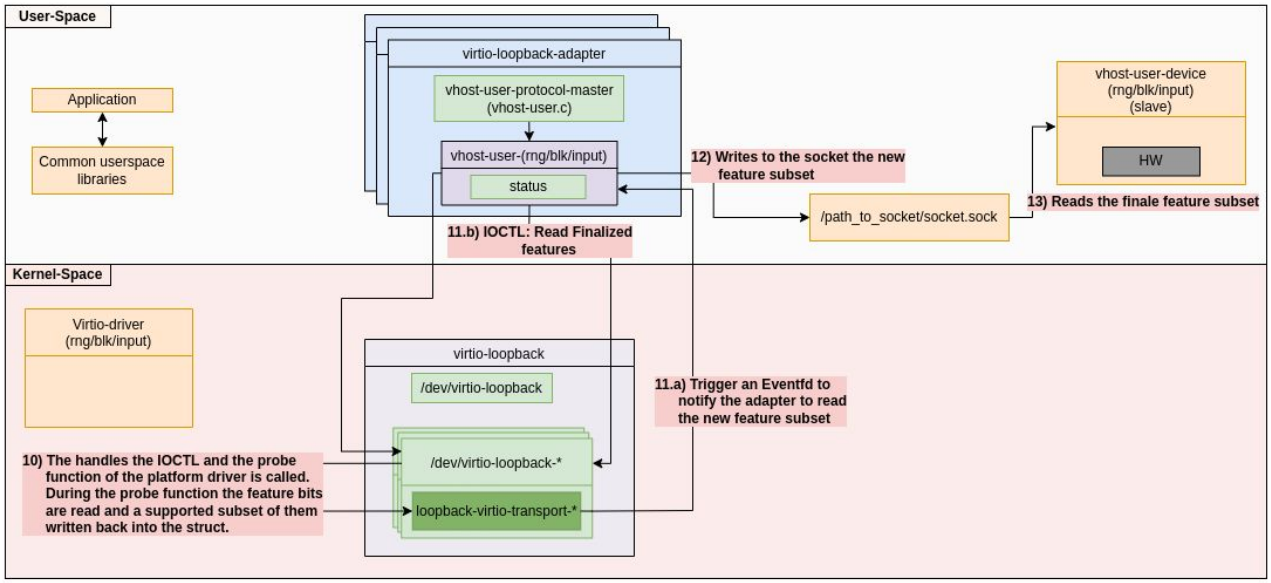

Figure 9: **virtio-loopback-transport initialization**: At this point, */dev/virtio-loopback-** probes the *virtio-loopback-transport* component with the information provided by the previous IOCTL (e.g., VendorID, Device Type, Magic Number, etc) and process the finalized features to be written back to the *vhost-user-(rng/blk/input)-device* (Step 10). This happens with an EventFD in direction of the virtio-loopback-adapter that then reads the features (via a SYS_CALL) (Step 11a and 11b). The *virtio-loopback-adapter* sends the finalized feature bits via the *Unix Socket* to the *vhost-user-(rng/blk/input)-device* (VHOST_USER_SET_FEATURES) (Step 12-13).

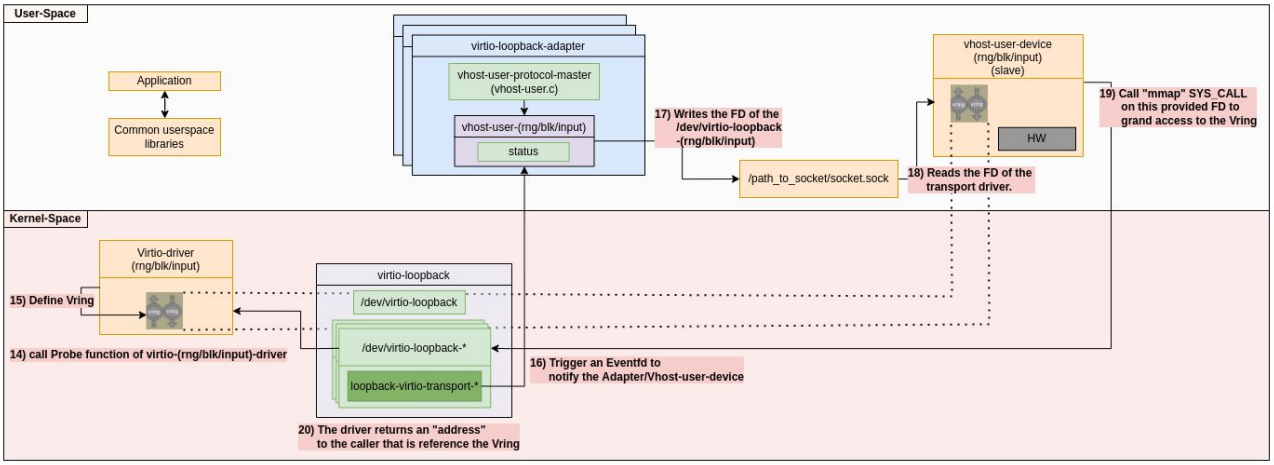

Figure 10: **virtio-(rng/blk/input)-driver initialization**: At this point, *virtio-loopback-transport* registers a new device *virtio-(rng/blk/input)* (Step 14). The virtio-(rng/blk/input) is started and the *Vrings* are created (Step 15). The *virtio-loopback-transport* notifies the *virtio-loopback-adapter* that the *Vrings* are created and can be shared (with the *vhost-user-(rng/blk/input)-device*) via an EventFD (Step 16). The *virtio-loopback-adapter* transfers this information to *vhost-user-(rng/blk/input)-device* over the Unix Socket with the following messages “VHOST_USER_ADD_MEM_REG”, “VHOST_USER_SET_VRING_NUM”, “VHOST_USER_SET_VRING_BASE”, “VHOST_USER_SET_VRING_ADDR”, “VHOST_USER_SET_VRING_KICK”, “VHOST_USER_SET_VRING_CALL” (Step 17-18). The *vhost-user-(rng/blk/input)-device* calls “mmap” to the File Descriptor (FD) referred to the */dev/loopback-** (Step 19). The “mmap” SYS_CALL is defined in the *virtio-loopback-transport* and returns a virtual address to the *Vrings* (Step 20).

7.2 Data Plane

This section describes the process followed in order to exchange data between the virtio-driver and *vhost-user-(rng/input/blk)-device*. In general, the mechanism used to share the virtual rings (*vrings*) between the *virtio-driver* and the *vhost-user-(rng/input/blk)-device* is based on a memory mapping shared between the *vhost-user-device* and the the device. This solution enables vrings direct access from the device, without the need to copy buffers from/to kernel (copy_to_user(), copy_from_user()) to exchange data. As a result, the proposed design meets REQ6 (no copies in data exchanges, to minimize overhead).

Here below we first describe the steps needed to send data from the driver to the device (Figure 11) and later we detail the opposite case, when the user space device sends data to the driver (Figure 12).

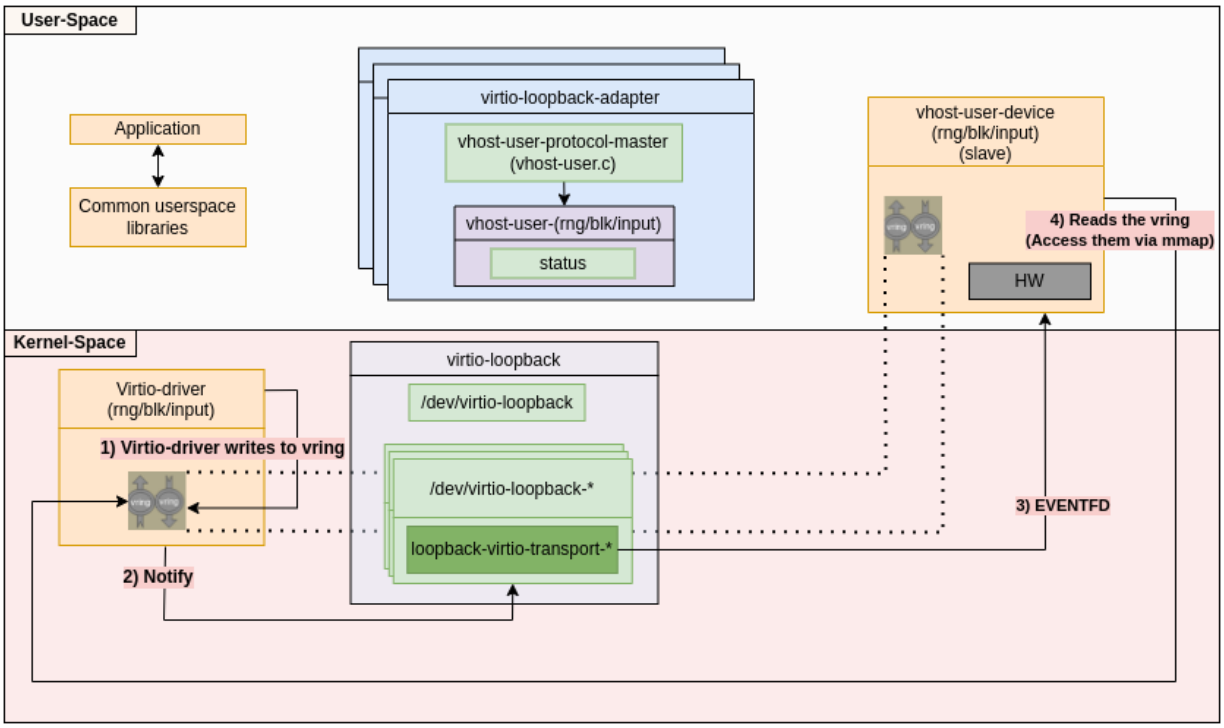

Figure 11: **virtio-driver sends data to *vhost-user-device***: After having written new data in the vrings, the *virtio-driver* notifies the *vhost-user-device* to read it via the *virtio-loopback-transport* driver by executing the notify function which is defined by the *virtio-loopback-transport* driver (Step 1-2). Such function triggers an eventfd to the *vhost-user-device*, that will know at that point that new data has been written in the *vrings* shared memory (Step 3-4).

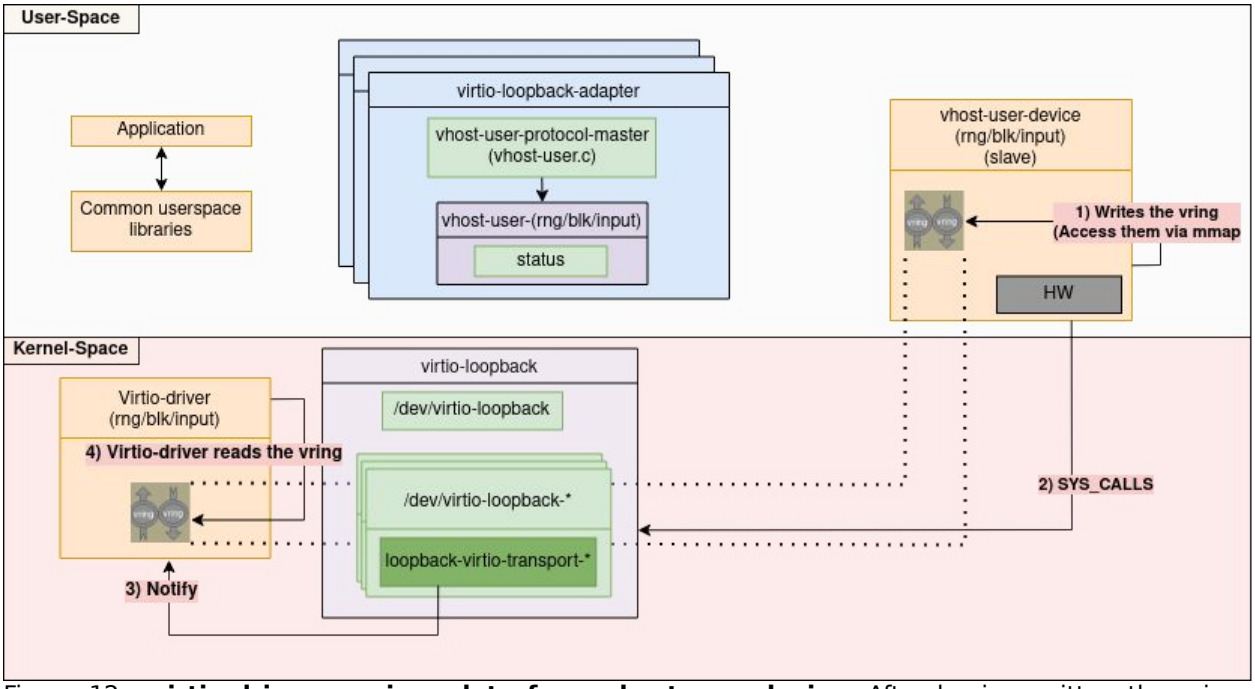

Figure 12: **virtio-driver receives data from vhost-user-device**: After having written the vrings memory, the *vhost-user-device* notifies the loopback component via an ioctl/write call(Step 1-2). At this point, as a result of the SYS_CALL, the *virtio-loopback-transport* notifies (by calling the function "vm_interrupt") the *virtio-driver* to read the vrings (Step 3). Finally, the *virtio-driver* reads the new data from the *vrings* (Step 4). This is an optimized execution flow that will be evaluated from the viewpoint of the impact it has on the existing device implementation. The alternative is to pass by the virtio-loopback adapter.

8 Vring Memory Management

The *vrings* are a very important component of the virtio protocol for transferring the data. The sharing of the *vrings* between the *virtio-loopback-transport* and the *vhost-user-device* is the key feature of this architecture.

This section includes:

1. A brief description of the *Vrings'* data structure
2. Accessing the *V*rings from Qemu and *vhost-user-device*.
3. Sharing the *Vrings* between the *virtio-loopback*-transport and the *vhost-user-device*.
4. Architectural approaches which can overcome the challenges.

8.1 Brief description of the Vring

The vrings are used to exchange data between the guest and the host. As visible in the Figure 13, the *vring structure* consists of the following three components:

1. *struct vring_desc*: Holds the Guest Physical Addresses of the shared buffers which include the data to be exchanged between the driver and the device.
2. *struct vring_avail*: Contains the indexes of the vring_desc structure which points to the data supplied by the driver to the device.
3. *struct vring_used*: Holds the indexes of the vring_desc structure which points to the data supplied by the device to the driver.

Among these three data structures, "*struct vring_desc*" is the most important because it contains the address of the data to be shared. In fact, the "*vring_desc_t* struct" includes the attribute "*addr*" which holds the Guest Physical Address (GPA) where the data is stored. The data is organized in a Scatter Gather List (GS) which is located outside this contiguous memory region. This SG entry are filled by the driver/device to transfer data (input/output) between the Guest and Host.

As the name suggests, the GPA requires the existence of a guest memory, which is not present in the virtio-loopback case. This point will be described further below.

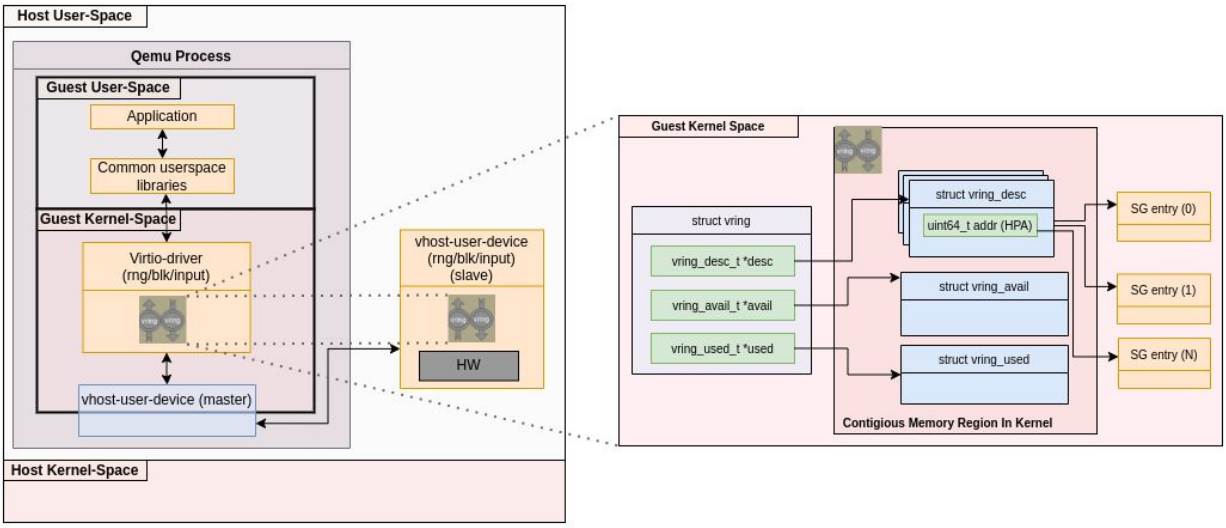

Figure 13: Description of the Vrings

8.2 Sharing Vrings – Qemu case

In this section we will focus how Qemu access the vrings and later how these are shared and accessed by *vhost-user-device*.

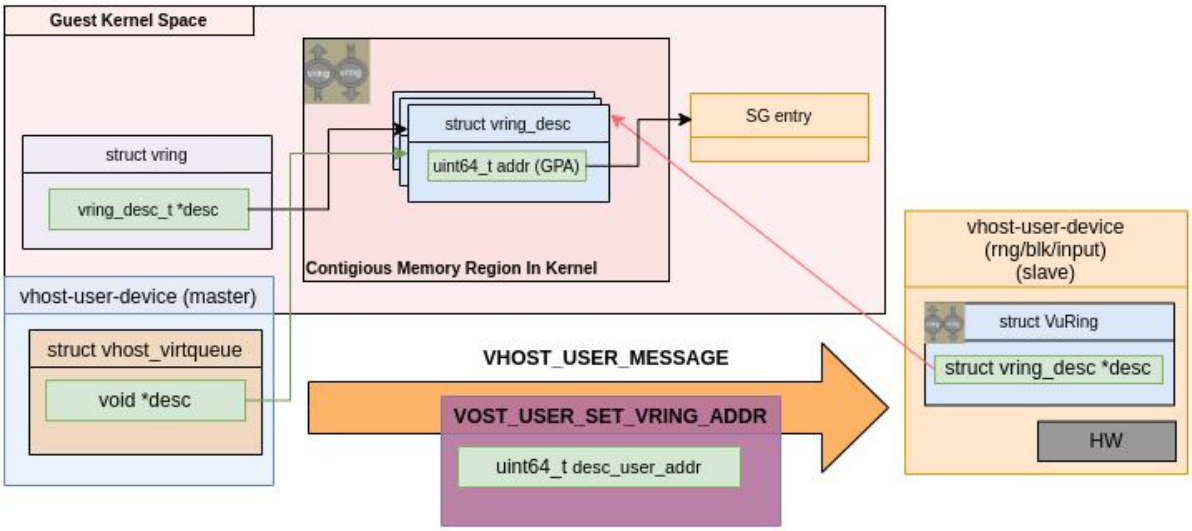

Figure 14: Qemu share the Vrings with vhost-user-device

The *vhost-user-device in order* to access the *Vrings* needs to: 1) create a pointer to the "vring_desc" data structure (detailed in section 8.1), and 2) decode the *addr* attribute, contained into the "vring_desc", in order to obtain the memory address of the shared data.

As for the first one, Qemu knows where the Vrings' data structure is located in the Guest Memory's (Guest Physical address), and via DMA operations is able to convert these address into HOST pointers to that data (Host Virtual Address, HVA). Later on Qemu is sharing these HVA pointers to the vhost-user-device via a vhost-user message. However, these pointers are valid in the Qemu address memory space only, and the vhost-user-device needs a mechanism to translate them into its own virtual address space. This is done via the "qva_to_va()" function from the vhost-user library.

The second point, attribute "addr", holds the GPA of the shared buffer and it is accessible by Qemu via DMA operations. Vhost-user-device at this point needs a similar mechanism able to translate a GPA to its own VA, this is the role of the "vu_gpa_to_va()" function.

8.3 Sharing Vrings – Virtio-loopback case

In this section we will describe the difference of accessing the Vrings in the *virtio*-loopback case compared to the Qemu shown in the previous section.

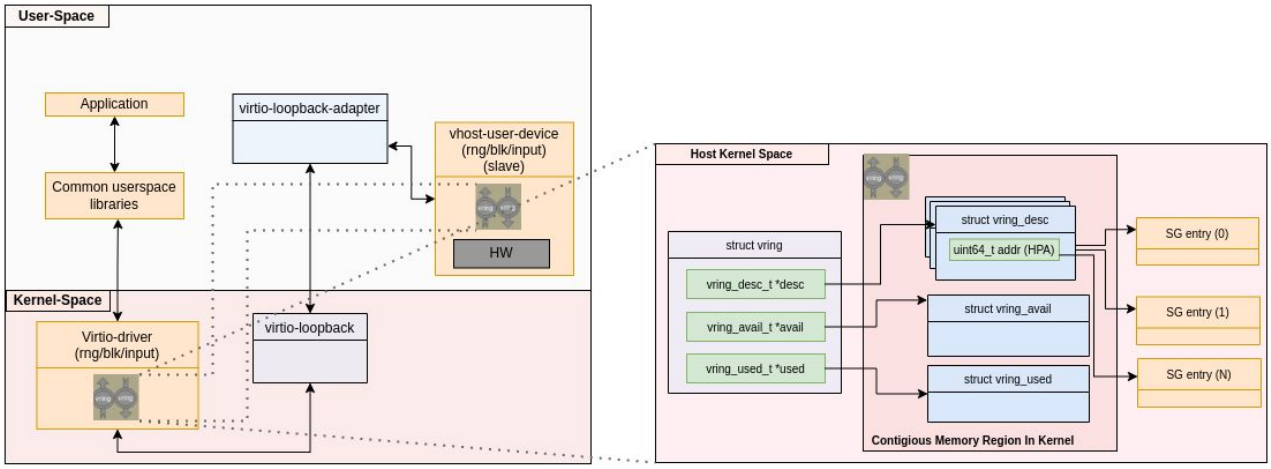

Figure 15: Sharing Vrings into the Virtio-Loopback architecture

As it is depicted in the Figure 15, the key difference is that, not having a guest in the *virtio-loopback* case, the vrings are not anymore in the guest kernel memory (host user space), but they are located in the host kernel space. Consequntly, the “uint64_t addr” attribute is a “Host Physical Address” and not anymore a “Guest Physical Address” (like in the Qemu case).

As a result, there is a need to make sure that *vhost-user-device* can access the “addr” attribute of the *vring* structure and translate/remap it in its own address space.

8.4 Architectural approaches for sharing the Vrings

One approach to enable *vhost-user-device* to access the data pointed by a HPA, is to make the *vhost-user-device* to call “mmap” in a way to define a base address *virt_base (Figure 16)*.

The *virtio-loopback-transport* driver will bind the return virtual pointer of this call to the lowest HPA (*l_hpa*) used for that *virtio-device.* Whenever, *vhost-user-device* wants to access a HPA (*l_hpa + offset_x)* will access the virtual address *virt_base + offset_x*. Then, a page_fault is going to take place and the *virtio-loopback-transport driver* will provide access to the correct data by applying the following formula to the *fault_addr*: *l_hpa + (fault_addr – virt_base) = l_hpa + offset_x.*

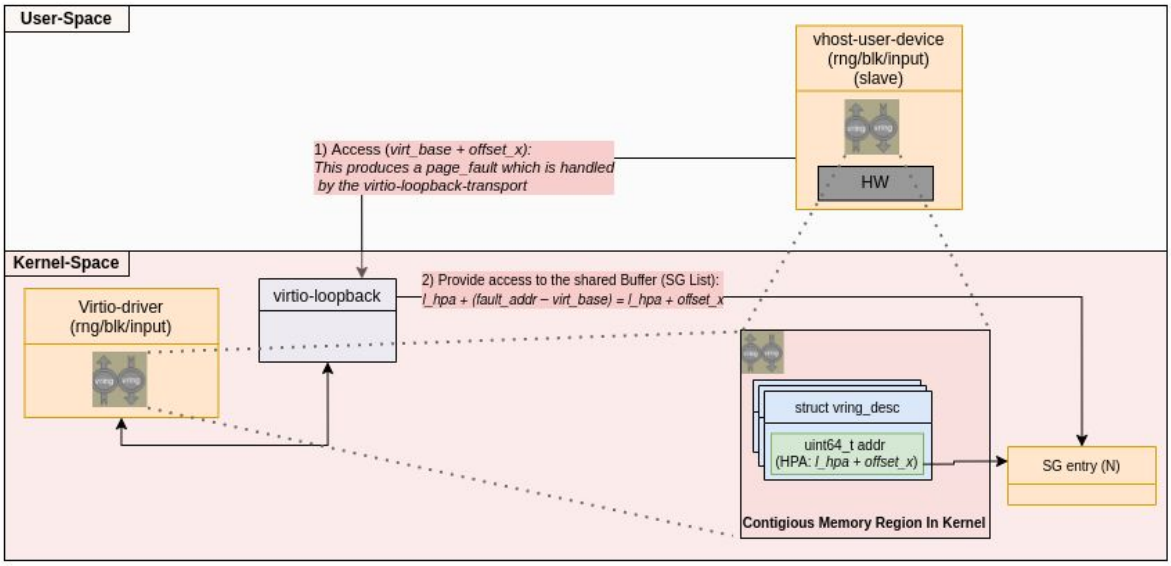

Figure 16: Sharing Vrings - Virtio-Loopback - Page Fault approach

Alternatively, the *vhost-user-device* calls “mmap” for each HPA that needs to access, and the *virtio-loopback-transport* driver provides the corresponding virtual pointer.

It is worth to mention that the above described approaches demand changes of “vhost-user” library, which is used by the the *vhost-user-device*. The reason of that is, “vhost-user” library is designed having in mind the presence of Qemu and a Guest (VM), which neither of them are part of our scenario. As a result, modification need to be applied at segments of the source code related to sharing and translating “pointer” and PA from Qemu (or Guest) to *vhost-user-device.*

9 Next steps

This document is intended as a first design description internal to AGL. It aims at receiving comments from EG-VIRT and from the AGL community.
After, once AGL comments are received and addressed, an RFQ for the Linux kernel will be prepared. Before doing this, a first version of the code will be prepared, in a way that we can attach a code example to the RFC. This milestone is important to capture early comments from kernel developers as well as to disseminate the AGL intentions to the kernel community.

The table below summarize the next milestones of the project.

Milestone	***Description***
2022-05-11 (Release of the present document)	- Touchscreen device selected and tested with Linux - virtio-loopback design virtio-loopback and touchscreen components (AGL internal RFC) presented at AGL Berlin Workshop
2022-07	Presentation of an RFC to Linux kernel developers community
2022-09 (AGL Event)	- AGL AMM presentation with demonstration of initial virtio-loopback implementation - Touchscreen device integrated in native AGL with plans for integration in virtio-loopback
2022-11	Final delivery: - virtio-loopback transport and adapter components - Touchscreen device integration in virtio-loopback - Benchmarks (REQ7)
2022-12 (AGL ALS)	Demonstration and benchmark reports

10 Conclusion

This document presented the virtio Hardware Abstraction Layer (HAL) solution for non virtualized environments design, together with its most important components and a plan for the implementation.
The design targets to address all the requirements (Section 3) listed by AGL EG-VIRT, and aims at evaluating the performance overhead that the proposed HAL creates.

This document is intended to be continuously updated, addressing comments arriving from AGL community and updating the design with additional information retrieved directly during implementation.

Annex I – Touchscreen sensitivity support

This project includes activities related to the identification of a touchscreen device with sensitivity support (REQ5) and with its integration in both the AGL standard set-up and the virtio-loopback case (REQ8).

After having evaluated different solutions, the MatrixOrbital HTT70A R1.0.0, a 7 inch 1024 x 600 HDMI Display with Touchscreen was accepted by EG-IVI. This solution provides an utility application (HTT_utility) to change the sensitivity of the device. The source code is available at https://github.com/MatrixOrbital/HTT-Utility.

```
155  int set_sensitivity(hid_device *handle, int sensitivity)
156  {
157          unsigned char buf[256];
158          buf[0] = REPORT_MXT_SENSITIVITY;
159          buf[1] = sensitivity;
160          int res = hid_send_feature_report(handle, buf, 2);
161          if (res < 0) {
162                  return 0;
163          }
164          return 1;
165  }
166
```

Figure 17: Implementation of the set_sensitivity function in the htt_util.cpp file

As shown in Figure 17, the set_sensitivity calls the hid_send_feature_report that, for the Linux implementation of the *hidapi* results in an *ioctl() in the HIDRAW device*. This feature has been tested on Linux development system (x86 desktop) and is in fact working.

For what concerns the touchscreen input, the device has been tested on the development system and is recongnized by the Linux kernel with no additional driver needed. The same was also tested in a KVM environment, where using vhost-user-input the events were visible in kernel space. The touchscreen device has been recognized at boot time with the following:

> *input: Matrix Orbital Multi-Touch Device as /devices/pci0000:00/0000:00:03.0/virtio0/input/input1*

For what concerns the Renesas R-Car M3 target, first investigations in the kernel configuration shown that the CONFIG_HIDRAW is not enabled by default in the current AGL configuration. This driver is needed to expose the /dev/hidraw devices used to set sensitivity (via ioctl). As a result, with AGL today the touchscreen device works but it is not possible to change sensitivity.

Next steps are to add CONFIG_HIDRAW to the M3 Linux BSP configuration, and to test it with AGL LL+ and kernel 5.4+. This, together with a plan for integration in virtio-loopback to be discussed with EG-IVI, will be released at AGL AMM 2022.

Lastly, the release license of the HTT-Utility application is not clear (nothing is mentioned in github, but the company customer support says it should be MIT or 3 clause BSD). A simple alternative user space application could be implemented if there are licensing issues in integrating htt_util in AGL.