



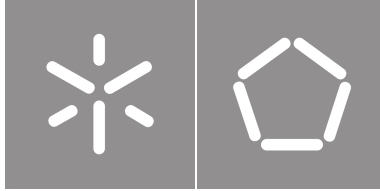
Universidade do Minho

Escola de Engenharia

António Nuno Alves Cardoso Capela Ribeiro

VirtIO Infrastructure for a Static Partition Hypervisor: virtio-blk and virtio-console

Março, 2023



Universidade do Minho

Escola de Engenharia

António Nuno Alves Cardoso Capela Ribeiro

VirtIO Infrastructure for a Static Partition Hypervisor: virtio-blk and virtio-console

Master Thesis

Master in Industrial Electronics and Computers

Embedded Systems

Work developed under the supervision of:

Dr. Sandro Emanuel Salgado Pinto

Março, 2023

COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

License granted to the users of this work



**Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International
CC BY-NC-SA 4.0**

<https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en>

Acknowledgements

I would like to thanks to my supervisor, Dr. Sandro Pinto, for all the guidance, the availability and for the knowledge that have made this process so interesting.

To everyone in the Embedded System Research Group laboratory for the friendship and funny moments. Specially, to PhD student José Martins, for the help during all this period, and to João Rodrigo for helping me in everything I needed.

To João Alexandre de Carvalho, for being a stupendous friend and being always by my side during these lasts years.

To my parents, for always pushing me forward, for all the support during my entire life and for the advises that I needed. In special, I want to thank my father for the guidance during all my student path.

To my lovely girlfriend, Iryna Kaplunska.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

_____, _____
(Place) (Date)

(António Nuno Alves Cardoso Capela Ribeiro)

"Live as if you were to die tomorrow. Learn as if you were to live forever." (Mahatma Gandhi)

Abstract

VirtIO Infrastructure for a Static Partition Hypervisor: virtio-blk and virtio-console

The automotive industry has increasingly focused on the integration of systems that perform very specific functions. At the same time, there is an increasing need to reduce Size, Weight, Power and Cost (SWaP-C) and, consequently, the need to group subsystems, usually with different levels of criticality, onto the same hardware platform. Currently, these systems are referred to as Mixed-Criticality System (MCS) and require strong isolation between systems of different criticality to respect the freedom from interference and real-time deadlines.

Virtualization emerged as a natural enabler for MCS, since, in addition to providing a way to integrate different subsystems onto the same hardware platform, it enforces a strong isolation among consolidated workloads. Following this trend, hypervisors, developed for server and cloud applications, were retrofitted to embedded system architectures such as Advanced RISC Machine (ARM). These hypervisors do not fulfil the requirements of real-time embedded systems, leading to the rise and adoption of static partitioning hypervisors. BAO is a static partitioning hypervisor, where the necessary resources are assigned to each Virtual Machine (VM) in the design. Hence, this hypervisor is not prepared for sharing peripherals between VMs, without failing to comply with the necessary requirements of MCSs.

Under this light, this thesis proposes the development of an infrastructure following a standard communication protocol, known as VirtIO, to communicate between two unprivileged VMs, keeping the hypervisor minimalistic and simple. With this infrastructure, there is a VM exclusively assigned and dedicated to sharing peripherals. Thus, whenever a VMs needs to perform an operation on a shared peripheral, it will have to make a request to the dedicated VM. To solve the limitation of sharing peripherals, it is necessary to relinquish part of the isolation between the dedicated VM and the other VMs. This thesis also proposes the implementation of two VirtIO devices, the virtio-console and the virtio-block, to test the infrastructure and, to share the Universal Asynchronous Receiver-Transmitter (UART) and Secure Digital (SD) Card peripherals.

Keywords: VirtIO, Input/Output (I/O) Virtualization, BAO, MCS, Hypervisor, Embedded Systems, ARM, Real-Time Operating System (RTOS), General Purpose Operating System (GPOS), Static Partitioning Hypervisors.

Resumo

Infraestrutura VirtIO para um Hypervisor de Partições Estáticas: *virtio-blk* e *virtio-console*

A indústria automóvel tem apostado, cada vez mais, na integração de sistemas que desempenham funções muito específicas. Ao mesmo tempo, há uma necessidade crescente de redução de SWaP-C e, consequentemente, a necessidade de agrupar subsistemas, normalmente com diferentes níveis de criticidade, nas mesmas plataformas de *hardware*. Atualmente, estes sistemas são referidos como MCS e necessitam de um forte isolamento entre os sistemas de criticidade diferentes para respeitarem a liberdade de interferência e prazos de tempo real.

A virtualização surgiu como um facilitador natural para MCS, visto que, para além de fornecer uma forma de integrar diferentes subsistemas na mesma plataforma de *hardware*, reforça o isolamento entre os mesmos. Seguindo essa tendência, os *hypervisors*, desenvolvidos para servidores e aplicações de *cloud*, foram adaptados para arquiteturas de sistemas embebidos, como a ARM. Devido ao facto destes *hypervisors* não cumprem os requisitos de sistemas embebidos de tempo-real, começamos a ver o surgimento e a adoção de *hypervisors* de partições estáticas. O Bao é um *hypervisor* de particionamento estático, onde os recursos necessários são atribuídos a cada VM em tempo de *design*. Por conta disso, este *hypervisor* não está preparado para a partilha de periféricos entre máquinas virtuais, sem deixar de cumprir os requisitos necessários dos MCS.

Dito isto, esta tese propõe o desenvolvimento de uma infraestrutura seguindo um protocolo de comunicação padrão, conhecido como VirtIO, para comunicar entre duas máquinas virtuais não privilegiadas, visando manter o *hypervisor* minimalista e simples. Com esta infraestrutura, existe uma VM exclusivamente atribuída e dedicada à partilha de periféricos. Assim sendo, sempre que uma VM necessitar de realizar uma operação num periférico partilhado, esta terá de realizar um pedido à VM dedicada. Para solucionar a limitação da partilha de periféricos é necessário abdicar de parte do isolamento entre a VM dedicada e as outras máquinas virtuais. Esta tese também propõe a implementação de dois periféricos VirtIO, o *virtio-console* e o *virtio-block*, para testar a infraestrutura e para partilhar os periféricos de UART e de cartão SD.

Palavras-chave: VirtIO, Virtualização, Bao, MCS, Hypervisor, ARM, RTOS, GPOS, Sistemas Embebidos, Partições estáticas Hypervisor.

Contents

List of Figures	xi
Acronyms	xv
1 Introduction	1
1.1 Motivation	1
1.2 Goals	3
1.3 Thesis Structure	4
2 Background, Fundamental Concepts and Related Work	5
2.1 Background and Functional Concepts	5
2.1.1 Virtualization	5
2.1.2 Virtual Memory	9
2.1.3 Inter-Partition Communication (IPC)	10
2.1.4 I/O Virtualization	11
2.1.5 ARMv8 Architecture	13
2.1.6 VirtIO	13
2.2 Related Work	16
2.2.1 VirtIO over Kernel-based Virtual Machine (KVM) Hypervisor	16
2.2.2 Para-virtualized Drivers and Quick EMUlator (QEMU) Emulation over Xen Hyper- visor	17
2.2.3 VirtIO over eXtensible Versatile hypervISOR (Xvisor) Hypervisor	18
2.2.4 VirtIO over ACRN Hypervisor	19
2.2.5 VirtIO for IPC in the Lightweight TrustZone-assisted (LTZ)Visor architecture .	20
2.2.6 Conclusion	21
3 Analysis and Design	22
3.1 VirtIO Analysis	22

3.1.1	Device status field	22
3.1.2	Feature bits	22
3.1.3	Notifications	23
3.1.4	Virtqueues	23
3.1.5	VirtIO over Memory-mapped Input/Output (MMIO)	29
3.1.6	VirtIO Block	30
3.1.7	VirtIO Console	32
3.2	VirtIO Infrastructure	34
3.2.1	Structural diagram	34
3.2.2	VirtIO Initialization	35
3.3	VirtIO under the perspective of the hypervisor	37
3.3.1	VirtIO mechanism using hypercalls	37
3.3.2	VirtIO mechanism using Trap-and-Emulate	38
3.4	VirtIO Block and Console	40
4	Implementation	42
4.1	VirtIO Infrastructure	42
4.1.1	VirtIO Initialization	42
4.1.2	VirtIO Write	46
4.1.3	VirtIO Read	48
4.1.4	VirtIO over MMIO	50
4.1.5	Trap-and-Emulate over Bao	51
4.2	VirtIO Console	61
4.2.1	Device Driver Application Programming Interface (API)	61
4.2.2	VirtIO Console Device Driver API	62
4.3	VirtIO Block	63
4.3.1	Device driver API	63
4.3.2	VirtIO Block Device Driver API	64
5	Functional validation and Testing	65
5.1	Validation: Setup	65
5.2	Validation: Methodology	65
5.3	Validation: VirtIO IPC	66
5.4	Validation: front-end for virtio-console device on QEMU	66
5.5	Validation: back-end of the Service Guest with front-end of the Linux for virtio-console	68
5.6	Validation: back-end of the Service Guest with Linux front-end for virtio-blk	72
6	Conclusion	74

6.1	Future Work	75
6.1.1	Split virtqueues	75
6.1.2	VirtIO devices	76
6.1.3	Platforms	76
6.1.4	FreeRTOS system	76
6.1.5	Performance evaluation	76
	Bibliography	78

List of Figures

1	MCS characteristics for each criticality level	2
2	Phases of operation for a system using Jailhouse virtualization	3
3	Bao ' s static partitioning architecture	3
4	System virtualization architecture	6
5	Difference of an attack on a virtualized system from a non-virtualized system	6
6	Difference between full and para-virtualization	7
7	Trap-and-emulate technique	8
8	Hypervisor topologies	9
9	Virtual memory accesses through the Memory Management Unit (MMU)	10
10	IPC mechanism inside hypervisor	10
11	I/O virtualization using emulation	12
12	I/O virtualization using pass-through	12
13	ARMv8 architecture exception level in the non-secure and secure world	13
14	Driver abstractions with VirtIO	14
15	VirtIO architecture	14
16	Peripheral sharing mechanisms in KVM architecture	16
17	Peripheral sharing mechanisms in Xen architecture	17
18	Peripheral sharing mechanisms in Xvisor architecture	18
19	Peripheral sharing mechanisms in ACRN architecture	19
20	VirtIO mechanism for IPC in LTZVisor architecture	20
21	Management of VirtIO virtqueues	24
22	Descriptor table – circular array with constitution of the descriptors	25
23	Available ring - buffers stored to be sent to back-end	26
24	Used ring - buffers to be sent to front-end	26
25	Descriptor ring - circular array with constitution of the descriptors	28
26	Structure relationship diagram design	34

27	Driver initialization design flowchart	35
28	Device initialization design flowchart	36
29	VirtIO hypercall mechanism design	37
30	VirtIO MMIO mechanism design	38
31	VirtIO MMIO mechanism design – sequence diagram	39
32	VirtIO block device design	40
33	VirtIO console device design	41
34	Test front-end on QEMU diagram	67

List of Listings

1	VirtIO split virtqueue structure layout	25
2	VirtIO block device configuration	31
3	VirtIO block device request	32
4	VirtIO console device configuration	33
5	VirtIO front-end initialization	43
6	VirtIO back-end initialization	44
7	VirtIO front-end feature bits negotiation	45
8	VirtIO back-end feature bits negotiation	45
9	VirtIO front-end/back-end write data	46
10	VirtIO front-end/back-end write descriptor on virtqueue	47
11	VirtIO front-end/back-end notify macros	47
12	VirtIO front-end/back-end write descriptor in the shared memory	48
13	VirtIO get descriptor macro	48
14	VirtIO front-end read descriptor from a virtqueue	49
15	VirtIO back-end read descriptor from a virtqueue	49
16	MMIO device register layout	50
17	VirtIO linkage initialization over Bao	51
18	VirtIO device parameters struct over Bao	52
19	VirtIO VM initialization	53
20	VirtIO service guest Central Processing Unit (CPU) initialization	53
21	VirtIO IPC initialization	54
22	VirtIO MMIO emulation handler	55
23	VirtIO message handler	56
24	VirtIO service guest notification handler	56
25	VirtIO pooling handler	57
26	VirtIO hypercall in the hypervisor	57
27	VirtIO pooling operation case	58

28	VirtIO asking operation case	58
29	VirtIO write and read operation case	59
30	VirtIO write/read operation to the struct	59
31	VirtIO interrupt operation case	60
32	VirtIO sends message to CPU	60
33	VirtIO message to a CPU handler	61
34	VirtIO console MMIO handler	63
35	VirtIO MMIO disable legacy on QEMU	67
36	Enable packed virtqueue on VirtIO	67
37	Extract devicetree from QEMU command	68
38	Convert from devicetree format to devicetree source	68
39	QEMU devicetree source	69
40	Bao configuration	70
41	Service guest configuration	70
42	Linux guest configuration	71
43	Linux devicetree changes	71
44	Block device driver test	72
45	Service guest configuration - block device added	72

Acronyms

ABI	Application Binary Interface 13
API	Application Programming Interface ix, 8, 17, 20, 34, 41, 42, 61, 62, 63, 64
ARM	Advanced RISC Machine vi, vii, viii, xi, 2, 13, 17, 18, 65, 66
CAN	Controller Area Network 76
CPU	Central Processing Unit xiii, xiv, 27, 39, 40, 52, 53, 54, 56, 59, 60, 61, 62, 68, 73
EL	Exception Level 13
FIFO	First In, First Out 61
GPIO	General-Purpose Input/Output 76
GPOS	General Purpose Operating System vi, vii, 20
HVM	Hardware Virtual Machine 17
I/O	Input/Output vi, xi, 3, 4, 5, 11, 12, 14, 15, 17, 18, 19, 20, 32, 76
I2C	Inter-Integrated Circuit 76
IOMMU	Input/Output Memory Management Unit 11
IPC	Inter-Partition Communication viii, ix, xi, xiii, 4, 5, 10, 11, 14, 20, 52, 54, 65, 66
ISR	Interrupt Service Routines 66
JTAG	Joint Test Action Group 73
KVM	Kernel-based Virtual Machine viii, xi, 2, 16, 18, 21, 74

LTZ	Lightweight TrustZone-assisted viii, xi, 20
MCS	Mixed-Criticality System vi, vii, xi, 1, 2, 6, 14, 20, 74
MMIO	Memory-mapped Input/Output ix, xii, xiii, xiv, 15, 29, 30, 35, 36, 37, 38, 39, 42, 45, 47, 50, 55, 57, 62, 63, 64, 65, 66, 67, 68, 73, 75
MMU	Memory Management Unit xi, 9, 10
OS	Operating System 5, 6, 7, 8, 9, 10, 11, 13, 19, 20, 21, 65
PA	Physical Address 9, 10
PCI	Peripheral Component Interconnect 15
PL	Programmable Logic 65
PS	Processing System 65
PV	ParaVirtualization 21
QEMU	Quick EMUlator viii, ix, xii, xiv, 4, 16, 17, 21, 42, 61, 65, 66, 67, 68, 69, 75
RAM	Random-Access Memory 9, 10
RISC	Reduced Instruction Set Computer 76
RISC-V	Reduced Instruction Set Computer version 5 2
RT	Real Time 14, 65
RTOS	Real-Time Operating System vi, vii, x, 1, 13, 20, 76
SD	Secure Digital vi, vii, 30, 63, 64, 66, 72, 73, 75
SoC	System-on-Chip 1
SWaP-C	Size, Weight, Power and Cost vi, vii, 1, 5
TCP/IP	Transmission Control Protocol/Internet Protocol 10
TZ	TrustZone 20
UART	Universal Asynchronous Receiver-Transmitter vi, vii, 32, 41, 61, 62, 66, 67, 68, 69
VA	Virtual Address 9
vCPU	Virtual Central Processing Unit 8
VM	Virtual Machine vi, vii, xiii, 3, 5, 6, 7, 8, 11, 19, 21, 52, 53, 54, 60, 74
VMM	Virtual Machine Monitor 7, 19

WFI Wait For Interrupt 73

Xvisor eXtensible Versatile hypervISOR viii, xi, 18, 21, 74

Introduction

This chapter is organized as follows: Section 1.1 explains the scientific motivation for this thesis, Section 1.2 explains the goals and objectives, and Section 1.3 explains how the chapters are structured, their contents and their purpose.

1.1 Motivation

Presently, we are increasingly surrounded by the most diverse technologies. Such technologies (automotive infotainment systems, smartphones, autonomous cars, etc.) have become indispensable in human life [1]. With the high need to increase and improve the functionalities of these technologies, the systems that integrate them have to and guarantee the requirements for their proper functioning. These systems are commonly referred to as embedded systems.

Embedded system engineers are being, constantly, pressured by different markets to reduce the SWaP-C and, consequently, subsystem consolidation measures have been taken, usually with different criticality levels, within the same hardware [2–5]. Furthermore, these systems are designed as MCS to balance the conflicting requirements of isolation for security and safety (Figure 1). Those requirements are critical and fundamental for several industries such as automotive and medical where the integrated systems have different criticalities. For example, in a car, it is very common for the infotainment system to be close to the ABS. With virtualization, it is possible to integrate components with different criticality levels onto modern multicore System-on-Chip (SoC) [3, 5–7]. For the infotainment system, a non-safety critical system, Linux is used; alternatively for ABS, a safety-critical system, RTOS is used.

Although virtualization technology emerged in the 60s, it was not adopted until the early 2000s. Technologies that enable virtualization, such as hypervisors, were developed to provide simultaneous access. While the idea sounds enticing, several technologies have emerged that have grown in popularity more than virtualization.

Virtualization was forgotten until the emergence of physical servers where enterprises had to contain underutilized physical hardware, where each server could only provide a single specific task. It was to solve this problem that virtualization was almost reborn. Virtualization has enabled companies to optimize their

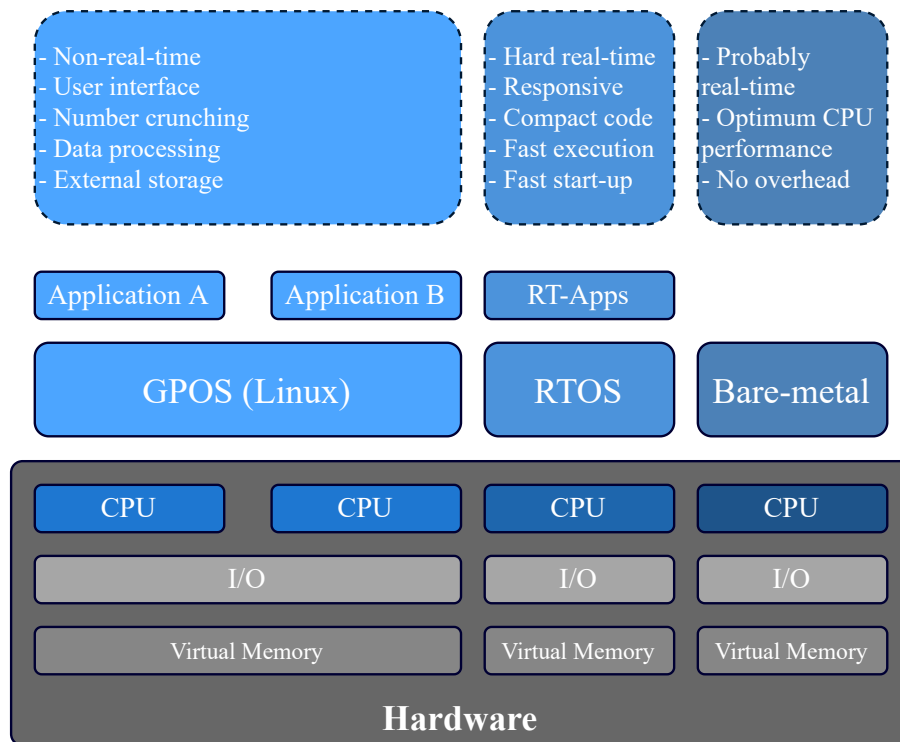


Figure 1: MCS characteristics for each criticality level

hardware, making servers more efficient and reduce costs, configuration, and maintenance. Currently, virtualization does not present solutions exclusively for servers, there has been a vast interest in its integration in different areas. In the scope of this thesis, virtualization appears as a natural solution for MCSs for its very desirable features, including isolation, high availability, workload balancing [8], etc.

For this solution to be possible, hypervisors, initially developed for the domain of servers, such as KVM [9, 10] and Xen [9, 11, 12], had to adapt to embedded architectures (mainly ARM). Unfortunately, these hypervisors have been adapted and not designed from scratch, that is, add-ons were created which do not fully meet the constraints and real-time requirements, essential for the proper functioning and veracity of the system [3, 5]. Nowadays, those hypervisors are not the most viable solution for MCS as, recently, some hypervisors were created specifically for MCS, such as the Jailhouse hypervisor created by Siemens' [13]. While KVM is a Linux-based and open-source full virtualization hypervisor and Xen is a different open-source full virtualization hypervisor, that, unlike KVM, supports both full virtualization and static partitioning and does not require Linux. The Jailhouse is an open-source static partitioning hypervisor [13] that, unfortunately, needs a normal Linux system to load and configure the hypervisor (see Figure 2), and its management is based on Linux infrastructures [13, 14].

Bao is a real-time static partitioning lightweight baremetal hypervisor that follows a similar approach to Jailhouse concerning the preference of a minimalist hypervisor. Designed for MCSs, supporting ARMv8 and Reduced Instruction Set Computer version 5 (RISC-V), it strongly focuses on isolation for real-time

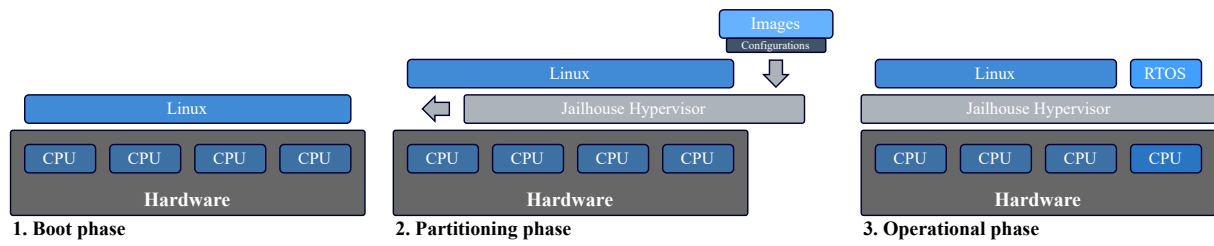


Figure 2: Phases of operation for a system using Jailhouse virtualization

behaviour, security, and safety. As mentioned, Bao architecture follows a static partitioning approach, i.e, resources are statically partitioned and assigned at the VM initialization time, on the system boot [3, 7]. Bao assigns peripherals to guests in a pass-through only I/O configuration. With this in mind, an obvious limitation appears: Bao cannot share peripherals between guests.

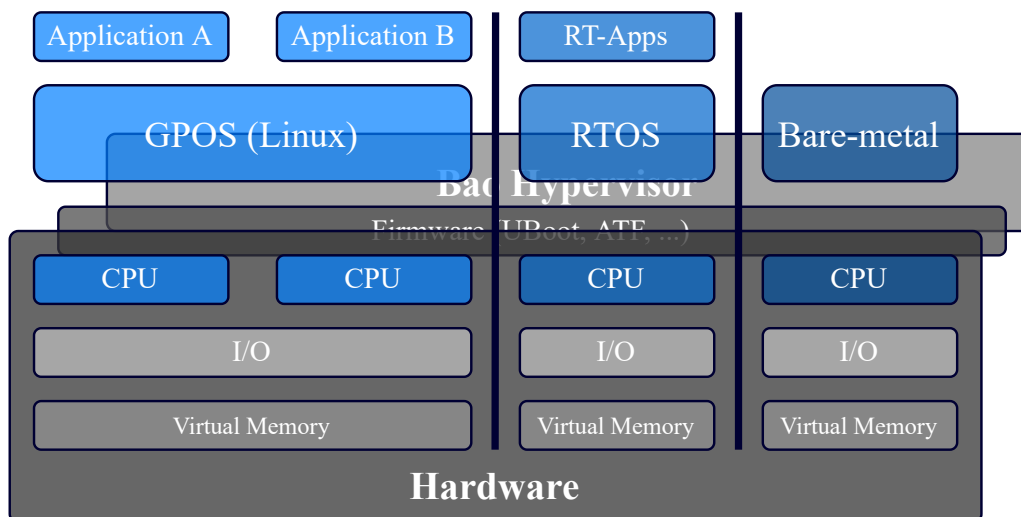


Figure 3: Bao's static partitioning architecture

1.2 Goals

The core concept of this thesis lies in the proposal of a solution to the aforementioned limitation through the implementation of an infrastructure that allows the sharing of devices between several guests. The goals to be achieved are summarized as follows:

1. Implement an infrastructure according to a standard communication protocol, known as VirtIO, to communicate between two non-privileged guests.
2. Implement the back-end and front-end driver to a console device.

3. Implement the back-end driver to a block device.
4. Test the front-end with the back-end of the QEMU to ensure the functionality of the VirtIO infrastructure implemented.
5. Test the back-ends of the console and block drives with the Linux front-ends drivers to ensure compatibility and functional working.

1.3 Thesis Structure

This section lays out the structure of the present work, i.e. the chapters, their contents, and purpose. The structure is the following:

1. **Introduction** (currently chapter): explores the core motivation of this thesis, along with the layout of the goals and the overall structure.
2. **Background, Fundamental Concepts, and Related Work**: covers the theoretical fundamentals and other works related to this thesis. Theoretical concepts cover virtualization, IPC, I/O Virtualization, and VirtIO.
3. **Analysis and Design**: analyses the VirtIO protocol, explaining most of the concepts used in more depth. The infrastructure and devices are designed based on what was analysed.
4. **Implementation**: presents the mechanisms that were designed, emphasizing the trap-and-emulate mechanism. The implementation of the block and console devices is also presented, in coherence with the implemented mechanism.
5. **Evaluation**: presents the tests that were carried out during the different phases of the implementation of the VirtIO infrastructure and the tests that were carried out on the implementation of the devices.
6. **Conclusion**: provides a summary and a conclusion of what was done, along with suggestions for future work to improve the implementation and performance of the VirtIO mechanism.

Background, Fundamental Concepts and Related Work

This chapter provides an overview of the background and related work in the field of I/O virtualization, specifically in the context of embedded systems. Section 2.1 explains the concept of virtualization and how it applies to embedded systems, including an introduction to the topic of inter-process communication and the importance of I/O virtualization in embedded virtualized systems. Additionally, an introduction to the VirtIO protocol is provided. In Section 2.2, various I/O virtualization implementations in different domains such as hypervisors and IPC are discussed.

2.1 Background and Functional Concepts

2.1.1 Virtualization

Virtualization of hardware platforms is a widely used technology, and underpins almost all modern cloud computing that saves resources by consolidating multiple servers onto a single physical machine [9, 15–17]. Virtualization is used by developers to run multiple Operating System (OS)s on a single machine, by sharing the hardware (see Figure 4), and to test software without the risk of damaging the main computing environment. Virtualization is widely used in server systems, and support for it is a requirement for most server-grade processors. This is because virtualization provides numerous benefits to data centres, including:

- **Isolation** – Virtualization provides isolation between VM running on the same physical hardware. This allows the sharing of different levels of criticalities such as, non-safety and safety systems. For example, in automotive systems, network-connected infotainment is deployed alongside safety-critical control systems [3, 18]
- **Workload balancing** – To satisfy the market SWaP-C requirements, it is necessary to use each hardware platform as much as possible. This can be achieved using migration of virtual machines,

or by co-hosting workloads on physical hardware. This means that the physical hardware is used for as much of their capacity as possible.

- **Sandboxing** – VMs can be used to provide sandboxes for applications that might interfere with the rest of hardware.

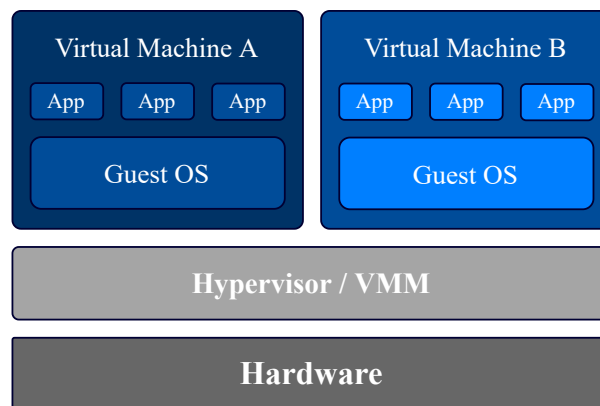


Figure 4: System virtualization architecture

One of the most common requirements of MCSs is the security of the mixed systems, and virtualization can provide that security. A virtual machine encapsulates a subsystem so that it interchangeably cannot interfere with other subsystems. For example, should the OS be compromised, by an application program that exploits a buffer or stack overflow in the kernel, any software running on top can be subverted, as shown on the left. Encapsulating a subsystem into a VM protects other subsystems from a compromised OS [19].

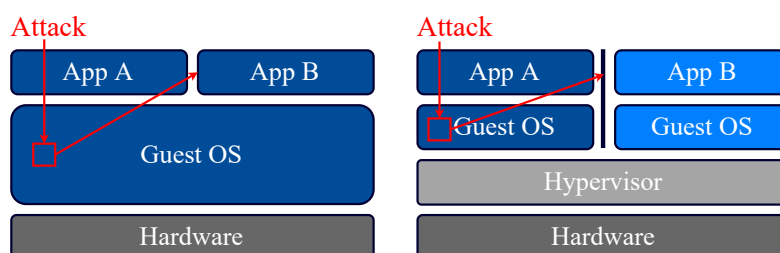


Figure 5: Difference of an attack on a virtualized system from a non-virtualized system

In the absence of virtualization, the high-level OS runs in privileged mode, and therefore, once compromised, can attack any part of the system. With virtualization, the high-level OS is less-privileged and unable to interfere with data belonging to other subsystems (see Figure 5), and its access to the processor can be limited to ensure that real-time components meet their deadlines.

Virtualization, therefore, consists of an software abstraction layer, commonly referred to as Virtual Machine Monitor (VMM) or Hypervisor. It serves as an interface between the VM and the physical hardware, ensuring each has access to the physical resources it needs to run, assigning each its slice share of the underlying computing power, memory, and storage. As mentioned previously, this prevents the VMs from interfering with each other, however, for that to happen the hypervisor needs to run at a more privileged level than the VMs, acting as guests, that run above it. These guests can be OSs or simple baremetals applications, and they may or may not have to be changed to run in a virtualized environment. This lays out the distinction between the full virtualization and para-virtualization hypervisors (see Figure 6), where [15, 20, 21]:

- **Full virtualization** – the guest OS runs on top of a hypervisor that sits on the baremetal. The guest is unaware that it is being virtualized and requires no changes to work in this configuration. For that, the hypervisor needs to emulate the hardware. This technique offers the best isolation and security for virtual machines.
- **Para-virtualization** – the guest OS is not only aware that it is running on a hypervisor, but includes code to make guest-to-hypervisor transitions more efficient. To do this, it is necessary to modify the guest, replacing non-virtualizable instructions with hypercalls that communicate directly with the virtualization layer hypervisor. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping.

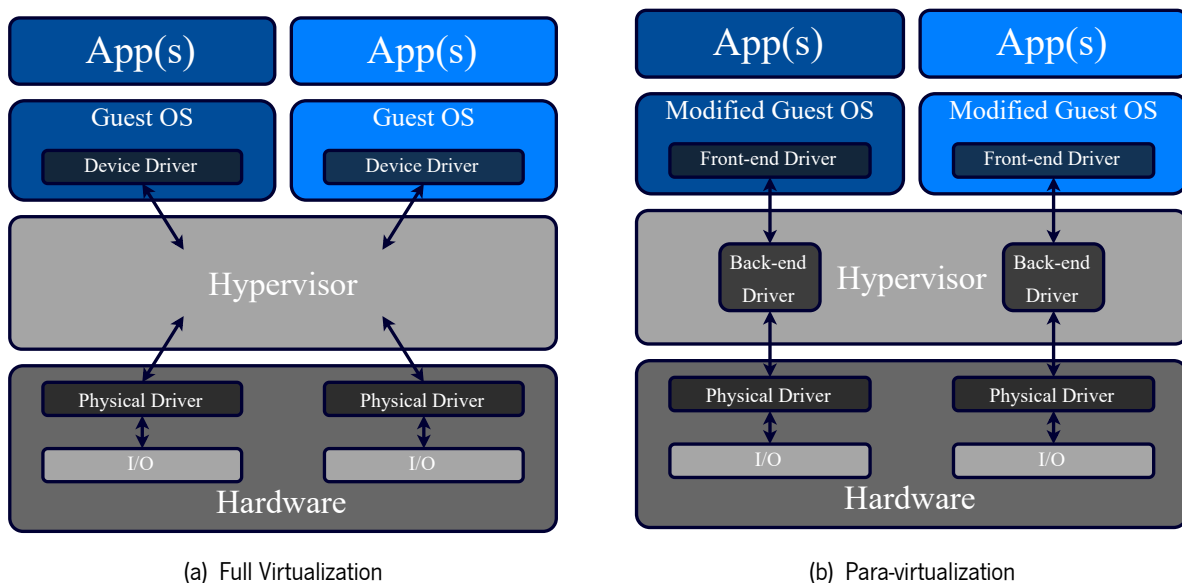


Figure 6: Difference between full and para-virtualization

Those methods have their advantages, the most obvious of which are the easiness with which one can incorporate guests in full virtualization and the performance boost from having a hypervisor compliant API in para-virtualization. Both, have also the possibility of using generic techniques, as:

1. **Emulation** is the process in which virtualization software emulates part of the hardware that is independent of the physical hardware to which the emulation corresponds. This technique provides sufficient VM portability and hardware compatibility, meaning it is a very easy technique to integrate into any virtualization system. The software that emulates hardware is typically based on interpretation, allowing the guest operating system to operate without being aware of the virtualization. However, this approach has the drawback of high computational costs, making it less commonly used in virtualization.
2. **Trap-and-Emulate** is an emulation-based technique with improved performance. Like the emulation technique, the application and OS run in User Mode, while the hypervisor runs in Privileged Mode. However, the applications in the referred technique execute the instructions natively in hardware: when an application or operating system executes a privileged instruction, a trap is generated for the hypervisor, passing control of the execution. In short, all privileged instructions are executed by the hypervisor (see Figure 7). After the hypervisor takes control of the execution due to a trap, an emulation is transmitted to the Virtual Central Processing Unit (vCPU) returning the control to User Mode, that is, to the application or OS.

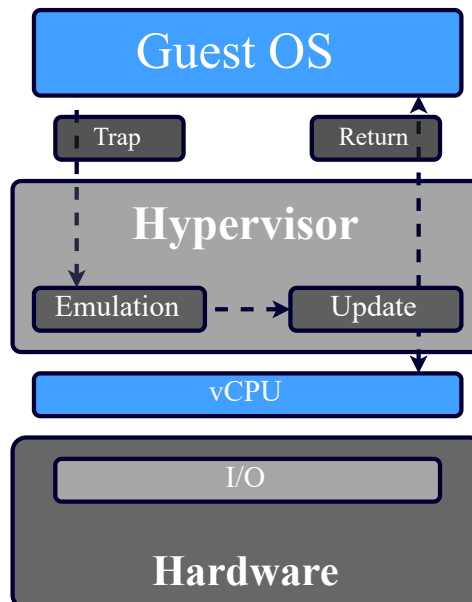


Figure 7: Trap-and-emulate technique

Hypervisor topologies can be essentially classified into two types, depending on the position of the virtualization layer in the software stack (see Figure 8):

- **Type-1 or baremetal** hypervisors run directly over the hardware in a higher privileged mode than its guests. They have complete control over hardware resources and arbitrate access to hardware resources among multiple guest OSs. As a consequence, performance will be highly influenced by the hypervisor, making this option better for systems with rigid time constraints, such as embedded system applications [22, 23].
- **Type-2 or hosted** hypervisors do not directly control hardware since they run inside a host OS as a typical process, meaning that the hypervisor does not run directly above the hardware layer. This type of hypervisor normally cannot access or perform any operation on the hardware directly, this responsibility is given to the host OS, which results in a lower performance when compared with type-1 hypervisors [22, 23].

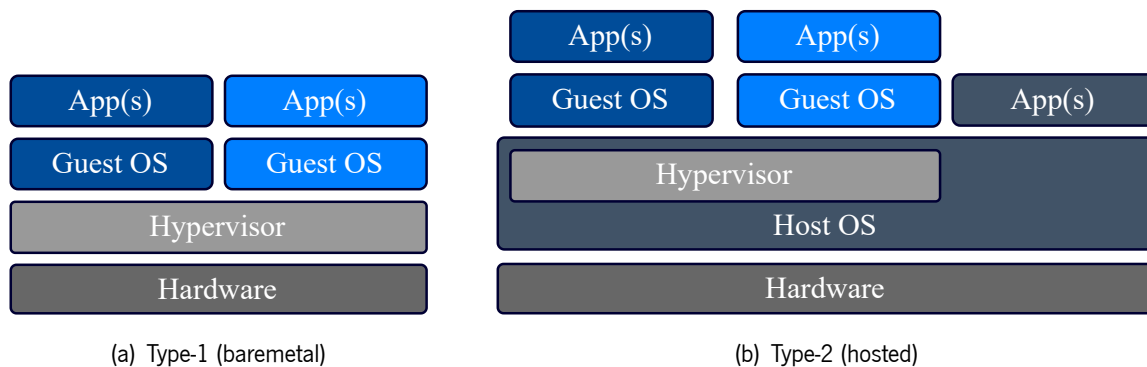


Figure 8: Hypervisor topologies

2.1.2 Virtual Memory

In virtualized systems, guest operating systems require their own distinct memory space. During the development of virtualized systems, several challenges arose related to sharing physical memory among guests and among applications within operating systems, such as lack of memory, gaps between memory sections of applications, and security concerns with allocated memory. To address these issues, a mechanism was developed to abstract the system from physical memory, resulting in the concept of virtual memory. This led to the introduction of virtual addresses Virtual Address (VA), which the guest has access to, and Physical Address (PA), which the MMU uses to communicate with Random-Access Memory (RAM), as shown in Figure 9. To access the data from the Guest, the following steps occur:

1. The Guest executes an access from the VA;
2. The MMU translates the VA to the PA;

3. (If the PA is not in memory, the needed data is loaded to the RAM);
4. The MMU reads from the RAM using the PA and returns the data to the guest.

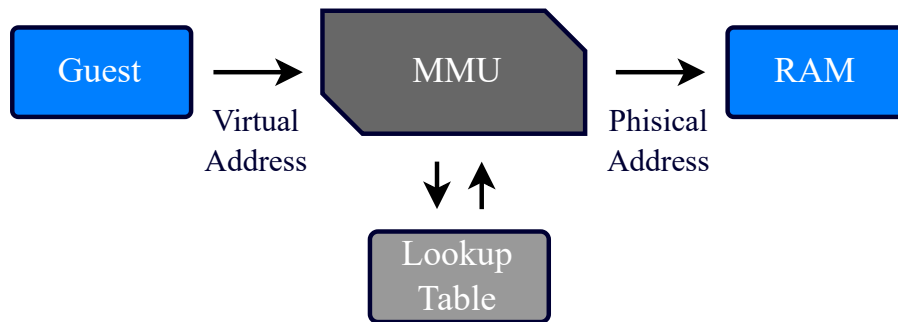


Figure 9: Virtual memory accesses through the MMU

2.1.3 IPC

IPC is a fundamental component of any OS as it allows data sharing between OSs. In a virtualization environment, there is a need for guests to communicate with each other. Usually, the most used communication mechanisms are through the Transmission Control Protocol/Internet Protocol (TCP/IP). However, this mechanism is impractical for embedded systems hypervisors, where communications have to be carried out in the shortest time possible. Therefore, the best strategy is to go down to the hypervisor layer to manage the communication between guests through IPC (see Figure 10).

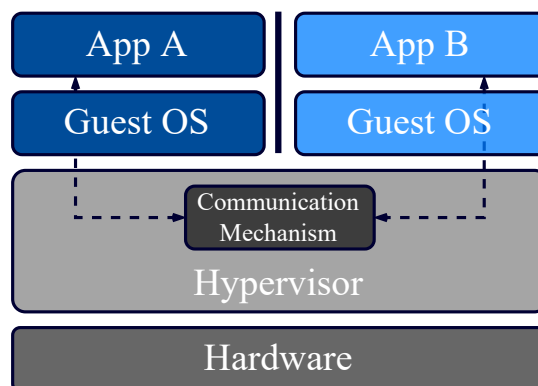


Figure 10: IPC mechanism inside hypervisor

Maintaining the isolation is one of the main requirements of the system, as such, the IPC mechanism will have to ensure that the guests remain separated while sharing data. IPC mechanisms can be classified

by: synchronization, privilege level, and data transfer mechanism. In the scope of this thesis, it is important to explain the different data channel mechanisms. They can be divided into two types:

- **Shared Memory** consists of an allocated block of memory that various system partitions can have access to, as a data channel to the IPC mechanism. It is a more efficient policy, but less secure as it needs to allocate a block of memory accessible by more than one partition, making it an easy target for buffer access [24]. However, there are steps that can be taken to address this problem, mainly with an access control strategy supported by a kind of execution privilege level. Although most systems implement their design for a shared memory IPC engine, more and more solutions are using the VirtIO technology standard [25, 26].
- **Direct Transfer** consists of copying data directly between the sent and receive buffers, thus reducing an extra copy of the shared memory approach [27].

2.1.4 I/O Virtualization

In embedded systems it is essential to share physical I/O devices across workloads. Similarly, in embedded system virtualization, the hypervisor has to ensure a secure virtual interface to access physical I/O devices shared between guests. The most difficult challenge in embedded virtualization is allocating, protecting, sharing and ensuring the efficiency of I/O devices [28].

To virtualize I/O devices, there are several methods, namely:

- Emulation.
- Pass-through.
- Shared Input/Output Memory Management Unit (IOMMU).
- Secure I/O Virtualization within Microkernels.

The most relevant methods will be discussed in the following sections.

2.1.4.1 Emulation

The traditional method of I/O virtualization is emulation, consisting of all guest OS accesses to device I/O resources being intercepted, validated, and translated into hypervisor initiated operations (see Figure 11).

This method maximizes reliability, security, and shareability. The guest cannot corrupt the system through the I/O device because all I/O accesses are protected via the hypervisor device driver. With emulation, it is possible to share a single device across multiple VM, and if one fails, the other VMs can continue to utilize the same physical I/O device. The biggest problem is the efficiency, since the emulation layer creates a significant overhead on all I/O operations.

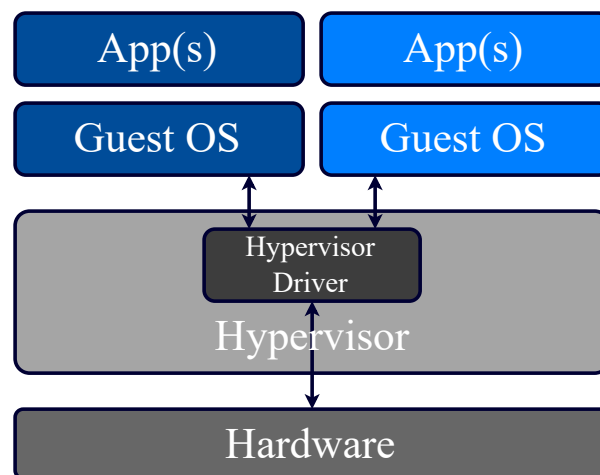


Figure 11: I/O virtualization using emulation

2.1.4.2 Pass-through

In contrast to emulation, a pass-through method gives the guest direct access to a physical I/O device (see Figure 12). This device can be shared between guests through a virtual I/O interface between the guest that owns the physical I/O device and another guest that requires the access of the device.

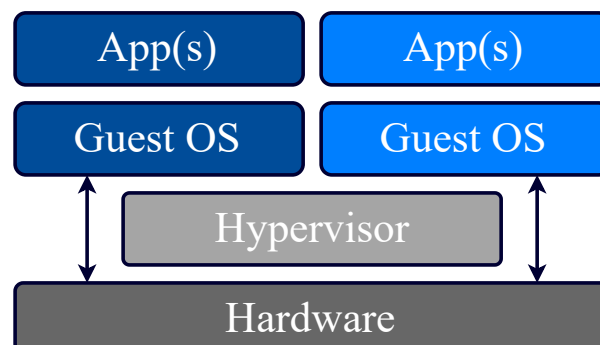


Figure 12: I/O virtualization using pass-through

The pass-through method can be very efficient but dangerous, as improper access by one guest can block another guest, an application, or even the entire system. Since one of the main characteristics of virtualization for embedded systems is the isolation between the systems, sharing devices between guests through pass-through can, at any time, violate the constraints required by these systems. For these reasons, hypervisors do not usually share devices using this method. Regarding Bao, as mentioned previously, devices are assigned exclusively to each guest without sharing devices.

2.1.5 ARMv8 Architecture

ARMv8 processors are the latest generation of 64-bit architectures. This processors have two security stages, Secure and Non-secure. In Non-secure stages, also referred to as Normal world, the execution levels occur in four levels, as illustrated in the Figure 13. The levels go from Exception Level (EL)0 to EL3, where EL0 has the lowest privilege level and EL3 the highest privilege level. Software that typically run at different exception levels are EL0 with normal user application, EL1 with OS kernel, EL2 with Hypervisor and the EL3 with Low-level firmware, including the Secure Monitor. Otherwise, the Secure stage have a trusted OS that runs alongside the guests OSs on the same hardware and offers protection against certain software and hardware attacks.

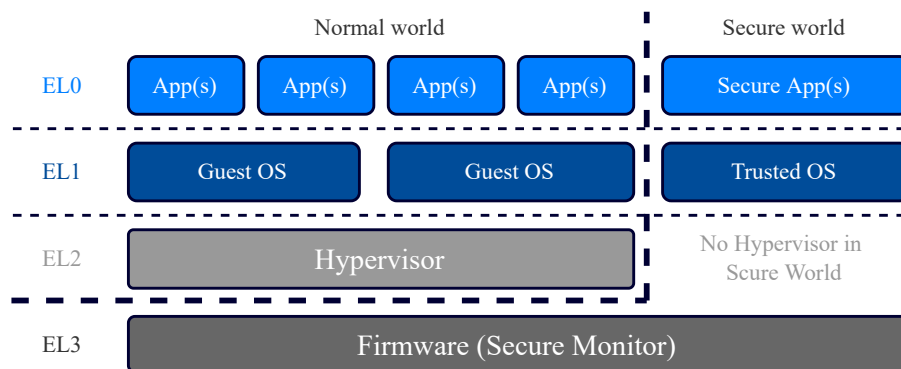


Figure 13: ARMv8 architecture exception level in the non-secure and secure world

2.1.6 VirtIO

With the emergence of the Linux Kernel, numerous virtualization solutions were developed, each with its drivers with different features and optimizations to communicate with different devices such as block, network, and console [29]. With this problem in mind, Rusty Russell solved, in cooperation with IBM, a series of Linux drivers that can be adapted to different hypervisors using shim layers. It was then that the VirtIO abstraction layer began to be implemented. In addition to drivers, VirtIO included simple functionality mechanisms for each of the driver and provided a ring buffer to carry the communication data. Initially, VirtIO was integrated and was developed for Linux systems, but later through the OpenAMP project, VirtIO was added to guests with baremetal systems and RTOS [30].

The VirtIO project was developed to achieve three initial goals:

- Remove implementations of other parts in Linux Kernel.
- Create a common Application Binary Interface (ABI) for general publication and good use of buffers.
- Use the virtio_ring infrastructure itself and the Linux API for virtual I/O devices (probing and configuration).

Currently, in addition to having been considered a standard abstraction layer above hardware in a para-virtualization environment, VirtIO provides front-end drivers applied to guests and back-end drivers built into the hypervisor allowing it to communicate with the physical hardware (see Figure 14), is also used for IPC [21, 25, 26].

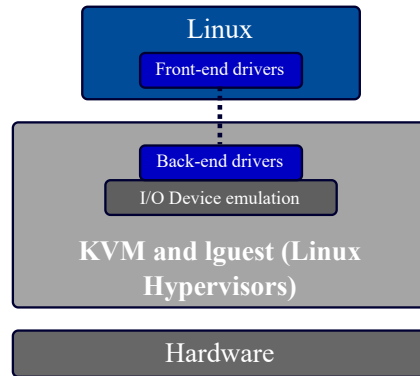


Figure 14: Driver abstractions with VirtIO

In virtIO architecture (see Figure 15), there are two layers specifically defined to support communication between the front-end and back-end: the infrastructure and transport layers. The infrastructure layer is a virtual queue interface that connects front-end drivers to back-end drivers. Drivers can use zero or more queues, depending on their needs. For example, the VirtIO block driver uses, at least, one virtual queue, whereas the VirtIO console driver uses two virtual queues (one for receiving and one for transmitting).

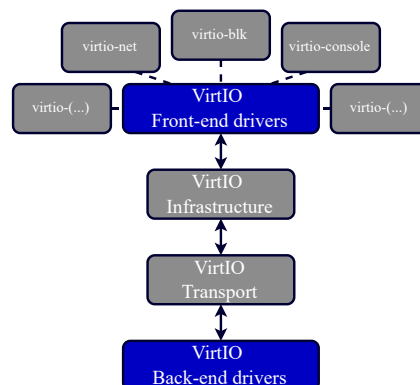


Figure 15: VirtIO architecture

The virtualization of I/O devices, especially when shared, introduces an increase in overhead to the system, this overhead, with Virtio, is reduced. Due to this fact, VirtIO is a very viable solution when it comes to Real Time (RT) embedded systems [31]. In the context of this thesis, VirtIO will be implemented so that it is possible to share devices without a big break in the isolation between MCS and a big increase in overhead.

To make a good abstract API, it was necessary to create a transport abstraction for all devices. Each VirtIO device contains the following parts [32]:

- Device status field.
- Feature bits.
- Notifications.
- Device Configuration space.
- One or more virtqueues.

In chapter 3, the topics will be covered in more detail, including how the driver (front-end) and device (back-end) begin communicating.

2.1.6.1 VirtIO Transport

The transport layer of a VirtIO device is a bus layer interface that is used by the driver for accessing basic facilities. For example, a VirtIO device implemented with Peripheral Component Interconnect (PCI) registers that could be accessed via PCI configuration space or MMIO, or VirtIO PCI driver will use those registers to set up the devices. VirtIO supports three transport options:

- PCI.
- MMIO
- Channel I/O.

2.1.6.2 Block Device

The virtio-blk device is a simple virtual block device to establish communication with a disk, an SD Card, etc. The driver places read, write, and other requests onto the virtqueue, so that the device can process them accordingly. The VirtIO device ID of the virtio-blk is 2, and it supports multiple virtqueue called *requestq*.

2.1.6.3 Console Device

The virtio-console is a simple device for data input and output. The VirtIO device ID of the virtio-console is 3 and can have many ports. Each port has a pair of input and output virtqueues used to communicate information between the driver and the device and to control them, virtio-console have two control virqueues, one for input and the other for output. This virqueues is just used if multiport is available.

2.2 Related Work

As expected, most hypervisors have adopted different techniques to perform peripheral sharing between guests. In the following sections, we describe relevant implementations of peripheral sharing will be presented.

2.2.1 VirtIO over KVM Hypervisor

KVM is an open-source hypervisor that supports both full virtualization and para-virtualization. This hypervisor primarily provides full virtualized guests and provides para-virtualization in the form of optional VirtIO devices. KVM is deeply integrated into the Linux Kernel with a module called `kvm.ko`. The host views a virtual machine as a QEMU process [33, 34].

QEMU utilizes the VirtIO mechanism for sharing hardware between guests, as shown in Figure 16. As the guest memory is mapped in the QEMU process, communication with the hardware is achieved by filling in the front-end virtqueue and sending a notification to KVM. Since KVM is integrated into Linux, it can utilize either the split or packed method.

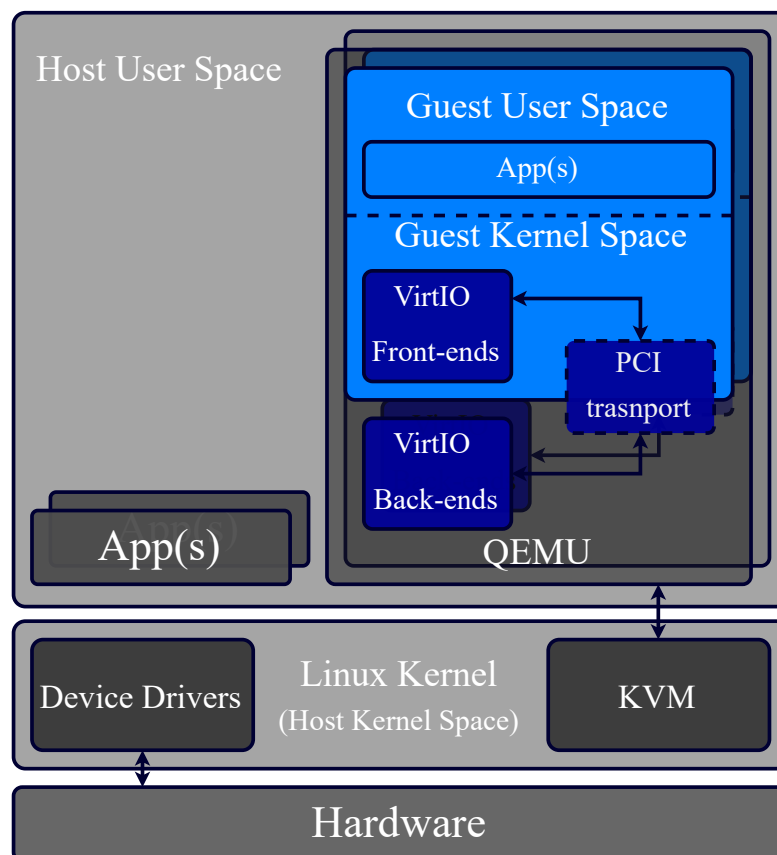


Figure 16: Peripheral sharing mechanisms in KVM architecture

2.2.2 Para-virtualized Drivers and QEMU Emulation over Xen Hypervisor

Xen is an open-source type 1 hypervisor with support for ARM that allows both full virtualization and para-virtualization guests. The core of the Xen hypervisor operates directly on top of hardware and hosts several guests called domains (see Figure 17).

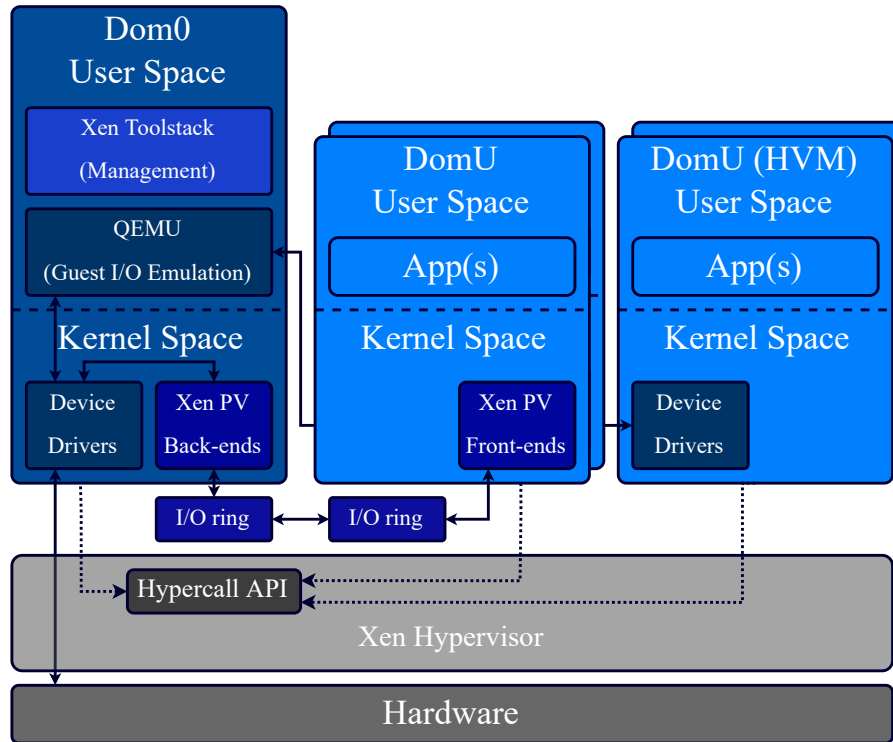


Figure 17: Peripheral sharing mechanisms in Xen architecture

The management domain, called Dom0, is a modified version of the Linux Kernel running all the management tools. With that, the hypervisor remains simple. The main purpose of Dom0 is to provide I/O virtualization services, and for that, it must have privileges over other DomU guests. There are two types of unprivileged guests: para-virtualized guests and Hardware Virtual Machine (HVM) guests.

Para-virtualized guests do not interact with the real hardware. Instead, the guest kernel communicates directly with the hypervisor using the hypercall API. The para-virtualized guest also requires virtual hardware devices. These are implemented in two parts, the front-end and back-end. The front-end driver runs in DomU and performs the function of a hardware device driver. When it is necessary to make a request to the hardware, the front-end must establish communication, through a mechanism called XenBus, which sends a request to the back-end driver hosted in the Dom0 guest. In comparison, HVM guests also don't have access to the physical hardware device. To enable this, Xen uses device emulation offered by QEMU. By default, each HVM guest has a corresponding QEMU process running in Dom0 guest. Para-virtualized drivers use a physical memory page shared between the Dom0 and DomU guests. This shared memory is used to implement I/O rings that send data between para-virtualized drivers [33, 34].

2.2.3 VirtIO over Xvisor Hypervisor

Xvisor is a hypervisor that supports both full virtualization and para-virtualization in the form of optional VirtIO devices. It aims to provide a lightweight hypervisor that can be used within embedded systems with less overhead and a small memory footprint [33]. Unlike KVM and Xen hypervisors, Xvisor ARM does not sustain any additional scheduling or context switch overhead in emulating guest I/O events. The front-ends of VirtIO are allocated on the guest kernel space and the back-ends are allocated on the hypervisor (see Figure 18). This allows the hypervisor to have access to the memory of all guests, facilitating communications.

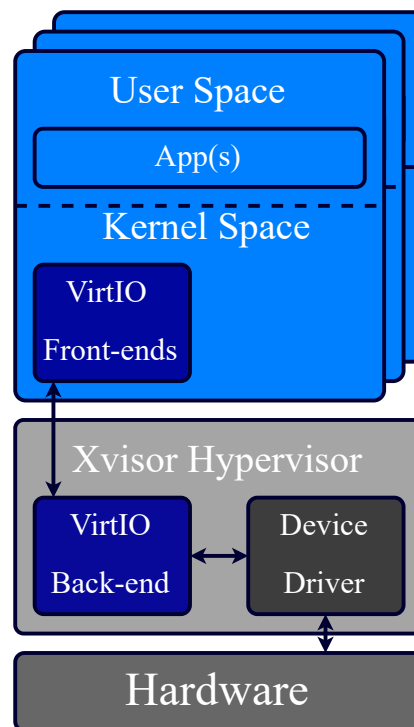


Figure 18: Peripheral sharing mechanisms in Xvisor architecture

2.2.4 VirtIO over ACRN Hypervisor

ACRN hypervisor is a type 1 hypervisor, running directly on baremetal hardware. It implements a hybrid VMM architecture, using a privileged service VM, that manages the I/O devices. This hypervisor provides a virtual machine, called service OS, to provide services to the other virtual machines (see Figure 19). To communicate between the different virtual machines and the service virtual machine, they will have to have a shared memory where the virtqueues referring to the VirtIO protocol are placed. ACRN provides its own hypercalls to establish communication between the guest service and the rest. The method currently implemented to provide VirtIO communication is split [35].

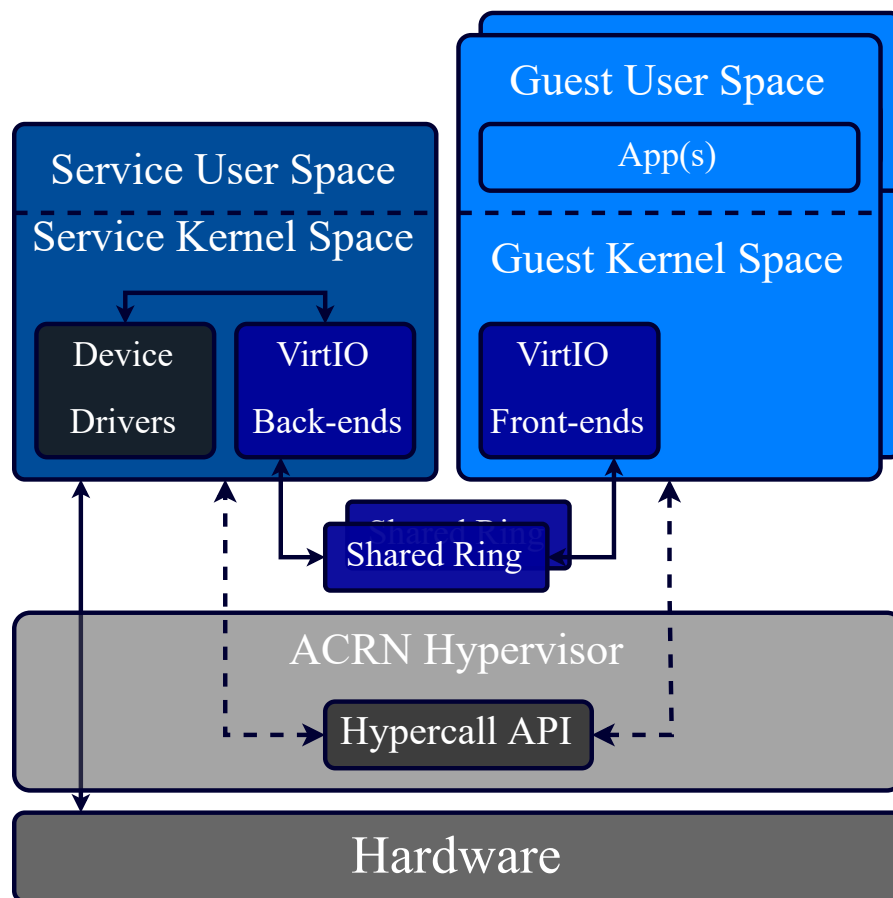


Figure 19: Peripheral sharing mechanisms in ACRN architecture

2.2.5 VirtIO for IPC in the LTZVisor architecture

The LTZVisor is designed to consolidate MCS. This hypervisor uses the dual-guest OS configuration which distributes the GPOS to the non-secure world while the RTOS is hosted in the secure world. LTZVisor runs in the most privileged mode, i.e., in the monitor mode.

The LTZVisor uses the VirtIO, not for I/O Virtualization, but for IPC using the RPMsg API, provided by Texas Instruments and OpenAMP. In Figure 20, the general architecture of the TrustZone (TZ)-VirtIO IPC mechanism in the LTZVisor is described, including both RPMsg virtual devices and its VirtIO abstraction, as well as the data and event paths. This IPC mechanism is based on shared memory and, as illustrated, makes use of the non-secure block of memory for the transport layer, i.e., data path. This non-secure memory block is statically allocated at boot-time.

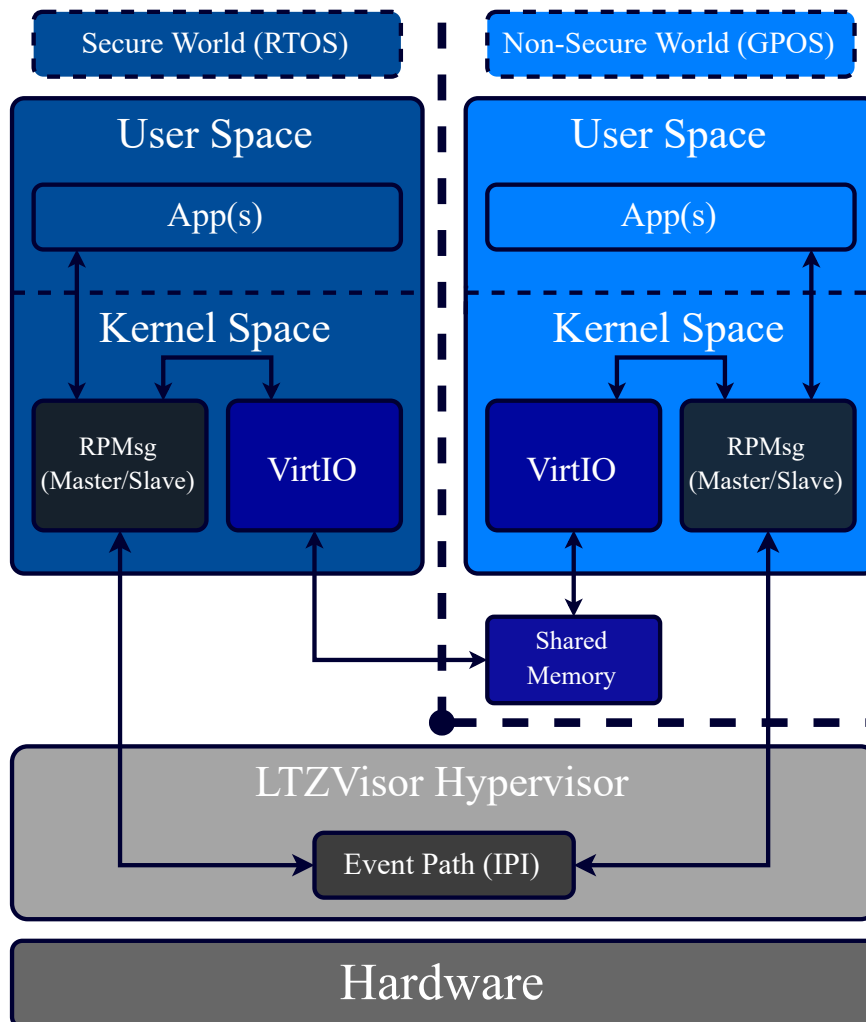


Figure 20: VirtIO mechanism for IPC in LTZVisor architecture

2.2.6 Conclusion

As previously mentioned, the Bao hypervisor currently lacks a mechanism for sharing peripherals between guests. After conducting thorough research on related works, it has been verified that no other hypervisors have a suitable implementation to be implemented in the Bao hypervisor.

In Section 2.2, we individually present each of the hypervisors that have peripheral sharing mechanisms. Additionally, this section highlights an academic work that employs the VirtIO protocol for communication between VMs. Although this approach is not suitable for the Bao hypervisor, it will be tested in an early stage to evaluate VirtIO communication between VMs. The approach used in the KVM hypervisor is not suitable for Bao as it aims to maintain simplicity and minimalism, and KVM requires the back-ends present in QEMU for peripheral sharing. The Bao hypervisor does not intend to rely on external software for its operation. The approach used in the Xen hypervisor is intriguing, as it utilizes a dedicated guest for direct communication with peripherals. Despite its appeal, it has limitations in compatibility, as the Xen ParaVirtualization (PV) drivers were created exclusively for the Xen hypervisor. The approach used in the Xvisor hypervisor is relatively simple and employs the VirtIO protocol, however the back-ends of this protocol are located in the hypervisor and as previously stated, it is not a suitable approach for Bao as it would increase complexity and unnecessary drivers. The approach adopted in the ACRN Hypervisor is the most comparable to the one that will be developed in this thesis, as it uses the VirtIO protocol and employs a dedicated guest for peripheral access and communication with other guests. However, this approach has significant limitations as the guest service is implemented using a Linux OS and is only compatible with communication through split virtqueues, which will be further discussed in the next chapter.

Since existing hypervisors do not present a suitable approach for a static partitioning hypervisor with high performance needs, this dissertation proposes an alternative approach. This approach involves a guest service dedicated solely to communication with other guests through shared memory, while also managing information exchange with peripherals. This approach allows for peripheral sharing while still adhering to the constraints of a static partitioning hypervisor, such as the Bao hypervisor.

Analysis and Design

This chapter offers a comprehensive examination of the VirtIO protocol, followed by an explanation of the VirtIO infrastructure design. It is organized as follows: Section 3.1 delves into various VirtIO topics such as feature bits, virtqueues, and includes a sequential diagram to demonstrate the flow of VirtIO communications and the specific flow for each device, including sending and receiving in the case of the console device and requests in the case of the block device. Section 3.2 presents two suggested VirtIO infrastructure designs: a hypercall based design and a trap-and-emulate based design.

3.1 VirtIO Analysis

In this section, a deep analysis of VirtIO is carried out, specifically, how the device status field works, what the feature bits serve for, how notifications are made, what virtqueues are and what they are for, and how VirtIO transport works. Additionally, the VirtIO devices to be developed are explained.

3.1.1 Device status field

The device status field is a string of bits that the device and driver use to provide a part of device initialization, following a defined order. At initialization time, the driver sets the bit **ACKNOWLEDGE (0×1)** of the device status field to indicate that it knows the device. Then, the driver sets the bit **DRIVER (0×2)** to indicate that it knows how to drive the device. Afterwards, comes the negotiation through the feature bit and, if successful, it sets the bit **FEATURES_OK (0×8)** and the bit **DRIVER_OK (0×4)**. From this moment on, communication can start. The device can set the bit **DEVICE_NEEDS_RESET (0×40)**, and the driver can set the bit **FAILED (0×80)** if any fatal failure occurs.

3.1.2 Feature bits

The feature bits of a device are used to communicate which features it supports and to negotiate with drivers on how they will be utilized. During initialization, as previously discussed, the driver reads

the features bits that the device offers and writes back the features it can handle. If there is agreement between the device and driver, the driver will allocate the virtqueues and notify the device.

3.1.3 Notifications

The driver and device send notifications to signal specific events. There are three types of notifications:

- Configuration change.
- Available buffer.
- Used buffer.

Both the *used buffer* and the *configuration change* notifications are sent by the device, indicating that the device configuration has changed and that there is new data, for the driver to see, in the used virtqueue, respectively. In contrast, the *available buffer* notification is sent by the driver, notifying that there is new data available, in the available virtqueue.

3.1.4 Virtqueues

A Virtqueue is the data transport mechanism in VirtIO, and it is simply a circular array of buffer pointers to establish communication between the driver and the device, through writing and reading, as shown in Figure 21). The driver makes requests to the device by adding an available buffer to the queue, i.e., adding a buffer describing the request and later triggering an event to the device. The device executes the request sent by the driver and, when completed, adds a used buffer to the queue, i.e., marking the buffer as used and later triggering an event to the driver. The device also needs to send the driver the number of bytes it has written to memory, commonly referred to as “used length”.

Each virtqueue can consist of up to three parts:

- **Area descriptor** – used for describing buffers.
- **Driver Area** – extra data supplied by the driver to the device.
- **Device Area** - extra data supplied by the device to the device.

There are two supported formats for virtqueues called:

- Split Virtqueues.
- Packed Virtqueues.

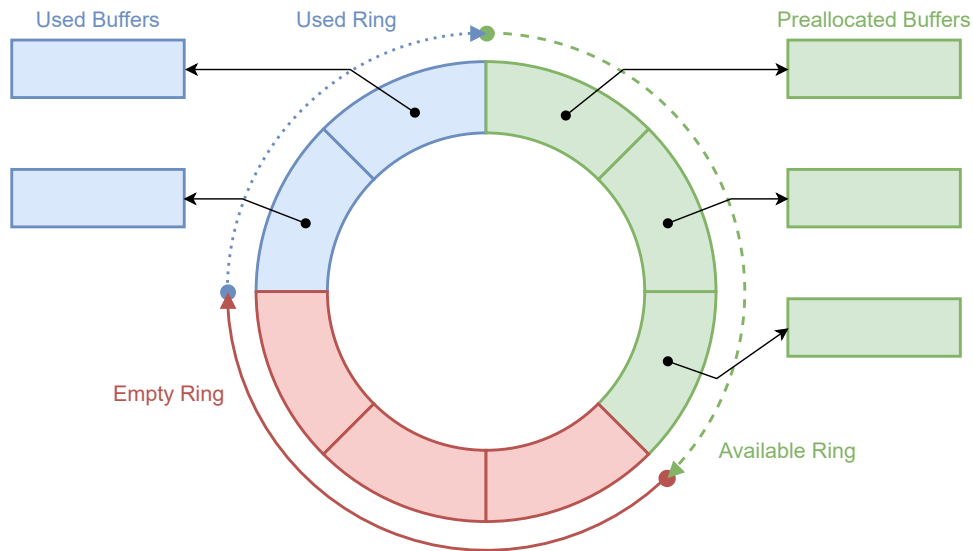


Figure 21: Management of VirtIO virtqueues

3.1.4.1 Split Virtqueues

Split Virtqueue was the only format supported until version 1.0 of the VirtIO standard. It separates the virtqueue into multiple parts, where each part can be read-only or write-only from the perspective of the device. The parts referred previously use different names in split virtqueue, these being:

- **Descriptor Table** is the Descriptor Area.
- **Available Ring** is the Driver Area.
- **Used Ring** is the Device Area.

This format has, as its main characteristic, the simplicity of transporting data between the device and the driver owed to a well-defined and separate organization. The split virtqueue communications can be summarized as:

1. When the driver wants to send a buffer to the device, it fills a slot in the Descriptor Table and writes the index of the descriptor inside the Available Ring.
2. Then the driver notifies the device.
3. When the device completes the operation, it writes the descriptor index on the Used Ring.
4. Finally, the device sends a notification to the driver.

According to the specification [32], the device and driver adhere to the following layout structure to locate elements in virtqueue (see Listing 1).

```

1 struct virtq {
2     // The actual descriptors (16 bytes each)
3     struct virtq_desc desc[ Queue Size ];
4     // A ring of available descriptor heads with free-running index.
5     struct virtq_avail avail;
6     // A ring of used descriptor heads with free-running index.
7     struct virtq_used used;
8 };

```

Listing 1: VirtIO split virtqueue structure layout

As mentioned, inside a virtqueue there is a Descriptor Table, an Available Ring, and a Used Ring. This virtqueue will be allocated by the driver in memory, and the device will have to parse the virtqueue to get the requests and process them. The **Descriptor Table** concerns the buffers that the driver sends to the device, indicating the **Address** and **Length**. Each descriptor also includes a set of flags that provide information about it. For instance, activating the flag **VIRTQ_DESC_F_NEXT (0x1)** indicates that a buffer continues in another buffer descriptor (**next**), and the flag **VIRTQ_DESC_F_WRITE (0x2)** indicates whether the buffer is write-only for the device or read-only. According to the specification [32], the Descriptor Table follows the following layout (see Figure 22).

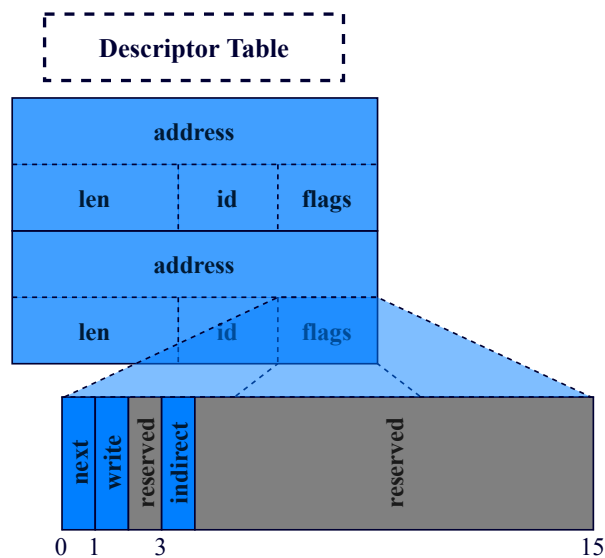


Figure 22: Descriptor table – circular array with constitution of the descriptors

The **Available Ring** is used by the driver, providing buffers for the device, inserting in it the index of the descriptor that the device will consume. It contains two fields that can only be written by the driver and read by the device: **index** and **flags**. The **index** field indicates where the driver will place the next descriptor entry on the ring. If the driver does not want to be notified, the **VIRTQ_AVAIL_F_NO_INTERRUPT (0x1)** flag must be set. Besides the fields described above, there is an array (**ring**) of the same size as

the descriptor where the descriptor index will be stored. According to the specification [32], the Available Ring follows the layout showed in Figure 23.

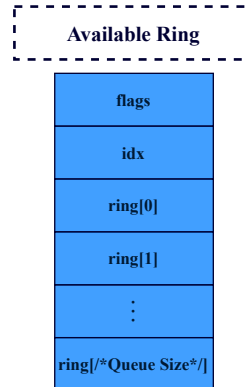


Figure 23: Available ring - buffers stored to be sent to back-end

The **Used Ring** is where the device returns the buffers when they are processed. Contrary to the Available Ring fields, the **flags** and **index** within the Used Ring can only be written by the device and read by the driver, and serve the same purpose but with different directions. The other difference is the name of the flag, now called **VIRTQ_AVAIL_F_NO_NOTIFY (0×1)**. After those two fields, there is an array (**ring**) of used descriptors containing, in the **ID** and **length** fields, the used of the descriptor start index and the buffers size, respectively. According to the specification [32], the Used Ring corresponds to the following layout (see Figure 24).

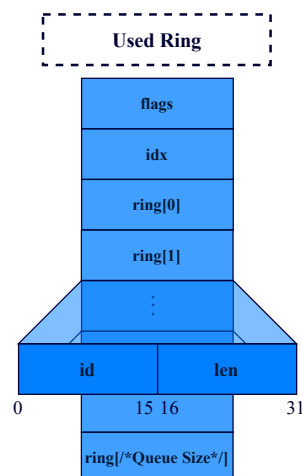


Figure 24: Used ring - buffers to be sent to front-end

3.1.4.2 Packed Virtqueues

As previously mentioned, the split virtqueue format is straightforward to implement and integrate into various systems. However, this format poses challenges in terms of memory usage, particularly in terms of increased memory consumption, due to the need for separate virtqueues for the different fields described in Section 3.1.4.1 [36]. In response to this issue, version 1.1 of the VirtIO specification [32] introduced the packed virtqueue format, which promises a more optimized format by reducing the overhead caused by the split virtqueue format, which can put pressure on CPU cache usage.

The solution involves merging the three rings in a memory that can be written to and read from both the driver and the device. To use this format, `VIRTIO_F_RING_PACKED` must be negotiated on the feature bits in the initialization time.

Each packed virtqueue consists of three parts:

- **Descriptor Ring** is the Descriptor Area.
- **Driver Event Suppression** is the Driver Area.
- **Device Event Suppression** is the Device Area.

The Descriptor Ring, in turn, consists of descriptors, each of which is composed of:

- Buffer ID.
- Element Address.
- Element Length.
- Flags.

Compared to split virtqueue, packed has an approach that makes more sense since the device is able to discard and overwrite the data it has already read from the driver, and the same in reverse. The packed communications can be summarized as:

1. When the driver wants to send a buffer to the device, it writes at least one available descriptor, describing the elements of the buffer, in the Descriptor Ring. Thus, the descriptor is associated with the buffer through the Buffer ID stored in the descriptor.
2. Then the driver notifies the device.
3. When the device finishes processing the buffer, it writes a used device descriptor including the Buffer ID into the Descriptor Ring (overwriting the previously available descriptor driver).
4. Finally, the device sends a used event notification.

According to the VirtIO specification [32], the Used Ring adheres to the following layout (see Figure 25).

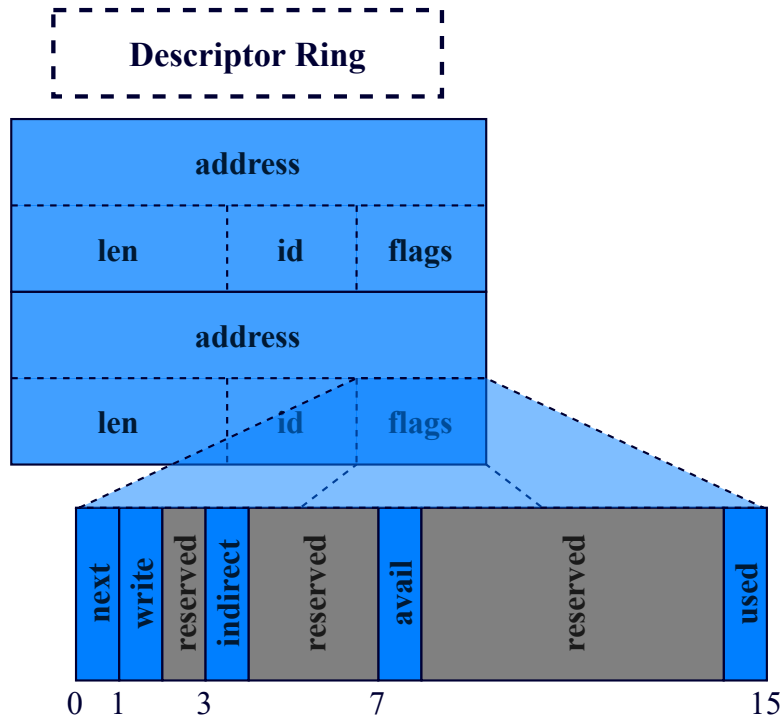


Figure 25: Descriptor ring - circular array with constitution of the descriptors

In this case, the **ID** field is not an index for the device and only has meaning for the driver. Both driver and the device are expected to maintain, internally, a single-bit ring wrap counter. The driver changes the value of the **Driver Ring Wrap Counter** each time it makes the last descriptor available in the ring. The same occurs with the **Device Ring Wrap Counter**, but when the device uses the last descriptor in the ring.

To mark a descriptor:

- as available, the driver makes the **VIRTQ_DESC_F_AVAIL (0×7)** flag the same as its internal **Driver Ring Wrap Counter**, and the **VIRTQ_DESC_F_USED** flag the *inverse*.
- as used, the driver makes the **VIRTQ_DESC_F_USED (0×15)** flag the same as its internal **Device Ring Wrap Counter**, and the **VIRTQ_DESC_F_AVAIL (0×7)** flag the *same*.

Thus, **VIRTQ_DESC_F_AVAIL** and **VIRTQ_DESC_F_USED** bits are different for an available descriptor and equal for a used descriptor.

3.1.5 VirtIO over MMIO

VirtIO provides transport for these types of devices through MMIO. Table 1 presents the general MMIO registers and a brief description of their function.

Register name	Descriptor
MagicValue	Magic value 0x74726976
Version	Device version number 0x2
DeviceID	Virtio Subsystem Device ID
VendorID	Virtio Subsystem Vendor ID
DeviceFeatures	Flags representing features the device supports
DeviceFeaturesSel	Device features word selection
DriverFeatures	Flags representing device features understood and activated by the driver
DriverFeaturesSel	Activated features word selection
QueueSel	Virtual queue index Write to this register to select the virtqueue to operate.
QueueNumMax	Maximum virtual queue size Read from the register to know the max number of virtqueues. This applies to the virtqueue selected by QueueSel.
QueueNum	Virtual queue size Write to this register to notify the device the number of elements on virtqueues.
QueueReady	Virtual queue ready bit Write to this register to notify the device that it can execute requests from this virtqueue.
QueueNotify	Queue notifier Write to this register notifies the device that there are new buffers to process.
InterruptStatus	Interrupt status Read from these register to know what causes the event. The event can be Used Buffer Notification and Configuration Change Notification.
InterruptACK	Interrupt acknowledge This register is used to indicate to the device that the events causing the interrupt were handled.

	Device status
Status	Read from this register to return the current device status flags. To trigger the reset, write zero on this register.
	Virtual queue's Descriptor Area 64 bit long physical address
QueueDescLow/High	Write to these registers to notifies the device about the location of the Descriptor Area of the virtqueue selected by <i>QueueSel</i> .
	Virtual queue's Driver Area 64 bit long physical address
QueueDriverLow/High	Write to these registers to notifies the device about the location of the Driver Area of the virtqueue selected by <i>QueueSel</i> .
	Virtual queue's Device Area 64 bit long physical address
QueueDeviceLow/High	Write to these registers to notifies the device about the location of the Device Area of the virtqueue selected by <i>QueueSel</i> .
SHMSel	Shared memory ID
SHMLenLow/High	Shared memory region 64 bit long length
SHMBaseLow/High	Shared memory region 64 bit long physical address
QueueReset	Virtual queue reset bit
ConfigGeneration	Configuration atomicity value
Config	Configuration space Device-specific configuration space.

Table 1: MMIO back-end register to perform the VirtIO transport with front-end

3.1.6 VirtIO Block

As mentioned before in Section 2.1.6.2, this device is a simple block device. This VirtIO block has at least one virtqueue called “requestq”. The purpose of this virtqueue is always from the point of view of the driver, that is, when the driver needs to access the block device, whether to read or write, it executes a request to the device.

3.1.6.1 Physical Block Device

Examples of real-world block devices are a hard disk, SD Card, or any other device that can store large amounts of data. These devices can store more data than main memory, but accessing the data takes longer compared to registers, cache, or memory. Each block device is divided into “blocks” for storing or reading data, and it is necessary to address each of these blocks. For instance, typically, most device blocks are organized in blocks of 512 bytes, so to read bytes 0-511, sector 0 is loaded, and to read bytes 512-1023, sector 1 is loaded.

3.1.6.2 Device configuration layout

The VirtIO block device has a structure that contains parameters that configure the virtual device. The physical device defines these parameters. This device has multiple parameters, represented below (see Listing 2).

```

1 struct virtio_blk_config {
2     uint64_t capacity;
3     uint32_t size_max;
4     uint32_t seg_max;
5     struct virtio_blk_geometry geometry;
6     uint32_t blk_size;
7     struct virtio_blk_topology topology;
8     uint8_t writeback;
9     uint8_t unused0;
10    uint16_t num_queues;
11    uint32_t max_discard_sectors;
12    uint32_t max_discard_seg;
13    uint32_t discard_sector_alignment;
14    uint32_t max_write_zeroes_sectors;
15    uint32_t max_write_zeroes_seg;
16    uint8_t write_zeroes_may_unmap;
17    uint8_t unused1[3];
18    uint32_t max_secure_erase_sectors;
19    uint32_t max_secure_erase_seg;
20    uint32_t secure_erase_sector_alignment;
21 };

```

Listing 2: VirtIO block device configuration

Most parameters, included in the structure, have a feature bit associated and, as mentioned, require negotiation in the initialization phase. If the feature was not negotiated, the feature is not considered for the work flow of the device. This device has a parameter called **capacity** that represents the size of the physical device in sectors of 512 bytes. This is the only parameter that does not need feature bits negotiation. For example, in the case of the **num_queues** parameter, the **VIRTIO_BLK_F_MQ** feature bits need to be negotiated.

3.1.6.3 Device Initialization

The initialization of the VirtIO console device consists of two phases:

1. Read physical device size from capacity phase.
2. Feature bits negotiation phase.

3.1.6.4 Device Operation

The operation of the VirtIO block device is based on the following process: When a driver wants to perform an action on the block device, it places a request buffer, as shown in Listing 3, in the virtqueue and then sends a notification to the device. The device then receives the notification, processes and executes the request and, when finished, sends the buffer back through the same virtqueue but with the status field updated.

```
1 struct virtio_blk_req {
2     uint32_t type;
3     uint32_t reserved;
4     uint64_t sector;
5     uint8_t data[];
6     uint8_t status;
7 };
```

Listing 3: VirtIO block device request

The type field, present in the VirtIO block device request, represents the type of the request and can be **read** - VIRTIO_BLK_T_IN (0), **write** - VIRTIO_BLK_T_OUT (1), **flush** - VIRTIO_BLK_T_FLUSH (4), **get device ID string command** - VIRTIO_BLK_T_GET_ID (8), **get device lifetime command** - VIRTIO_BLK_T_GET_LIFETIME (10), **discard** - VIRTIO_BLK_T_DISCARD (11), **write zeros** - VIRTIO_BLK_T_WRITE_ZEROES (13) and **secure erase** - VIRTIO_BLK_T_SECURE_ERASE (14). The sector field indicates the offset, in multiples of 512 bytes, where the read or write will occur. For commands that are not read and write, this field is unused and is set to 0. The data field is populated with different data depending on the command type. In a read command, the data is populated with the content of the sector read by the block device and in a write command the data is populated with the content to be written by the device in the desired sector. The status field is written by the device to inform the driver operation's result that was performed. If the operation, from the point of view of the device, was:

- **success**, the device sends, from the status field, the VIRTIO_BLK_S_OK (0) to the driver;
- **error**, the device sends, from the status field, the VIRTIO_BLK_S_IOERR (1) to the driver;
- **unsupported**, the device sends, from the status field, the VIRTIO_BLK_S_UNSUPP (2) to the driver.

3.1.7 VirtIO Console

As mentioned before in Section 2.1.6.3, this device is a simple device for input and output (i.e. UART, etc) having, at least, two virtqueues for communication and two for control. The control virtqueues are used to communicate information between the device and the driver. For data I/O, one or more empty

buffers are placed in the receiveq queue for incoming data and, for outgoing data, they are placed in the transmit queue, from the perspective of the driver.

3.1.7.1 Device configuration layout

The VirtIO console device has a structure that contains parameters that configure the virtual device. These parameters are defined by the physical device, which has four parameters, represented below (see Listing 4).

```
1
2 struct virtio_console_config {
3     uint16_t cols;
4     uint16_t rows;
5     uint32_t max_nr_ports;
6     uint32_t emerg_wr;
7 };
```

Listing 4: VirtIO console device configuration

For each parameter, included in the structure, there is a feature bit associated which must be negotiated in the initialization phase. If the feature was not negotiated, the feature is not considered for the work flow of the device. For example, the parameter **emerg_wr** has an associated feature bit called **VIRTIO_CONSOLE_F_EMERG_WRITE** and if it is negotiated, the driver can use emergency write to output a single character without initializing VirtIO queues.

3.1.7.2 Device Initialization

The initialization of the VirtIO console device consists of two phases:

1. Feature bits negotiation phase;
2. Receive virtqueue filling phase.

3.1.7.3 Device Operation

The operation of the VirtIO console device can be summed in the following situation: when a driver wants to output data, a buffer is placed in the **transmitq**, and when the device receives data from the “external world”, through a terminal for example, the device placed the data into **receiveq**. When the driver receives the data from the device and uses the buffer, it sends a used buffer notification.

3.2 VirtIO Infrastructure

After analysing the VirtIO protocol, the communication and transport mechanisms between the device and driver, and the various devices provided, the design of the VirtIO infrastructure and the design for each device can commence. The initial step consist in creating a structure relationship diagram that defines the VirtIO infrastructure API used by the Service Guest and others guests.

3.2.1 Structural diagram

The suggested VirtIO API is represented in Figure 26 in the gray blocks. The blue blocks represent the instantiation for each device.

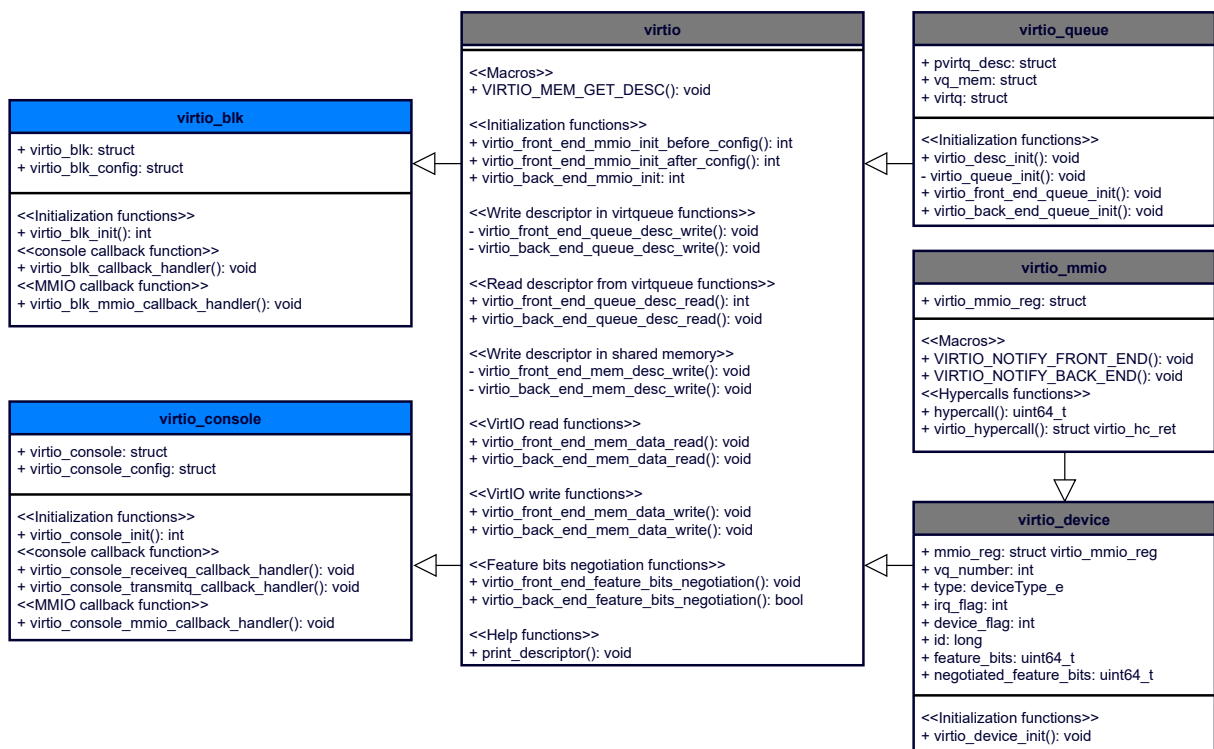


Figure 26: Structure relationship diagram design

As illustrated above, following the topology shown in Figure 15, it is possible to identify the divisions in the API, these being:

- VirtIO Infrastructure - virtio, virtio_queue, and virtio_device;
- VirtIO Transport - virtio_mmio;
- VirtIO Back-end drivers - virtio_blk and virtio_console.

3.2.2 VirtIO Initialization

For the VirtIO protocol to work correctly, the infrastructure has to comply with the initialization of several components. Initialization for the Device is significantly different from the Driver (front-end), as the Device (back-end) needs to initialize MMIO registers and the Driver needs to start negotiations, for example. The Figures 27 and 28 illustrate the initialization flow for driver and device, respectively.

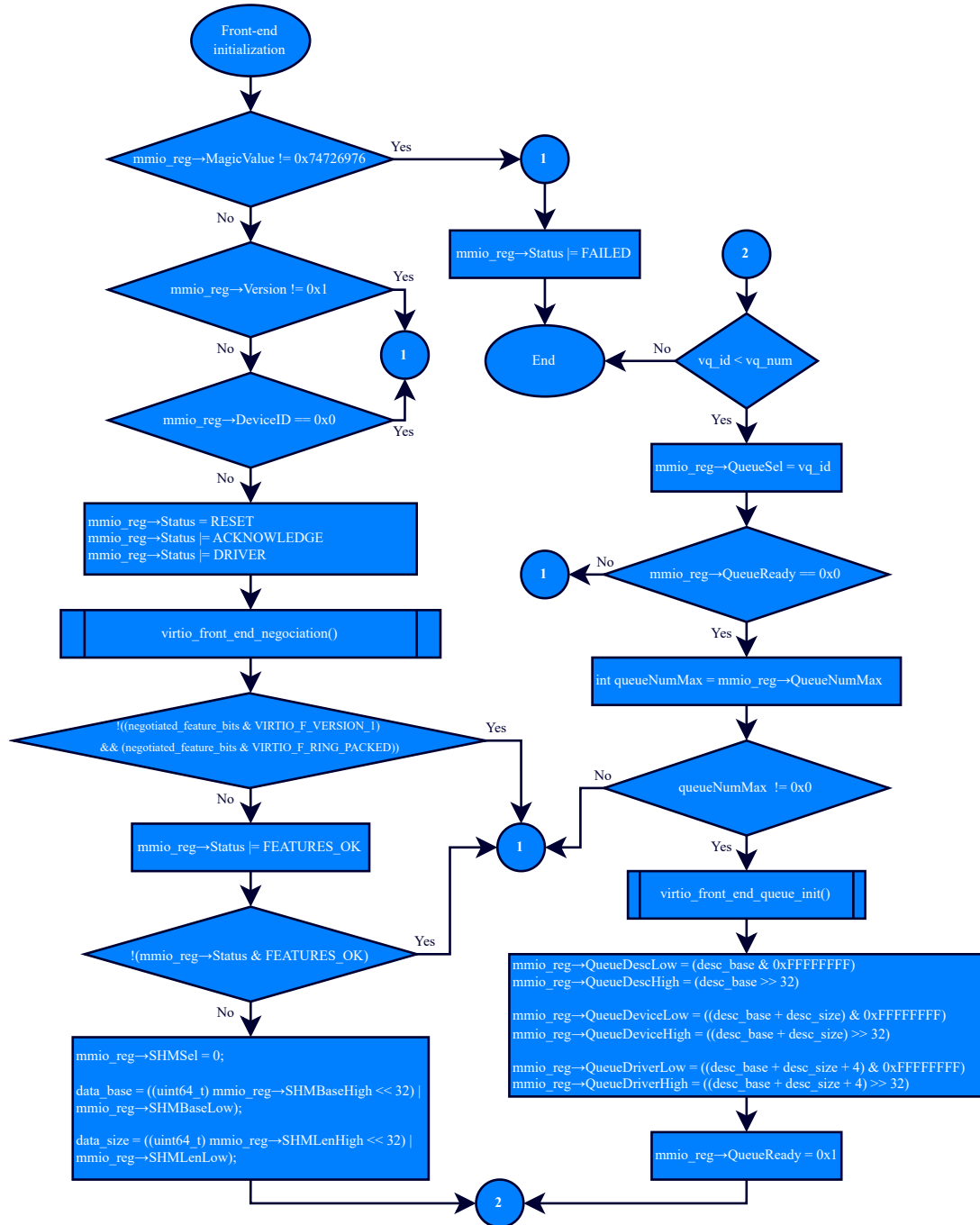


Figure 27: Driver initialization design flowchart

The Driver initialization is much more complex than the Device initialization, because the Drive is responsible for :

1. Verification of the values onto MMIO registers;
2. Attribution of the states;
3. Initialization of the Feature bits negotiation;
4. Acquisition of the addresses of memory that are shared, through the MMIO registers;
5. Initialization of each of the virtqueue;
6. Notification, for each virtqueue, which are the memory addresses;
7. Attribution the failed state onto status MMIO register when something goes wrong.

The Device initialization is straightforward. As illustrated, it performs the initialization of the MMIO registers to the default values and initializes the needed virtqueues.

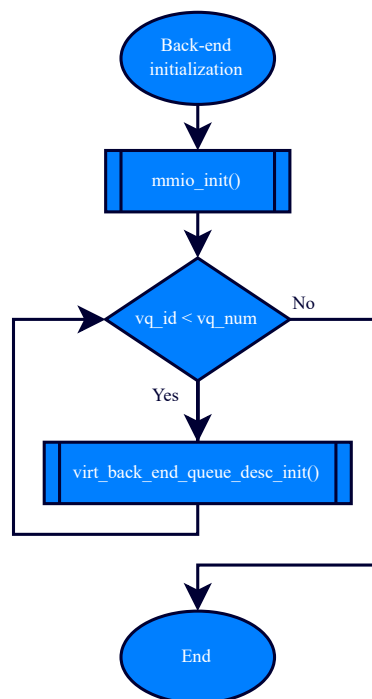


Figure 28: Device initialization design flowchart

3.3 VirtIO under the perspective of the hypervisor

This section presents two mechanisms for communication between the Service Guest and the other guests. In the first mechanism, communications are designed through hypercalls in conjunction with sharing virtqueues using shared memory. In the second mechanism, VirtIO transport was designed using MMIO by adding some functionalities to trap-and-emulate in the hypervisor.

3.3.1 VirtIO mechanism using hypercalls

In the initial stage, a communication mechanism was designed between the Service Guest and two driver guests by placing the virtqueues in shared memory between them, as illustrated in Figure 29. To do this, it was necessary to decide which type of virtqueues would be used in the context of the thesis. Based on the analysis carried out previously, in chapters 3.1.4.1 and 3.1.4.2, it was decided to design the mechanism using packed virtqueues, due to their optimized format, that places less pressure on CPU cache usage.

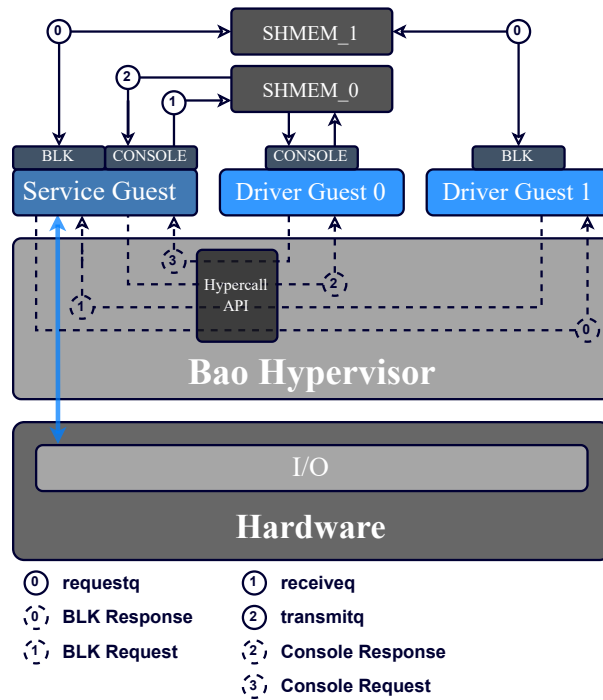


Figure 29: VirtIO hypercall mechanism design

In summary, the communication mechanism consists of creating interrupts, configured in the hypervisor, associated with each hypercall. Consequently, when the communication is necessary, the guest stores in the shared memory, inside the virtqueue, the buffer that needs to be sent to the Service Guest. Later, the guest executes the request hypercall, which calls the hypervisor, that identifies the type of hypercall and communicates with the Service Guest through the interrupt that corresponds to the request

hypercall on the Service Guest side. Subsequently, the Service Guest reads the buffer contained in the virtqueue from shared memory and executes the request stored in the buffer. When the request is completed by the Service Guest, it sends a response hypercall, informing the guest that the request has been executed. It is worth observing the difference between each of the devices illustrated in Figure 29, this is due to the fact that the block has a bidirectional virtqueue while the console requires two virtqueues to perform communications, as was explained in Sections 3.1.7 and 3.1.6. This design does not negotiate feature bits in the initialization phase, as the transport mechanism is not available.

3.3.2 VirtIO mechanism using Trap-and-Emulate

Once the basic VirtIO infrastructure had been implemented through shared memory communication and notifications with hypercalls, an adaptation of the hypercall design began to be based on the trap-and-emulate technology. This technology already exists in the hypervisor, where some functions can be used or adapted for the VirtIO implementation through trap-and-emulate. Both at initialization time and during data exchange between the back-end and the front-end, it is necessary to write and read from the MMIO registers. However, these registers are located in the memory of the back-end. To allow the front-end to perform these write and read operations, a mechanism must be created. That mechanism consist in: when the front-end attempts to access a memory location it does not have access to, a trap is generated for the hypervisor to inform the Service Guest, providing information about the type of operation and which register needs to be written to. The proposed design is explained below and illustrated in Figures 30 and 31.

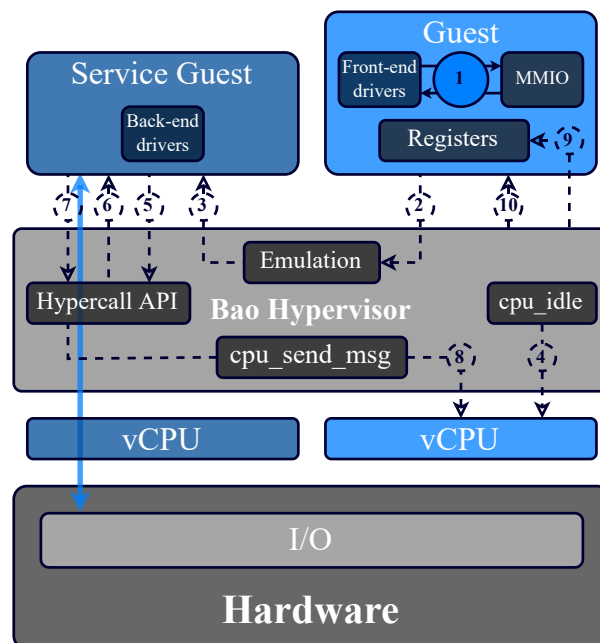


Figure 30: VirtIO MMIO mechanism design

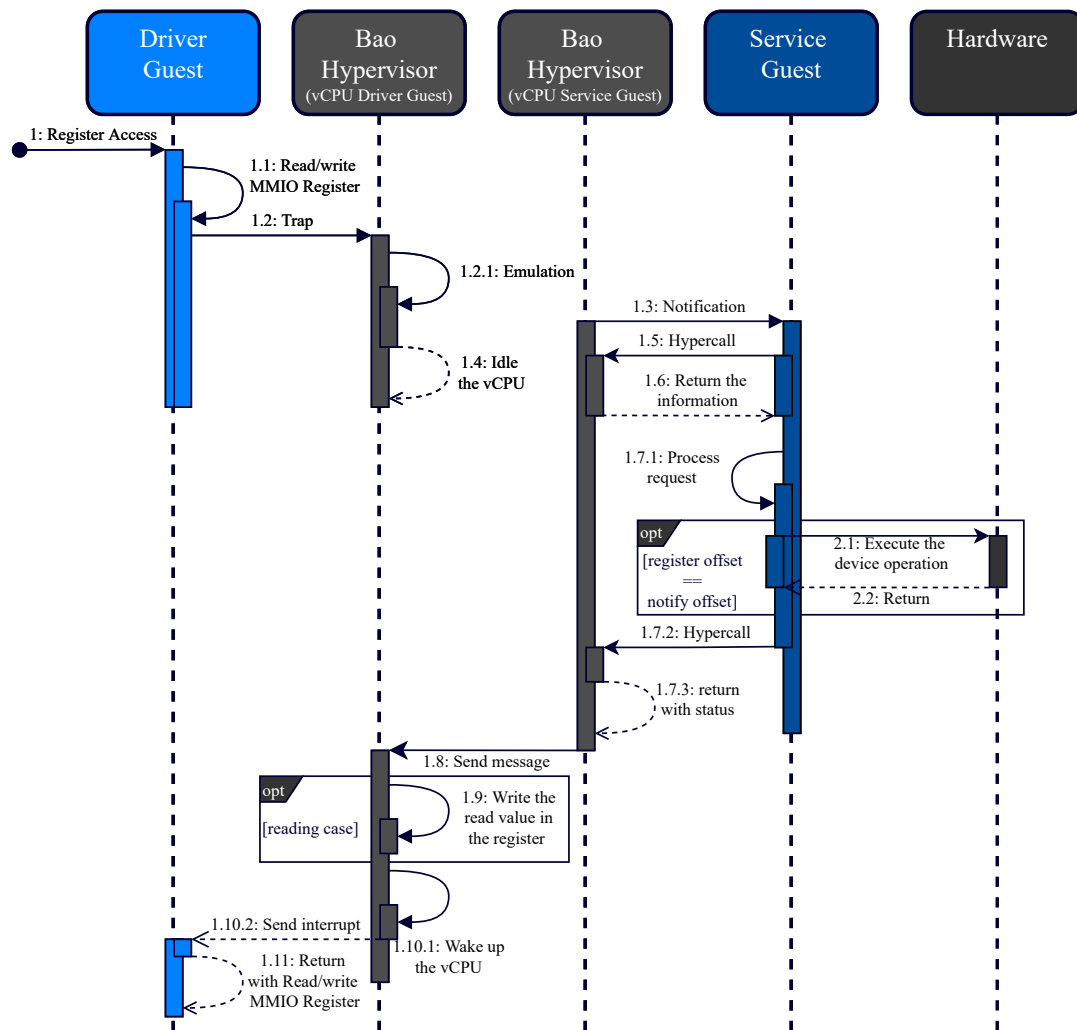


Figure 31: VirtIO MMIO mechanism design – sequence diagram

In summary, the mechanism performs the following steps, as illustrated in Figures 30 and 31:

1. The Driver Guest tries to access a MMIO register that is not part of its memory;
2. A trap is triggered, giving control to the hypervisor that proceeds to the emulation;
3. The handler associated with accessing the MMIO register stores the register offset, operation, and desired value (if the operation is write) in the device structure. It then sends a notification to the Service Guest;
4. After sending the notification, the CPU corresponding to the Driver Guest goes into idle, waiting for the register value to be sent back;
5. The Service Guest receives the notification and "asks" the hypervisor what happened, through a hypercall;

6. The hypervisor receives the hypercall, and returns the register offset, the operation and the value, in the case of write operation, that are in the device struct stored in step 3;
7. The Service Guest receives and processes the data. At the end, it sends another hypercall to inform the status of the operation and the value of the register in the case of a read operation;
8. The hypervisor sends a message to the CPU corresponding to the driver through the *cpu_send_msg* function;
9. The CPU corresponding to the driver receives the message and writes the received values into registers;
10. Finally, the hypervisor injects the device interrupt, waking the CPU corresponding to the driver.

3.4 VirtIO Block and Console

The Figure 32 shows the suggested design for the case of using a device block. As illustrated, the Service Guest provides an API for the SD card in which, depending on the type of request, the read or write function is called. These functions communicate directly via pass-through with the SD Card hardware. Figure 32 is abstracting the implementation of the trap-and-emulate mechanism to simplify the visualization and the complexity of the diagram.

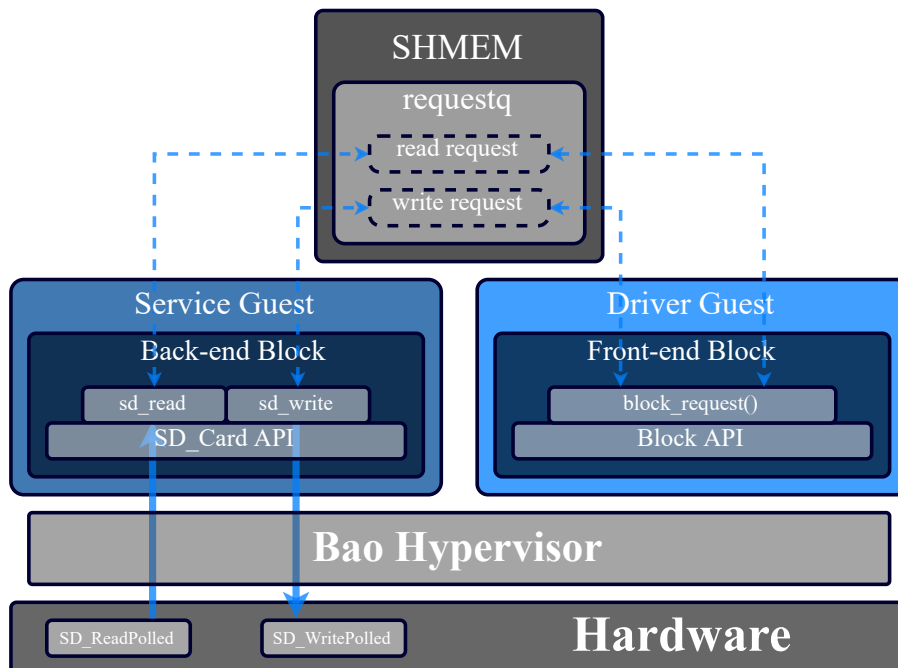


Figure 32: VirtIO block device design

Figure 33 illustrates the suggested design for a device console. As illustrated, the Service Guest offers a UART API where, when one or more buffers are placed in the transmitq virtqueue, the characters in the buffer are placed at the output of the UART using the *uart_puts()* function. When the UART receives characters, the interrupt for receiving is handled and a buffer is placed in the receiveq virtqueue for the driver to process.

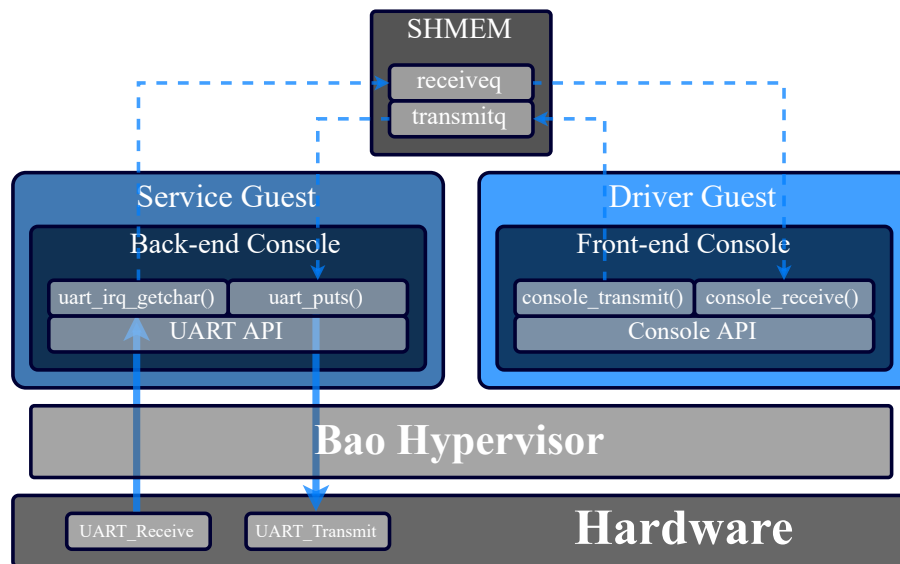


Figure 33: VirtIO console device design

Implementation

As discussed and designed in Chapter 3, the implementation of the VirtIO infrastructure was carried out to virtualize hardware between different guests and the back-ends of the devices presented in that chapter. To accomplish this, adjustments were made to the VirtIO infrastructure to function alongside the Bao hypervisor architecture using hypercall mechanisms. In the hypervisor, additional features were added to the trap-and-emulate mechanisms. In implementing the back-ends for the console device, the `virtio_console` driver was created, which utilizes the existing driver in Bao (`zynq_uart`) and for the block device, the `virtio_block` driver was created, which uses the driver provided in the Xilinx repository (`xsdps`).

4.1 VirtIO Infrastructure

Several structures and functions that define the VirtIO infrastructure API were implemented and distributed across four files, as outlined in Section 3.2. Some of these features, such as initialization, MMIO implementation, and trap-and-emulate mechanisms, will be described in more detail to better understand the functioning of the system.

4.1.1 VirtIO Initialization

The VirtIO initialization process is different for front-end and back-end drivers, as previously discussed in Section 3.2.2. Although the current work focuses on the implementation of back-end drivers, it was also necessary to implement the front-end drivers to test the infrastructure in early stages and to more accurately evaluate the proper functioning of the design, which was done through a test with QEMU (presented in the next chapter). In contrast to the initialization of the back-end driver, the front-end driver initialization flow is quite different, as described in Section 3.2.2. A function was implemented, following the flowchart of the front-end driver initialization (see Figure 27, which can be found in Listing 7. In the first phase, the front-end initialization function checks if the associated back-end driver exists and has already been initialized, verifies the Magic Value, the version as non-legacy and the DeviceID. Then it updates the Status

register with the information that it has recognized the back-end driver, and it knows how to work with it, proceeding to negotiate the feature bits.

```

1  int virtio_front_end_mmio_init( struct virtio_device *device,
2      struct pvirtq_desc **vqs_desc_ptr, int *vqs_size_ptr,
3      struct virtq **vqs_ptr, struct vq_mem *mem, const int vq_num) {
4      if ((mmio_reg->MagicValue != VIRTIO_MAGIC_VALUE) ||
5          (mmio_reg->Version != VIRTIO_VERSION_NO_LEGACY) ||
6          (mmio_reg->DeviceID == 0))
7          mmio_reg->Status |= FAILED; return -1;
8      mmio_reg->Status = RESET;
9      mmio_reg->Status |= ACKNOWLEDGE;
10     mmio_reg->Status |= DRIVER;
11     virtio_front_end_feature_bits_negotiation(device);
12     if(!((negotiated_feature_bits & VIRTIO_F_VERSION_1) &&
13         (negotiated_feature_bits & VIRTIO_F_RING_PACKED)))
14         return -1;
15     mmio_reg->Status |= FEATURES_OK;
16     if(!(mmio_reg->Status & FEATURES_OK))
17         mmio_reg->Status |= FAILED; return -1;
18     for(int vq_id = 0; vq_id < vq_num; vq_id++) {
19         mmio_reg->QueueSel = vq_id;
20         if(mmio_reg->QueueReady == 0x0) {
21             if(mmio_reg->QueueNumMax != 0x0) {
22                 virtio_front_end_queue_init(vqs_ptr[vq_id],
23                     vqs_desc_ptr[vq_id], NULL, vqs_size_ptr[vq_id],
24                     &temp_mem, vq_id);
25                 mmio_reg->QueueReady = 0x1;
26             } else
27                 mmio_reg->Status |= FAILED; return -1;
28         } else
29             mmio_reg->Status |= FAILED; return -1;
30     } return 0;
31 }

```

Listing 5: VirtIO front-end initialization

Because the infrastructure was designed and implemented using the packed virtqueues approach, the negotiated feature bits must have the **VIRTIO_F_RING_PACKED** bit set in order for both drivers to communicate using the same protocol. Once the negotiation is complete, the front-end driver writes to the status register to indicate that it has accepted the feature bits (if they are applicable), and then reads the same register to confirm that the write was successful. If this occurs, the back-end driver has accepted the negotiations. After the negotiations are finished, the front-end driver can initialize the virtqueues to enable communication with the device. To do this, the front-end initializes each of the virtqueues that the device incorporates individually: one virtqueue in the case of the block device and four virtqueues in the

case of the console. The initialization process for each virtqueue involves selecting the virtqueue by writing to the **QueueSel** register, checking if the virtqueue has not been initialized by reading the **QueueReady** register, checking if the virtqueue is available by reading the **QueueNumMax** register, initializing the memory for the virtqueue, and in the final phase of initialization, setting the **QueueReady** register so that the back-end driver is informed that the virtqueue is ready to communicate. Finally, the back-end driver also performs an initialization, which includes initializing registers with default values and initializing virtqueues individually.

```
1 int virtio_back_end_mmio_init( struct virtio_device *device,
2     struct pvirtq_desc **vqs_desc_ptr, int *vqs_size_ptr,
3     struct virtq **vqs_ptr, struct vq_mem *mem, const int vq_num,
4     int queue_num_max, deviceType_e type) {
5     mmio_reg->DeviceFeaturesSel = 0;
6     mmio_reg->DriverFeatures    = 0;
7     mmio_reg->DriverFeaturesSel = 0;
8     mmio_reg->QueueSel          = 0;
9     mmio_reg->QueueNum          = 0;
10    mmio_reg->QueueNotify        = 0;
11    mmio_reg->InterruptACK       = 0;
12    mmio_reg->QueueDescLow       = 0;
13    mmio_reg->QueueDescHigh      = 0;
14    mmio_reg->MagicValue         = 0x74726976;
15    mmio_reg->Version             = 0x2;
16    mmio_reg->DeviceID           = type;
17    mmio_reg->VendorID           = 0;
18    mmio_reg->DeviceFeatures     = 0;
19    mmio_reg->QueueNumMax        = queue_num_max;
20    mmio_reg->InterruptStatus    = 0;
21    mmio_reg->SHMLenLow          = mem->data_size;
22    mmio_reg->SHMLenHigh         = mem->data_size >> 32;
23    mmio_reg->SHMBaseLow         = mem->data_base;
24    mmio_reg->SHMBaseHigh        = mem->data_base >> 32;
25    mmio_reg->ConfigGeneration   = 0;
26    mmio_reg->QueueReady         = 0x0;
27    mmio_reg->Status             = 0;
28    for(int vq_id = 0; vq_id < vq_num; vq_id++)
29        virtio_back_end_queue_init( vqs_ptr[vq_id], vqs_desc_ptr[vq_id],
30            NULL, vqs_size_ptr[vq_id], &temp_mem, queue_num_max, vq_id);
31    return 0;
32 }
```

Listing 6: VirtIO back-end initialization

4.1.1.1 Feature bits negotiation

When initializing the front-end, it is necessary to conduct feature bit negotiations. As the feature bit registers for both the device and the driver are greater than 32 bits, and memory access of MMIO is limited to 32-bits at a time, multiple accesses to the registers will be required, as shown in Listing 7. During each access, the front-end compares the feature bits it accepts with those provided by the back-end, and the result of the comparison is stored in the **DriverFeatures** register.

```

1 void virtio_front_end_feature_bits_negotiation(struct virtio_device
  ↳ *device) {
2     uint64_t comparison = 0;
3     for(int i = 0; i < MMIO_FEATURE_SEL_SIZE; i++) {
4         device->mmio_reg->DeviceFeaturesSel = i;
5         device->mmio_reg->DriverFeaturesSel = i;
6         comparison = device->mmio_reg->DeviceFeatures &
  ↳ (device->feature_bits >> (i * 32));
7         device->mmio_reg->DriverFeatures = comparison;
8         device->negotiated_feature_bits |= (comparison << (i * 32));
9     }
10 }
```

Listing 7: VirtIO front-end feature bits negotiation

On the back-end side, it is important to confirm that the feature bits sent by the front-end are compatible with the feature bits that the back-end is able to accept. The select register is used to shift the feature bits register in 32-bit increments.

```

1 bool virtio_back_end_feature_bits_negotiation(struct virtio_device
  ↳ *device) {
2     uint64_t aux = device->mmio_reg->DriverFeatures;
3     if( (device->feature_bits | (aux << (32 *
  ↳ device->mmio_reg->DriverFeaturesSel))) == device->feature_bits) {
4         device->negotiated_feature_bits |= device->feature_bits & (aux <<
  ↳ (32 * device->mmio_reg->DriverFeaturesSel));
5         return true;
6     }
7     else
8         return false;
9 }
```

Listing 8: VirtIO back-end feature bits negotiation

Once VirtIO is initialized, the infrastructure is ready to begin the execution phase and facilitate communication between the front-end and back-end. To enable this communication, functions were developed in line with the design outlined in Chapter 3, which allow for writes to and reads from virtqueues.

4.1.2 VirtIO Write

The data writing operation is facilitated by the functions `virtio_""_mem_data_write`, are present in Listing 9. These functions are implemented as `virtio_back_end_mem_data_write` and `virtio_front_end_mem_data_write` for back-ends and front-ends respectively. It is important to note that the back-end write function requires an additional parameter, **struct pvirtq_desc* desc**, to indicate in which descriptor the data should be written, whereas the front-end function creates a new descriptor whenever data needs to be written to the virtqueues. In the function `virtio_""_mem_data_write`, all necessary variables such as the length of the data and the address in shared memory are created. These include the descriptor if it is the function for the front-end. Then the function `virtio_""_queue_desc_write` is called to write the descriptor and enable VirtIO to carry out communication.

```
1 void virtio_""_mem_data_write(char* data, struct virtq *vq, struct
  ↳ virtio_device *device, "struct pvirtq_desc* desc") {
2     struct pvirtq_desc desc;
3     int len = strlen(data);
4     char* address = NULL;
5     uint16_t flags = 0; // For the case of front_end function
6     if(vq->mem.last_data_addr >= vq->mem.data_base + vq->mem.data_size)
7         vq->mem.last_data_addr = vq->mem.data_base;
8     address = vq->mem.last_data_addr;
9     strcpy(address, data);
10    desc->addr = (long)address; // For the case of back_end function
11    virtio_desc_init(&desc, (long)address, len, 0, flags); // For the
  ↳ case of front_end function
12    virtio_""_queue_desc_write(vq, device, &desc);
13    vq->mem.last_data_addr += len;
14 }
```

Listing 9: VirtIO front-end/back-end write data

The implementation of the `virtio_""_queue_desc_write` functions, present in Listing 10, was based on the example provided in Section 2.8.21.3 of the VirtIO specification. In summary, the function checks if the descriptor flag parameter contains the bit **VIRTQ_DESC_F_WRITE** set. If confirmed, it updates the descriptor data in the descriptor array that constitutes the virtqueue. It processes the flags, calls the function `virtio_""_mem_desc_write` to increment the value of the virtqueue index for the next iteration and applies a mask to implement a ring virtqueue. At the end, if it is the front-end, it notifies the back-end through the `VIRTIO_NOTIFY_BACK_END` macro, present in Listing 11. If it is the back-end, it updates the

InterruptStatus register of MMIO to inform that the buffer was used for the active virtqueue, and sends a notification to the front-end via the `VIRTIO_NOTIFY_FRONT_END` macro, present in Listing 11.

```

1 void virtio_""_queue_desc_write(struct virtq *vq, const struct
  ↳ virtio_device *device, struct pvirtq_desc *desc){
2     // For the case of back_end function
3     if(desc->flags & VIRTQ_DESC_F_WRITE == 0)
4         return;
5     short avail = 0;
6     short used = 0;
7     uint16_t f = 0;
8     // "" -> "used" for back_end function and "avail" for front_end function
9     vq->desc[vq->next_""].addr = desc->addr;
10    vq->desc[vq->next_""].len = desc->len;
11    avail = vq->""_wrap_count ? VIRTQ_DESC_F_AVAIL : 0;
12    used = !vq->""_wrap_count ? VIRTQ_DESC_F_USED : 0;
13    desc->flags &= ~VIRTQ_DESC_F_AVAIL;
14    desc->flags &= ~VIRTQ_DESC_F_USED;
15    f = desc->flags | avail | used;
16    vq->desc[vq->next_""].flags = f;
17    vq->desc[vq->next_""].id = vq->next_"";
18    virtio_front_end_mem_desc_write(vq);    // For the case of front_end
  ↳ function
19    virtio_back_end_mem_desc_write(vq);    // For the case of back_end
  ↳ function
20    vq->next_""++;
21    if (vq->next_"" >= vq->size) {
22        vq->next_"" = 0;
23        vq->""_wrap_count ^= 1;
24    }
25    VIRTIO_NOTIFY_BACK_END(vq->id);    // For the case of front_end
  ↳ function
26    VIRTIO_NOTIFY_FRONT_END(device->id);    // For the case of back_end
  ↳ function
27 }
```

Listing 10: VirtIO front-end/back-end write descriptor on virtqueue

```

1 #define VIRTIO_NOTIFY_FRONT_END(dev_id) virtio_hypercall(dev_id, 0,
  ↳ INTERRUPT_OP, 0)
2 #define VIRTIO_NOTIFY_BACK_END(vq_id) device->mmio_reg->QueueNotify =
  ↳ vq_id
```

Listing 11: VirtIO front-end/back-end notify macros

The functions `virtio_""_mem_desc_write`, present in Listing 12, are responsible for copying descriptors to the shared memory using the `memcpy` function.

```
1 static void virtio_""_mem_desc_write(struct virtq *vq) {
2     int offset = 0;
3     struct pvirtq_desc *base = NULL;
4     struct pvirtq_desc *descriptor = NULL;
5     // "" -> "used" for back_end function and "avail" for front_end function
6     offset = vq->next_"" * VIRTQ_DESC_ALIGN_SIZE;
7     descriptor = &vq->desc[vq->next_""];
8     char *aux = vq->mem.desc_base + offset;
9     base = (struct pvirtq_desc *) aux;
10    memcpy(base, descriptor, sizeof(struct pvirtq_desc));
11 }
```

Listing 12: VirtIO front-end/back-end write descriptor in the shared memory

4.1.3 VirtIO Read

For the read operation, the `virtio_""_queue_desc_read` functions have been developed, as shown in Listings 14 and 15, depending on whether the guest contains front-end or back-end drivers. The `virtio_front_end_queue_desc_read` function returns true if it is successful. These functions were based on the example provided in Section 2.8.22 of the VirtIO specification. In the case of the front-end function, it takes a descriptor from shared memory using the `VIRTIO_MEM_GET_DESC` macro, as shown in Listing 11, which returns the address of the descriptor to be read.

```
1 #define VIRTIO_MEM_GET_DESC(base, offset) ((struct pvirtq_desc*) (((char
↪ *) base) + offset * VIRTQ_DESC_ALIGN_SIZE))
```

Listing 13: VirtIO get descriptor macro

The descriptor is then loaded into the front-end virtqueue, followed by the `virtio_front_end_mem_data_read` function being called. In the case of the `virtio_back_end_queue_desc_read` function, the descriptor flags are loaded, processed and written in the descriptor. The function `virtio_back_end_mem_data_read` is called to read the data stored in the shared memory and, after the data is processed, the descriptor is written again through the function `virtio_front_end_mem_desc_write` to inform the front-end that the descriptor has been used.

```

1  int virtio_front_end_queue_desc_read(struct virtq *vq, struct
   ↪  virtio_device *device) {
2      desc = VIRTIO_MEM_GET_DESC(vq->mem.desc_base, vq->next_used);
3      if(desc->flags & VIRTQ_DESC_F_WRITE == 0)    return 0;
4      flags = desc->flags;
5      avail = flags & VIRTQ_DESC_F_AVAIL;
6      used = flags & VIRTQ_DESC_F_USED;
7      if (avail != vq->used_wrap_count || used != vq->used_wrap_count)
   ↪  return 0;
8      vq->desc[vq->next_used] = *desc;
9      virtio_front_end_mem_data_read(vq, device->data);
10     vq->next_used++;
11     if (vq->next_used >= vq->size) {
12         vq->next_used -= vq->size;
13         vq->used_wrap_count ^= 1;
14     } return 1;
15 }

```

Listing 14: VirtIO front-end read descriptor from a virtqueue

```

1  void virtio_back_end_queue_desc_read(struct virtq *vq, struct
   ↪  virtio_device *device) {
2      uint16_t flags = vq->desc[vq->next_avail].flags;
3      if(flags & VIRTQ_DESC_F_WRITE) return;
4      flags &= ~VIRTQ_DESC_F_AVAIL;
5      flags &= ~VIRTQ_DESC_F_USED;
6      avail = vq->avail_wrap_count ? VIRTQ_DESC_F_AVAIL : 0;
7      used = vq->avail_wrap_count ? VIRTQ_DESC_F_USED : 0;
8      f = flags | avail | used;
9      vq->desc[vq->next_avail].flags = f;
10     virtio_back_end_mem_data_read(vq, device->data);
11     virtio_front_end_mem_desc_write(vq);
12     vq->next_avail++;
13     if (vq->next_avail >= vq->size) {
14         vq->next_avail -= vq->size;
15         vq->avail_wrap_count ^= 1;
16     } }

```

Listing 15: VirtIO back-end read descriptor from a virtqueue

4.1.4 VirtIO over MMIO

As outlined in section 3.1.5, and as shown in Table 1, a struct that groups all registers of MMIO was created. Since this structure needs to include the config structure of each device and these configs have different sizes, the "aligned" attribute was added to ensure that this struct has enough space to include any of the configuration structs and the "packed" attribute was added to save space, i.e., the variables in memory will be stored consecutively.

```
1 struct virtio_mmio_reg {
2     uint32_t MagicValue;           // Offset 0x000
3     uint32_t Version;              // Offset 0x004
4     uint32_t DeviceID;             // Offset 0x008
5     uint32_t VendorID;             // Offset 0x00c
6     uint32_t DeviceFeatures;       // Offset 0x010
7     uint32_t DeviceFeaturesSel;    // Offset 0x014
8     uint32_t DriverFeatures;       // Offset 0x020
9     uint32_t DriverFeaturesSel;    // Offset 0x024
10    uint32_t QueueSel;             // Offset 0x030
11    uint32_t QueueNumMax;          // Offset 0x034
12    uint32_t QueueNum;            // Offset 0x038
13    uint32_t QueueReady;          // Offset 0x044
14    uint32_t QueueNotify;         // Offset 0x050
15    uint32_t InterruptStatus;      // Offset 0x060
16    uint32_t InterruptACK;         // Offset 0x064
17    uint32_t Status;              // Offset 0x070
18    uint32_t QueueDescLow;         // Offset 0x080
19    uint32_t QueueDescHigh;        // Offset 0x084
20    uint32_t QueueDriverLow;       // Offset 0x090
21    uint32_t QueueDriverHigh;      // Offset 0x094
22    uint32_t QueueDeviceLow;       // Offset 0x0a0
23    uint32_t QueueDeviceHigh;      // Offset 0x0a4
24    uint32_t SHMSel;              // Offset 0x0ac
25    uint32_t SHMLenLow;            // Offset 0x0b0
26    uint32_t SHMLenHigh;           // Offset 0x0b4
27    uint32_t SHMBaseLow;           // Offset 0x0b8
28    uint32_t SHMBaseHigh;          // Offset 0x0bc
29    uint32_t QueueReset;           // Offset 0x0c0
30    uint32_t ConfigGeneration;     // Offset 0x0fc
31    uint32_t Config;               // Offset 0x100
32 } __attribute__((__packed__, aligned(0x1000)));
```

Listing 16: MMIO device register layout

4.1.5 Trap-and-Emulate over Bao

Following the design elaborated in Section 3.3.2, a new trap-and-emulate mechanism was developed in the hypervisor based on existing mechanisms. Once the mechanism is initialized in the hypervisor it changes to the execution phase.

4.1.5.1 Initialization

Firstly, in the initialization phase, the function *virtio_linkage_init* is called which, as represented in Listing 17, makes the connection between the back-end and the front-end and initializes the list of devices.

```

1 void virtio_linkage_init() {
2     list_init(&virtiodevicelist);
3     list_init(&poolingdevicelist);
4     for (int i = 0; i < vm_config_ptr->virtiodevicelist_size; i++) {
5         struct virtiodeviceparams *node = objcache_alloc(&par_cache);
6         vm_config_ptr->virtiodevicelist[i].backend_id = -1;
7         vm_config_ptr->virtiodevicelist[i].frontend_id = -1;
8         node->id = i;
9         list_push(&virtiodevicelist, (node_t*)node);
10    }
11    for (int vm_id = 0; vm_id < vm_config_ptr->vmlist_size; vm_id++) {
12        struct vm_config *vm_config = &vm_config_ptr->vmlist[vm_id];
13        for (int j = 0; j < vm_config->platform.virtiodevices_num; j++) {
14            struct virtiodevice *dev=&vm_config->platform.virtiodevices[
↪ j ];
15            if(dev->is_back_end) {
16                if(vm_config_ptr->virtiodevicelist[ dev->device_id
↪ ].backend_id == -1)
17                    vm_config_ptr->virtiodevicelist[ dev->device_id
↪ ].backend_id = vm_id;
18            }
19            else {
20                if(vm_config_ptr->virtiodevicelist[ dev->device_id
↪ ].frontend_id == -1)
21                    vm_config_ptr->virtiodevicelist[ dev->device_id
↪ ].frontend_id = vm_id;
22            }
23        }
24    }
25 }

```

Listing 17: VirtIO linkage initialization over Bao

To accomplish this, the function first creates a list of devices which is made up of the struct represented in Listing 18, initializing the back-ends and front-ends to -1 and placing the IDs in a list so that they can be accessed by the hypercall. Once the lists are initialized, the function establishes the connection between the back-ends and the front-ends for each device, replacing the previously loaded values of -1 with the ID of the VM.

```
1 struct virtiodeviceparams {
2     node_t node;
3     uint64_t id;
4     unsigned long reg_off;
5     unsigned long access_width;
6     unsigned long op;
7     unsigned long value;
8     unsigned int cpu_id;
9     unsigned int sg_cpu_id;
10    unsigned long reg;
11 };
```

Listing 18: VirtIO device parameters struct over Bao

The struct represented in Listing 18 contains the parameters used in the device emulation, including the device ID, information about accesses to the device, and to which CPU the device is assigned. As this struct is an element of a linked list, it must also include an element that stores the link to the next element, commonly referred to as a "node". The following elements are used to store the information when a request to the Service Guest is emulated: *reg_off* is used to indicate which register offset is being accessed, *access_width* is used to indicate the width of the access (8, 16, or 32 bits), *op* is the operation that must be performed, which can be a **Read** or a **Write**. In case the operation is a **Write**, the element *value* contains the value that should be written at the offset of the register.

Next, the hypervisor starts the VMs through its own function called *vm_init*, which upon execution, initializes memory regions, devices, IPC, and VirtIO. The initialization of VirtIO in the VM is called *vm_init_virtio* and is represented in Listing 19.

The *vm_init_virtio* function first checks if the VM has devices and, if so, calls the function *virtio_ipc_init* to initialize the shared memory, as represented in Listing 21. It then saves the ID number and information of the devices associated with the VM that is being started.

The function then searches for front-end devices, creates a struct called **emu** where the information on the found devices is stored, so that when it is necessary to emulate a device, the information to handle it is available. Since the device is only emulated on the front-ends, this struct is only created in case the device is a front-end. As shown in the Listing 19, it is necessary to verify, in the hypervisor configuration mentioned later in Chapter 5, that the device does not refer to the back-end. In the end, the function *virtio_serviceguest_cpu_init* is called to initialize the connection between the front-end and the back-end.

```

1 static void vm_init_virtio( struct vm* vm,
2                             const struct vm_config* config) {
3     if (config->platform.virtiodevices_num > 0) {
4         virtio_ipc_init(vm, config);
5         vm->virtiodevices_num = config->platform.virtiodevices_num;
6         vm->virtiodevices = config->platform.virtiodevices;
7         vm->pooling = config->platform.pooling;
8         for (int i = 0; i < config->platform.virtiodevices_num; i++) {
9             struct virtio_device *virtio_dev;
10            virtio_dev = &config->platform.virtiodevices[i];
11            if(!virtio_dev->is_back_end) {
12                size_t dev_size
13                dev_size = ALIGN( sizeof(struct virtio_mmio_reg),
14                                PAGE_SIZE);
15                struct emul_mem emu;
16                emu = { .va_base= virtio_dev->va,
17                      .size= dev_size,
18                      .handler= virtio_mmio_emul_handler};
19                vm_emul_add_mem(vm, &emu);
20            }
21        }
22        virtio_serviceguest_cpu_init();
23    }
24 }

```

Listing 19: VirtIO VM initialization

To enable communication between the front-ends and back-ends, a connection must be established within the hypervisor. To accomplish this, it is necessary to verify, for each device, if the ID of the CPU of the VM that is being initialized matches the ID of the back-end of the device. If a match is confirmed, the ID corresponding to the Service Guest is assigned to the **sg_cpu_id** element in the device parameters struct, as shown in Listing 20.

```

1 void virtio_serviceguest_cpu_init() {
2     list_foreach( virtio_devicelist,
3                   struct virtio_deviceparams,
4                   virtio_device){
5         if(vm_config_ptr->virtio_devicelist[virtio_device->id].backend_id
↪ == cpu.vcpu->vm->id)
6             virtio_device->sg_cpu_id = cpu.id;
7     }
8 }

```

Listing 20: VirtIO service guest CPU initialization

The `virtio_ipc_init` function was implemented based on the IPC mechanism already implemented on the hypervisor and, as mentioned below, connects the shared memory between the VMs.

```
1 static void virtio_ipc_init(struct vm* vm,
2                             const struct vm_config* config) {
3     for (size_t i = 0; i < config->platform.virtiodevices_num; i++) {
4         struct virtiodevice *dev = &config->platform.virtiodevices[i];
5         uint8_t shmem_id;
6         shmem_id = vm_config_ptr->virtiodeviceclist[ dev->device_id
↪ ].shmem_id;
7         struct shmem *shmem = ipc_get_shmem(shmem_id);
8         if(shmem == NULL)
9             continue;
10        size_t size = vm_config_ptr->virtiodeviceclist[ dev->device_id
↪ ].shmem_size;
11        if(size > shmem->size)
12            size = shmem->size;
13        struct mem_region reg = {
14            .base = vm_config_ptr->virtiodeviceclist[ dev->device_id
↪ ].shmem_base,
15            .size = size,
16            .place_phys = true,
17            .phys = shmem->phys,
18            .colors = shmem->colors
19        };
20        vm_map_mem_region(vm, &reg);
21    }
22 }
```

Listing 21: VirtIO IPC initialization

4.1.5.2 Emulation

Once initialization is complete, the trap-and-emulate mechanism is fully operational. The functions that pertain to operation in execution mode will be presented below in order of execution. Following the design outlined in Section 3.3.2, the function `virtio_mmio_emul_handler` was implemented (Listing 22). This function was registered in the emulation struct initialized in the function `vm_init_virtio`, so that whenever a VM attempts to access a register of a device to which it does not have access, the hypervisor initiates the emulation and calls this function to provide a handler for the emulation.

The hypervisor passes the access information, such as the address that the guest tried to access, the operation, and the CPU being used, as parameters to the function so that it can provide the emulation handler. At the beginning of the function, the first task is to determine which device was accessed by searching for the device address that corresponds to the one passed as a parameter. If the device is not

```

1  bool virtio_mmio_emul_handler(struct emul_access *acc) {
2      struct vm* vm = cpu.vcpu->vm;
3      for(i = 0; i < vm->virtiodevices_num; i++) {
4          virtio_dev = vm->virtiodevices[i];
5          if (acc->addr >= virtio_dev.va &&
6              acc->addr <= virtio_dev.va + virtio_dev.size)
7              break;
8      }
9      if (i == vm->virtiodevices_num)
10         return false;
11     list_foreach( virtiodevicelist, struct virtiodeviceparams,
↪ virtio_device) {
12         if(virtio_device->id == virtio_dev.device_id) {
13             struct cpu_msg msg = {VIRTIO_CPUMSG_ID, VIRTIO_NOTIFY_SG,
↪ virtio_dev.device_id};
14             virtio_device->cpu_id = cpu.id;
15             virtio_device->reg_off = acc->addr - virtio_dev.va;
16             virtio_device->reg = acc->reg;
17             virtio_device->access_width = acc->width;
18             if(acc->write) {
19                 int value = vcpu_readreg(cpu.vcpu, acc->reg);
20                 virtio_device->op = WRITE_OP;
21                 virtio_device->value = value;
22             } else {
23                 virtio_device->op = READ_OP;
24             }
25             for(int j = 0; j < vm_config_ptr->vmlist[
↪ virtio_dev.backend_id ].platform.virtiodevices_num ; j++)
26                 if(vm_config_ptr->vmlist[ virtio_dev.backend_id
↪ ].platform.virtiodevices[ j ].device_id == virtio_dev.device_id) {
27                     if(vm_config_ptr->vmlist[ virtio_dev.backend_id
↪ ].platform.virtiodevices[ j ].pooling)
28                         msg.event = VIRTIO_NOTIFY_SG_POOLING;
29                     cpu_send_msg(virtio_device->sg_cpu_id, &msg);
30                     cpu_idle();
31                     return true;
32                 }
33             break;
34         }
35     }
36     return false;
37 }

```

Listing 22: VirtIO MMIO emulation handler

found, the function returns false. Next, the list of devices is traversed until the ID is found and, when the corresponding device is found, the data is inserted into the parameter struct of the device. During the search, if the device is classified as a pooling device (devices that do not have an interrupt, so they need to check every time if something change related to them), the event parameter of the *cpu_msg* is changed to **VIRTIO_NOTIFY_SG_POOLING**. Finally, it sends the *cpu_msg* through the *cpu_send_msg* function, with the device ID and places the CPU associated with the guest at idle.

Then, the CPU assigned to the Service Guest associated with the back-end that is connected to the front-end that sent the message, calls the VirtIO message handler (*virtio_handler* function), presented in Listing 23. Once this function is called, it goes to the *virtio_notify_serviceguest_handler* function or the *virtio_insert_to_pooling_queue* function if the device is configured as pooling.

```
1 static void virtio_handler(uint32_t event, uint64_t data) {
2     switch(event){
3         case VIRTIO_NOTIFY_SG: virtio_notify_serviceguest_handler(data);
4         break;
5         case VIRTIO_NOTIFY_SG_POOLING: virtio_insert_to_pooling_queue(data);
6         break;
7         case VIRTIO_READ_NOTIFY:
8         case VIRTIO_WRITE_NOTIFY: virtio_cpu_msg_handler(event, data);
9         break;
10        case VIRTIO_INJECT_INTERRUPT: virtio_inject_interrupt(data);
11        break;
12    }
13 }
```

Listing 23: VirtIO message handler

The *virtio_notify_serviceguest_handler* function search for the device ID, correspondent with the data parameter, and injects its interrupt.

```
1 static void virtio_notify_serviceguest_handler(uint64_t data) {
2     for(int i = 0; i < cpu.vcpu->vm->virtiodrives_num; i++)
3         if(cpu.vcpu->vm->virtiodrives[i].device_id == data) {
4             irq_id = cpu.vcpu->vm->virtiodrives[data].interrupt;
5             break; }
6     if(irq_id)
7         vcpu_inject_irq(cpu.vcpu, irq_id);
8 }
```

Listing 24: VirtIO service guest notification handler

In the *virtio_insert_to_pooling_queue* function, the device ID is added to the list of devices that work through pooling. After injecting the interrupt to the Service Guest, it does not have information about the

```

1 static void virtio_insert_to_pooling_queue(uint64_t data) {
2     struct virtiopoolingparams *node;
3     node->device_id = data;
4     list_push(&poolingdevicelist, (node_t*)node);
5 }

```

Listing 25: VirtIO pooling handler

action that generate the interrupt, so it sends a hypercall to the hypervisor to ask for that information about the interrupt, sending the operation of **ASK_OP (2)** or **POOLING_OP (3)** if the device is configured as pooling. In the hypercall, the Service Guest sends the type of the hypercall, which in this case is type 2, the device ID, and the operation. The remaining elements, such as the register offset and value, must be zero. When the hypercall is called, the hypervisor calls the *hvc64_handler* function, which checks the hypercall type and directs it to the appropriate hypercall type function, which then directs it to the *virtio_hypercall* function. This function returns **-HC_E_SUCCESS** if all was done successfully, **-HC_E_FAILURE** if something fails, and **-HC_E_INVALID_ARGS** if the operation parameter is not available.

```

1 unsigned long virtio_hypercall() {
2     unsigned long ret      = -HC_E_SUCCESS;
3     unsigned long dev_id   = cpu.vcpu->regs->x[1];
4     unsigned long reg_off  = cpu.vcpu->regs->x[2];
5     unsigned long op       = cpu.vcpu->regs->x[3];
6     unsigned long value    = cpu.vcpu->regs->x[4];
7     switch(op){
8         case WRITE_OP:
9         case READ_OP:      // shown in the next Listing 29
10        case POOLING_OP:    // shown in the next Listing 27
11        case ASK_OP:        // shown in the next Listing 28
12        case INTERRUPT_OP:  // shown in the next Listing 31
13        default: ret = -HC_E_INVALID_ARGS; break;
14    }
15    return ret;
16 }

```

Listing 26: VirtIO hypercall in the hypervisor

If the operation sent by the Service Guest is **ASK_OP**, the function retrieves the device parameters struct corresponding to the device ID sent by the Service Guest. Then, it sets the values of the register offset and value in the corresponding fields of the struct, based on the operation that was performed. It also updates the interrupt status register in the MMIO struct and sends a notification to the front-end using the *VIRTIO_NOTIFY_FRONT_END* macro. In case of pooling devices, it calls the *virtio_insert_to_pooling_queue* function, which inserts the device ID into the pooling queue and sends a notification to the front-end. The function then returns **-HC_E_SUCCESS** to indicate that the operation was successful.

```
1 case POOLING_OP:
2     if(dev_id != 0) {
3         ret = -HC_E_FAILURE;
4         break;
5     } else {
6         node = (struct virtiopoolingparams*)list_pop(&poolingdevicelist);
7         if(node == NULL) {
8             ret = -HC_E_FAILURE;
9             break; }
10        dev_id = node->device_id; }
```

Listing 27: VirtIO pooling operation case

The *virtio_hypercall* function, when called for an **ASK_OP** operation, first confirms that the parameters not needed for this operation, such as the register offset and value, are set to zero. It then searches for the device in the list of devices and writes the register offset, the operation, and the value in the parameters to be received by the Service Guest. This way, the Service Guest can retrieve the necessary information about the interrupt and handle it accordingly.

```
1 case ASK_OP:
2     if(reg_off != 0 || value != 0) ret = -HC_E_FAILURE;
3     else {
4         list_foreach(    virtiodevicelist,
5                          struct virtiodeviceparams,
6                          virtio_device)
7         if(virtio_device->id == dev_id) {
8             vcpu_writereg(cpu.vcpu, 1, dev_id);
9             vcpu_writereg(cpu.vcpu, 2, virtio_device->reg_off);
10            vcpu_writereg(cpu.vcpu, 3, virtio_device->op);
11            vcpu_writereg(cpu.vcpu, 4, virtio_device->value);
12            vcpu_writereg(cpu.vcpu, 5, virtio_device->access_width);
13            return ret;
14        }
15        ret = -HC_E_FAILURE;
16    }
17 break;
```

Listing 28: VirtIO asking operation case

The *virtio_hypercall_w_r_operation* function is responsible for handling write and read operations from the Service Guest to the hypervisor. It starts by verifying that the parameters passed by the Service Guest are valid, such as confirming that the register offset and the value are set to zero for write operations, and that the operation is either **WRITE_OP** or **READ_OP**.

Once the parameters are validated, the function searches for the device in the list of devices and performs

the necessary write or read operation on the specified register. If the operation is a read, it returns the value of the register to the Service Guest through the parameters.

After the operation is completed, the function sends a *cpu_msg* to the originating CPU through the *virtio_cpu_send_msg* function, to inform that the operation has been completed successfully.

```

1 case WRITE_OP:
2 case READ_OP:
3     if(!virtio_hypercall_w_r_operation(dev_id, reg_off, value))
4         ret = -HC_E_FAILURE;
5     else
6         virtio_cpu_send_msg(dev_id, op);
7 break;

```

Listing 29: VirtIO write and read operation case

In the *virtio_hypercall_w_r_operation* function, the list of devices is searched to find the desired device, and if it is found, the supplied offset is verified to correspond to the device offset in the list. If everything is correct, the value parameter on the device is updated accordingly, whether it is a write or read operation.

```

1 static bool virtio_hypercall_w_r_operation( unsigned long dev_id,
2                                           unsigned long reg_off,
3                                           unsigned long value) {
4     list_foreach(    virtiodevicelist,
5                     struct virtiodeviceparams,
6                     virtio_device) {
7         if(virtio_device->id == dev_id) {
8             if(virtio_device->reg_off != reg_off)
9                 return false;
10            virtio_device->value = value;
11            return true;
12        }
13    }
14    return false;
15 }

```

Listing 30: VirtIO write/read operation to the struct

The hypervisor checks if the parameter **reg_off** and **value** are equal to zero in the case of an **INTERRUPT_OP** operation. If this is the case, it sends a message to the CPU associated with the device. This operation is used when the Service Guest needs to notify the corresponding front-end and is carried out through the **VIRTIO_NOTIFY_FRONT_END** macro, as shown in Listing 11.

```
1 case INTERRUPT_OP:
2     if(reg_off != 0 || value != 0)
3         ret = -HC_E_FAILURE;
4     else
5         virtio_cpu_send_msg(dev_id, op);
6 break;
```

Listing 31: VirtIO interrupt operation case

When execution is directed to the function *virtio_cpu_send_msg*, a message is created and sent to the CPU that originated the trap. The message contains the event to be sent to the guest. The function then searches the list of devices to obtain the ID of the CPU associated with the device, and passes the message to the hypervisor function *cpu_send_msg* for communication between the CPUs.

```
1 static void virtio_cpu_send_msg(unsigned long dev_id, unsigned long op){
2     uint64_t data = dev_id;
3     struct cpu_msg msg = {VIRTIO_CPUMSG_ID, VIRTIO_WRITE_NOTIFY, data};
4     if(op == READ_OP)
5         msg.event = VIRTIO_READ_NOTIFY;
6     else if(op == INTERRUPT_OP)
7         msg.event = VIRTIO_INJECT_INTERRUPT;
8     list_foreach(    virtiodevicelist,
9                     struct virtiodeviceparams,
10                     virtio_device) {
11         if(virtio_device->id == dev_id) {
12             cpu_send_msg(virtio_device->cpu_id, &msg);
13             break;
14         }
15     }
16 }
```

Listing 32: VirtIO sends message to CPU

The *cpu_msg* struct sent by the Service Guest is received by the CPU of the guest that originated the trap in the *virtio_handler* function. This function redirects execution to the *virtio_cpu_msg_handler* function, which completes the trap-and-emulate mechanism of VirtIO. The function searches the list of devices for the intended recipient and, if the message was a read, writes to the register that generated the trap. After this process is complete, the previously idle guest is awoken by resuming the operation of the VM.

```

1 static void virtio_cpu_msg_handler(uint32_t event, uint64_t data) {
2     list_foreach(    virtiodevicelist,
3                     struct virtiodeviceparams,
4                     virtio_device) {
5         if(virtio_device->id == data) {
6             switch(event) {
7                 case VIRTIO_READ_NOTIFY:
8                     vcpu_writereg(    cpu.vcpu,
9                                     virtio_device->reg,
10                                    virtio_device->value);
11
12                     break;
13                 case VIRTIO_WRITE_NOTIFY:
14                     break;
15             }
16             break;
17         }
18     }

```

Listing 33: VirtIO message to a CPU handler

4.2 VirtIO Console

To communicate with a console device like a UART, the hypervisor provides various platform-specific API drivers for platforms like QEMU or zcu104. To make the drivers platform-independent, the hypervisor virtualizes them through functions that will be described in later sections. These functions make platform-specific function calls at compile-time.

4.2.1 Device Driver API

In the *uart_init* function, the UART is initialized and enabled. This function is called in Service Guest at initialization time from the CPU master. Once the device is initialized, it is fully operational. Whenever it is necessary to perform some operation on the device, the functions described in this section are called. In order for UART to transmit a character, the *uart_putc* function is used, providing the character to be transmitted as a parameter of the function. In case it is necessary to receive characters, there are 2 mechanisms, the interrupt mechanism and the storage mechanism in a First In, First Out (FIFO). To use the FIFO mechanism, the *uart_getchar* function is used to retrieve a character stored in FIFO. To use the interrupt mechanism, it is first necessary to enable the interrupt of the UART through the function *uart_enable_rxirq*, and then whenever the interrupt associated with UART is received, in the handler, the function *uart_irq_getchar* is called returning the character.

When it is necessary to disable the use of the interrupt by UART the function *uart_clear_rxirq* can be called not only to disable the interrupt but also to clear the reception buffer.

4.2.2 VirtIO Console Device Driver API

To enable the VirtIO infrastructure to communicate with the console device, functions were implemented to handle initialization and communication through both virtqueues and MMIO registers. The *virtio_console_init* function initializes the device, loads the config at the end of the MMIO registers, and sets the back-end to default values using the *virtio_back_function_end_mmio_init*.

To enable the Service Guest to transmit data received through the transmitq virtqueue, the *virtio_console_transmitq_callback_handler* function was implemented. This function retrieves a descriptor from shared memory using the macro *VIRTIO_MEM_GET_DESC*, checks if the used and available flags differ, indicating that the descriptor needs to be processed, updates the new descriptor in the virtqueue, calls the *virtio_back_end_queue_desc_read* function to process the data in the virtqueue and retrieves the data which is placed in the address of the descriptor. Finally, it transmits the data in string format through the UART using a loop with the *uart_putc* function.

When the Service Guest receives characters via UART and needs to send them to a guest, the *virtio_console_receiveq_callback_handler* function is called. It retrieves a descriptor previously made available by the guest using the macro *VIRTIO_MEM_GET_DESC*, checks if the used and available flags differ, indicating that the descriptor needs to be processed and loaded with the data, updates the new descriptor in the virtqueue, calls the *virtio_function_back_end_queue_desc_write* to load the characters into shared memory, and handles the VirtIO communication.

When a Guest tries to access a MMIO register and the hypervisor executes the trap-and-emulate mechanism, the Service Guest calls the *virtio_console_mmio_callback_handler* function. This function is responsible for modifying all writable registers and returning the contents of all registers that the hypervisor reads. To do this, the function first retrieves information about the interrupt from the hypervisor via the *virtio_hypercall*, present in Listing 34, with the **ASK_OP** operation. This is done using the instruction command "**HVC**" which causes the CPU to jump to the exception level of the hypervisor, where it returns the structure containing the MMIO access information, including the device ID, registry offset, operation, value to write (if it is a write operation) and access size (8, 16 or 32 bits).

After the hypercall, the function checks if the parameters are valid, such as if the access size is within limits and if the offset is appropriate for the console device. If everything is correct, the function proceeds with the MMIO mechanism of the VirtIO. If the register offset matches the **QueueNotify** register offset, the Service Guest calls the *virtio_console_transmitq_callback_handler* function to process the descriptor in the virtqueue and transmit the data in the shared memory. When the front-end is initialized, negotiation is performed and when writing the negotiated feature bits in the **DriverFeatures** register, the Service Guest calls the *virtio_back_end_feature_bits_negotiation* function.

```

1 static inline struct virtio_hc_ret virtio_hypercall(uint64_t dev_id,
  ↪ uint64_t reg_off, uint64_t op, uint64_t value) {
2     volatile register uint64_t x0 asm("x0") = 2; // VIRTIO_HC_ID
3     volatile register uint64_t x1 asm("x1") = dev_id;
4     volatile register uint64_t x2 asm("x2") = reg_off;
5     volatile register uint64_t x3 asm("x3") = op;
6     volatile register uint64_t x4 asm("x4") = value;
7     volatile register uint64_t x5 asm("x5");
8     struct virtio_hc_ret dev;
9     asm volatile(
10         "hvc 0\n\t" : "=r"(x0), "=r"(x1), "=r"(x2), "=r"(x3), "=r"(x4),
  ↪ "=r"(x5) : "r"(x0), "r"(x1), "r"(x2), "r"(x3), "r"(x4) : "memory"
11     );
12     dev.ret = x0;
13     dev.dev_id = x1;
14     dev.reg_off = x2;
15     dev.op = x3;
16     dev.value = x4;
17     dev.access_width = x5;
18     return dev;
19 }

```

Listing 34: VirtIO console MMIO handler

4.3 VirtIO Block

To establish communications with a block device, in this case, the SDCard, an existing drive in the Xilinx repository was used for the zcu104 platform.

4.3.1 Device driver API

The *XSdPs_LookupConfig* function is the first to be called for initializing the SD Card. It is used to analyse and extract the settings from the SD Card and it requires the ID of the SD Card controller to be passed as an argument. If the function is successful, it returns a struct *XSdPs_Config* which stores all information extracted from the device.

Next, still in the device initialization process, the *XSdPs_CfgInitialize* function is responsible for initializing a specific instance so that the driver is ready to use. It requires the address of the *XSdPs* structure as a parameter, so it can fill the information about the SD Card such as the number of sectors it contains. It also requires the configuration returned by the *XSdPs_LookupConfig* function and the base address in the virtual memory address space, which is the address contained in the *BaseAddress* parameter in the *XSdPs_Config* struct.

The function *XSdPs_CardInitialize* is responsible for the final phase of initialization of the SDCard

device, comprised of starting the Card with an Identification mode sequence, which requires sending the *XSdPs* struct as a parameter. Once the device is initialized, it is possible to communicate with the SDCard through the functions *XSdPs_ReadPolled* and *XSdPs_WritePolled*. In these functions, it is necessary to send the following parameters to read/write the device:

- InstancePtr - is a pointer to the instance to be worked on
- Arg - is the address that is to be read/written
- BlkCnt - Number of blocks to be read/written
- Buff – Pointer to the data buffer to put/take the data to be read/written

4.3.2 VirtIO Block Device Driver API

The implementation of VirtIO for the block device is similar to the console device, but it only uses a single virtqueue for communication between the front-end and the back-end. The *virtio_blk_init* function is used for initializing the device.

Afterwards, the *virtio_blk_callback_handler* function was implemented to handle requests from the front-end. This function retrieves the descriptor, performs the VirtIO processing, and extracts the struct containing the request from the shared memory provided by the front-end. The request is then processed and executed, and the result of the operation is sent back to the front-end in the same buffer.

Similar to the console device, the block device also has a function to handle MMIO, using the same algorithm. However, it applies to the block configuration registers and uses a single virtqueue.

Functional validation and Testing

In addition to the VirtIO infrastructure implementation, tests were conducted to confirm that the features were properly implemented. The first test verified the correct implementation of the virtqueues through an IPC mechanism available in the hypervisor. Next, when the trap-and-emulate mechanism was implemented and the VirtIO transport was developed through MMIO, a test was performed using a baremetal application executing in QEMU to check if the front-ends that were created worked with the back-ends provided by the emulator. Once the VirtIO infrastructure tests were confirmed, the devices were tested, starting with the console device and then moving on to the block device.

5.1 Validation: Setup

The validation of each phase was carried out on two different platforms, with the infrastructure and VirtIO devices being validated in the initial stages on QEMU and in the final stages on the zcu104 board. QEMU is a platform that emulates multi-platforms, based on Linux, frequently used to verify the behaviour of software developed on different hardware architectures. This platform runs on a Linux system, in this case, Ubuntu OS. The zcu104 board is equipped with the Zynq UltraScale+ MPSoC, which combines a powerful Processing System (PS) and Programmable Logic (PL) on the same device. The PS features the ARM Cortex-A53 64-bit quad-core processor and the Cortex-R5 dual-core RT processor. On this thesis was used the Arm Cortex-A53 on both platforms.

5.2 Validation: Methodology

The methodology used for validating the VirtIO infrastructure consists of two tests to be performed on the QEMU platform. The first test aims to validate the communications through hypercalls, between the front-end and the back-end using the virtqueues loaded into shared memory between the guest and the Service Guest. In the second test, once the communications through hypercalls have been validated, the trap-and-emulate mechanism was validated by placing a front-end on a baremetal to make requests to the back-end embedded in the platform. With this test, it was possible to validate the initialization of the

VirtIO infrastructure and the communications through the trap-and-emulate mechanism by writing to the MMIO registers allocated on the Service Guest. In the final phase, to validate the VirtIO devices created, the methodology consisted of validating the virtio-console device on both platforms, that is, on QEMU and on the zcu104 board. The test to validate the console device, consisted of communications between the front-end of the console device embedded in the Linux guest and the back-end created and embedded in the Service Guest. Unlike the console device test, in which the UART device driver was already embedded, in the block it was necessary to test the imported Xilinx device driver initially, and altered for the project. For the validation of the block device, the methodology mentioned for validating the console device was followed. To compile the image of Bao, to be loaded in the QEMU or in the SD Card to be stored on the zcu104, it was used the compiler provided by ARM. The compiler used is the AArch64 ELF baremetal target (gcc-arm-10.3-2021.07-x86_64-aarch64-none-elf), as the host system is based on x86 architecture.

5.3 Validation: VirtIO IPC

When the infrastructure was finalized, it was necessary to test the virtqueues mechanism. Specifically, it was necessary to test if the flags were toggled correctly and if the virtqueue was correctly loaded in the shared memory. Additionally, it was necessary to test if the virtqueues were accurately accessed by the receiver. Finally, it was necessary to test if the descriptors were subsequently correctly classified as used.

The architecture outlined in Section 3.3.1 was implemented to conduct this test. This involved configuring at least one interrupt for each virtqueue during initialization. Information to be sent to the Service Guest was added by loading a descriptor into the shared memory virtqueue and using a hypercall to notify the receiver of new data to process. Each interrupt resulting from a hypercall had a corresponding function that incremented a flag, which was checked in the main function infinite loop to prevent long execution from blocking new entries in the Interruption Service Routines (ISR). The example used for this test involved modifying a string allocated in the Service Guest by sending write or read commands. After processing the command, the Service Guest loaded the descriptor with the information to be returned through the receiveq for the device console and the same descriptor in the virtqueue requestq for the device block. A new hypercall was then sent to notify the original guest of the response so that it could access it. This test revealed a bug in the implementation of virtqueues when swapping the available flags to used when the buffer was processed and the operation of the front-end was executed.

5.4 Validation: front-end for virtio-console device on QEMU

This test aimed to validate that the front-ends, which were validated with the developed back-ends, were compatible with the back-ends available on the QEMU platform, which are similar to those used in embedded Linux systems. The test was designed as shown in Figure 34. The goal of the baremetal was to perform transmissions through the virtqueue transmitq and receive information through the receiveq

allocated in the driver guest memory. Then, QEMU was notified of the transitions from the Driver, and since QEMU has access to the entire memory of the guest, it accessed the virtqueue to obtain the address and size information it needs to pass to the UART. In order for QEMU to send characters received by the UART, the driver guest needs to provide empty descriptors in the receiveq virtqueue, ready to be filled by the back-end.

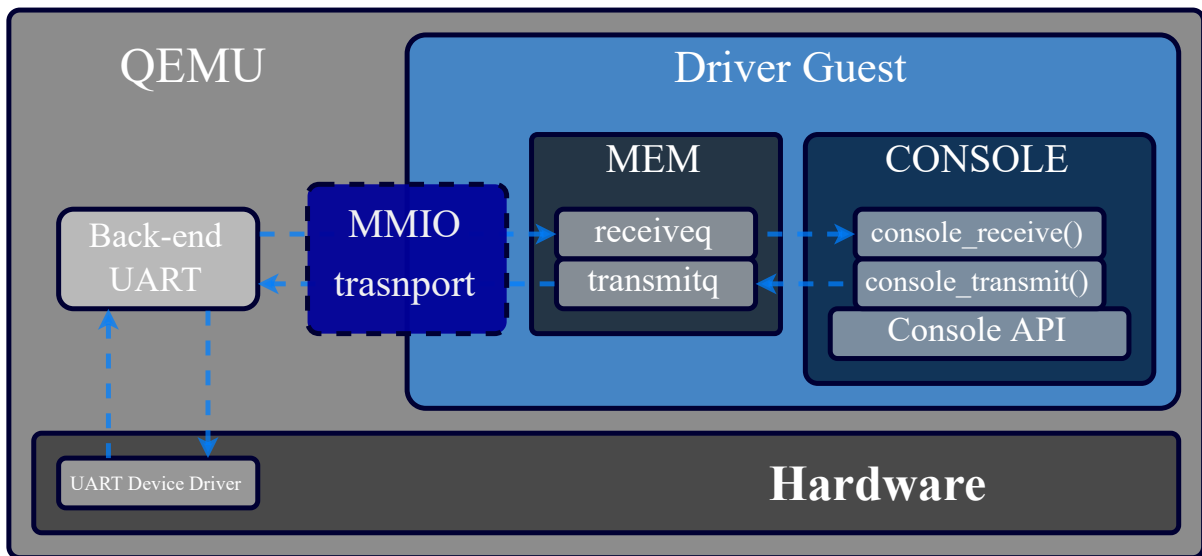


Figure 34: Test front-end on QEMU diagram

Due to the fact that the front-end is not compatible with legacy, it was necessary to add to the QEMU command the argument represented in the Listing 35.

```
1 -global virtio-mmio.force-legacy=false
```

Listing 35: VirtIO MMIO disable legacy on QEMU

To inform QEMU of the type of virtqueues to be used in the communication between the front-end and the back-end (the former of which can only communicate through packed virtqueue) the command represented in Listing 36 was used.

```
1 -device virtio-serial-device,packed=on
```

Listing 36: Enable packed virtqueue on VirtIO

To obtain the addresses and the interrupt that is assigned to the VirtIO device provided by QEMU, the argument described in Listing 37 was included in the QEMU command.

```
1 dumpdtb=.../virt.dtb
```

Listing 37: Extract devicetree from QEMU command

The resulting file from the extraction is in devicetree format and is not readable by the user. To solve this, it is necessary to convert the file to devicetree source format using the command represented in Listing 38.

```
1 dtc -I dtb -O dts -o .../virt.dts .../virt.dtb
```

Listing 38: Convert from devicetree format to devicetree source

By analysing the devicetree converted file from QEMU, in Listing 39 it is possible to observe that several memory addresses and interrupts are allocated. At boot time, QEMU assigns VirtIO devices from the highest address to the lowest address, i.e. it starts at address 0xa003e00 with the associated interrupt of 0x2f.

In the end, the functional component of the VirtIO infrastructure and the MMIO transport were validated.

5.5 Validation: back-end of the Service Guest with front-end of the Linux for virtio-console

To carry out this test, a hypervisor configuration was created in which one of the CPUs available on the used platform was assigned to the Service Guest and another CPU was assigned to a previously compiled embedded Linux image. To test the device console, the UART peripheral was assigned to the Service Guest, so that it can transmit and receive data through the command line associated with the peripheral.

The Service Guest, as represented in Listing 41, is at virtual memory address 0x00000000 and has a size of 0x08000000. Two devices are assigned, UART0 on the board to carry out the operations required by the device console of VirtIO and a timer. In the VirtIO device configuration, the Service Guest was assigned a device with the ID 0, interrupt number 52 and this Guest was defined as a back-end so that, at the time of initialization, the hypervisor saves its ID by connecting the device to this Service Guest.

In the Linux configuration, as represented in Listing 42, it was defined that the guest would be addressed in region 0x50000000, with a size of 0x20000000. The Linux Guest was assigned with the devices of UART1 and the timer, which is essential for the Kernel to work. Next, a VirtIO device was assigned with the ID 0 and the same address as the one defined in the devicetree, represented in Listing 43.

The embedded Linux image used was built from scratch using buildroot, an open-source and fast framework that allows the creation of embedded Linux systems for different boards. Once the Linux image

```

1 virtio_mmio@a000000 {
2     dma-coherent;
3     interrupts = <0x00 0x10 0x01>;
4     reg = <0x00 0xa000000 0x00 0x200>;
5     compatible = "virtio,mmio";
6 };
7 /*...*/
8 virtio_mmio@a003e00 {
9     dma-coherent;
10    interrupts = <0x00 0x2f 0x01>;
11    reg = <0x00 0xa003e00 0x00 0x200>;
12    compatible = "virtio,mmio";
13 };
14 /*...*/
15 pl011@9120000 {
16     clock-names = "uartclk\0apb_pclk";
17     clocks = <0x8000 0x8000>;
18     interrupts = <0x00 0x0c 0x04>;
19     reg = <0x00 0x9120000 0x00 0x1000>;
20     compatible = "arm,pl011\0arm,primecell";
21 };
22 pl011@9110000 {
23     /*...*/
24 };
25 pl011@9100000 {
26     /*...*/
27 };
28 pl011@90000000 {
29     /*...*/
30 };

```

Listing 39: QEMU devicetree source

was generated, its devicetree was changed to include a VirtIO device so that Linux, at its initialization, is aware that it needs to look for, identify and carry out the negotiation of the VirtIO device.

Once the hypervisor was compiled with both guests, i.e., the Service Guest and the Linux, it was initially run on QEMU and later on the zcu104 board. At the end of the initialization of both guests, it was verified that, in the **dev** folder that corresponds to the file where the Linux devices are located, there was the device “**HVC**” that corresponds to a device that is virtualized through the VirtIO protocol, using the front-end available in the Linux environment. Next, an echo was performed for these devices, and it was verified that the messages typed on the Linux side were transmitted from the UART on the Service Guest side. A small issue was found with this test that, when the “/r” character was sent, the Linux front-end sent two descriptors separately, that is, one with the message and the other with the “/r/n”, which was

```
1 struct config config = {
2     .shmemlist_size = 1,
3     .shmemlist = (struct shmem[]) {
4         [0] = { .size = 0x00020000 } }
5     .virtio_devicelist_size = 1,
6     .virtio_devicelist = (struct virtio_device[]) {
7         [0] = { .shmem_id = 0, .shmem_base = 0x40000000,
8                 .shmem_size = 0x00020000 } }
9     .vmlist_size = 2,
10    .vmlist = {
11        { /* Service Guest Configuration */,
12          { /* Linux Guest Configuration*/ } }
13    };
```

Listing 40: Bao configuration

```
1 .image = { .base_addr = 0x00000000, /* Image loader */ },
2 .entry = 0x00000000,
3 .platform = {
4     .cpu_num = 1,
5     .region_num = 1,
6     .regions = (struct mem_region[]) {
7         { .base = 0x00000000, .size = 0x80000000 } },
8     .dev_num = 2,
9     .devs = (struct dev_region[]) {
10         { // UART0 - zcu104
11             .pa = 0xFF000000, .va = 0xFF000000,
12             .size = 0x10000, .interrupt_num = 1,
13             .interrupts = (irqid_t[]) {53} },
14         { // TIMER
15             .interrupt_num = 1, .interrupts = (irqid_t[]) {27} } },
16     .virtio_devices_num = 1,
17     .virtio_devices = (struct virtio_device[]) {
18         { .device_id = 0, .is_back_end = true, .interrupt = 52 } },
19     .arch = { .gic = { /* GIC configuration */ } }
20 },
```

Listing 41: Service guest configuration

```

1  .image = { .base_addr = 0x50200000, /* Image loader */ },
2  .entry = 0x50200000,
3  .platform = {
4      .cpu_num = 1,
5      .region_num = 1,
6      .regions = (struct mem_region[]) {
7          { .base = 0x50000000, .size = 0x20000000,
8            .place_phys = true, .phys = 0x50000000 } },
9      .dev_num = 2,
10     .devs = (struct dev_region[]) {
11         { // UART1 - zcu104
12             .pa = 0xFF010000, .va = 0xFF010000, .size = 0x10000,
13             .interrupt_num = 1, .interrupts = (irqid_t[]) {54} },
14         { // TIMER
15             .interrupt_num = 1, .interrupts = (irqid_t[]) {27} } },
16     .virtiodevices_num = 1,
17     .virtiodevices = (struct virtiodevice[]) {
18         { .va = 0x80000000, .size = 0x200, .device_id = 0 } }
19     .arch = { .gic = { /* GIC configuration */ } }
20 },

```

Listing 42: Linux guest configuration

```

1  memory@50000000 {
2      device_type = "memory";
3      reg = <0x0 0x50000000 0x0 0x20000000>; };
4  virtio_mmio@80000000 {
5      interrupts = <0x0 0x28 0x1>;
6      reg = <0x0 0x80000000 0x0 0x200>;
7      compatible = "virtio,mmio"; };
8  serial@ff010000 {
9      u-boot,dm-pre-reloc;
10     compatible = "cdns,uart-r1p12", "xlnx,xuartps";
11     status = "okay";
12     interrupts = <0x0 0x16 0x4>;
13     reg = <0x0 0xff010000 0x0 0x1000>;
14     clock-names = "uart_clk", "pclk";
15     clocks = <&uartclk &uartclk>; };

```

Listing 43: Linux devicetree changes

solved post-haste. At the end of the test, it was concluded that the device console was functionally correct and compatible with the virtio-console front-end on Linux.

5.6 Validation: back-end of the Service Guest with Linux front-end for virtio-blk

In this test, it was initially necessary to import the drivers mentioned in the Section 4.3.1 into the project. Next, an example was executed in which the controller and the SD Card were initialized. A write was performed in a specific sector and the same sector was immediately read in order to, at the end, be compared, as presented in Listing 44.

```
1  #define TEST 7
2  #define NUM_BLOCKS 16          /* Number of SD blocks to test */
3  #define SECTOR_OFFSET 204800  /* Sector offset to test */
4  u32 FileSize = (512*NUM_BLOCKS);
5  u32 Sector = SECTOR_OFFSET;
6  for(BuffCnt = 0; BuffCnt < FileSize; BuffCnt++)
7      Source[BuffCnt] = TEST + BuffCnt;
8  Status = Sdps_init(&SdInstance);
9  /* Check if was successfully */
10 Status = XSdPs_WritePolled(&SdInstance, Sector, NUM_BLOCKS, Source);
11 /* Check if was successfully */
12 Status = XSdPs_ReadPolled(&SdInstance, Sector, NUM_BLOCKS, Destination);
13 /* Check if was successfully */
14 for(BuffCnt = 0; BuffCnt < FileSize; BuffCnt++)
15     if(Source[BuffCnt] != Destination[BuffCnt])
16         return XST_FAILURE;
```

Listing 44: Block device driver test

Through the hypervisor configuration, the Service Guest was assigned the device block so that, when the device driver of the block performs operations on its address, the hypervisor allows access to physical addresses 0xFF170000 – FF17FFFF.

```
1  .dev_num = 3,
2  .devs = (struct dev_region[]) {
3      { /* UART0 - zcu104 */ },
4      { /* Block - zcu104 */
5          .pa = 0xFF170000, .va = 0xFF170000, .size = 0x10000 },
6      { /* TIMER */ } },
```

Listing 45: Service guest configuration - block device added

Once the hypervisor was compiled with both guests, that is, the Service Guest and Linux, Bao was loaded onto a SD Card and run on the zcu104. When Linux was booting, a problem was verified that came from the adaptation made in the implementation of the MMIO mechanism. This adaptation caused that when the Linux CPU left the idle state, it would lose all information present in the registers. To solve this problem, the CPU was placed in Wait For Interrupt (WFI) instead of idle. This implementation originated an issue in this test, in which, once the CPU is in WFI, it would resume the execution of the Linux boot regardless of the interruption received, that is, an external interruption, a timer interruption or an interruption from the Service Guest reporting that Linux can proceed with booting VirtIO. What has been verified is that if VirtIO initialization takes longer and Linux returns to the execution state halfway through due to an interruption other than the expected one, the value that will be read by the VirtIO initialization will be considered “garbage”, because it is a random value found in the address that was accessed. A solution to this problem would be to save the necessary registers for the Linux boot, put the CPU assigned to Linux in idle and when it can return to the running state, Bao will load the registers previously, so that the Linux boot can proceed successfully. Another solution is to make Linux resume the execution state only for the interruption that concerns VirtIO initialization. An attempt was made to simplify the communication of the mechanism as much as possible on the part of the Service Guest, but Linux always ended up interrupting communication with the Service Guest. With this experiment, it was possible to observe that the initialization could continue a little longer, but it could not finish successfully.

Also in this test, the baremetal that validate the SD Card controller driver was run over Bao hypervisor, to validate if the SD Card controller driver is working with the hypervisor. With this, it was possible to observe that Bao does not allow the SD Card, where the image is loaded, to be replaced with other SD Card to perform the initialization. A possible solution is to load the Bao image via Joint Test Action Group (JTAG), or using the other SD Card controller provided by zcu104.

Conclusion

Bao is a hypervisor designed to support MCSs, with a focus on ensuring security. As it assigns peripherals to VMs at boot time using pass-through, it is not possible to share peripherals while also maintaining the requirements of MCSs. In order to be able to share a peripheral, a viable solution was presented that meets the requirements of the MCS, that is, a VirtIO infrastructure and respective devices.

In the first phase of the present work, a survey of the state of the art was carried out on how the problem of sharing peripherals was solved in other popular hypervisors such as KVM, Xen, Xvisor and ACRN. With the research carried out, it was possible to have a broader understanding on how the existing hypervisors dealt with the sharing of peripherals, even if none of the mechanisms used were strongly viable for a static partition hypervisor, since they do not comply with the various necessary requirements associated. Then, with a stronger foundation on how to solve the problem of sharing peripherals, it was decided to use a standard VirtIO protocol to be responsible for the communications between the peripherals and the different guests. The main challenge in this protocol is that, as present in most hypervisors, it was designed to have back-ends incorporated into the hypervisor. Since the hypervisor aims to remain simple and minimalist, it was necessary to think of a solution that would have the least impact. After gaining a deeper understanding of the VirtIO protocol and its capabilities, a solution was devised that involves having a dedicated Service Guest with access to peripherals that will be shared among different guests. This solution was based on the implementations of Xen and ACRN, both of which utilize a privileged guest to handle peripheral operations.

Subsequently, in Section 3, an analysis was carried out explaining in depth how the VirtIO protocol works through the knowledge acquired in the first phase, knowledge such as how virtqueues work, how initialization is carried out, i.e., how negotiations are carried out, the differences between split and packed virtqueues, and what transport types exist. Also in this section, it was explained how the VirtIO Console and VirtIO Block peripherals work and what configurations can be used in them. In an early outline, a VirtIO infrastructure was designed and implemented to test how the mechanism works using the hypercalls of the Bao hypervisor. This implementation did not incorporate any type of VirtIO initialization, i.e., the feature bits negotiations; however, it was useful to implement the exchanges between virtqueues and to confirm its operation. With the exchanges between virtqueues validated, a trap-and-emulate mechanism was designed

to be incorporated into the hypervisor so that a guest can indirectly access the MMIO registers allocated in the Service guest, thus was the VirtIO transport mechanism designed. This mechanism is essential for VirtIO to function.

The implementation is thoroughly explained in Section 4, following an execution order. The section begins with the implementation of the VirtIO initialization, followed by the implementation of functions for writing and reading shared memory descriptors. It then covers the creation of the MMIO register structure and a detailed explanation of how the trap-and-emulate mechanism of VirtIO was implemented. Additionally, the section also covers the implementation of VirtIO devices, starting with the device drivers that the Service Guest accesses, and ending with the VirtIO device driver that facilitates communication between guests and the Service Guest.

In the various stages of the thesis, validations were carried out to verify each of the phases. The initial validation confirmed the functional veracity of the communications of the virtqueues through the shared memory. The second validation was carried out to verify the trap-and-emulate mechanism integrated in the Bao through QEMU to verify that the VirtIO infrastructure works correctly with a back-end driver existing in the emulator. Finally, in the final phase, the VirtIO infrastructure and device drivers were validated on the zcu104 board.

Performing an analysis of the proposed objectives and comparing them with what was implemented in the thesis, it can be concluded that most of them were achieved, i.e., the VirtIO infrastructure was fully implemented according to the VirtIO protocol, the VirtIO device driver for the console was operating on both platforms (QEMU and the zcu104 board). Unfortunately, it was not possible to validate the implementation of the VirtIO block device due to its problems related to the Linux boot time, along with a physical limitation with the board only having one available SD Card controller, as mentioned in Section 5. In general, it should be noted that most of the objectives were successfully met, although there are still a lot of work and improvements to be made.

6.1 Future Work

With the initial infrastructure implemented and validated, further improvements are needed to optimize its operation, enhance compatibility with other systems, and expand support for a wider range of boards. To achieve these goals, the following topics are proposed for future development

6.1.1 Split virtqueues

The VirtIO infrastructure developed utilizes the packed virtqueue method. However, many existing VirtIO implementations on various hypervisors rely on the split virtqueue method, which has long been the standard method of data transportation. Implementing this feature would greatly enhance the compatibility with other systems.

6.1.2 VirtIO devices

The VirtIO protocol is prepared to support several devices, and as such, Linux is prepared to support these devices as well. Therefore, it is suggested to implement other devices that are necessary for the applications that the Bao hypervisor can support, such as:

- Inter-Integrated Circuit (I2C) adapter device for applications to communicate with devices via the I2C protocol;
- General-Purpose Input/Output (GPIO) device to support I/O ports that can be configured with Input or Output mode with logic levels of low, or high;
- Bluetooth device to support the communications that the applications running on Bao need to carry out for wireless devices.
- Controller Area Network (CAN) device to support that communication protocol. Since one of the main objectives of Bao is to integrate internal devices in the automotive sector, it is essential for Bao to have incorporated CAN devices as this is the main protocol in this branch.

6.1.3 Platforms

The VirtIO infrastructure is platform-agnostic, however, the drivers for the devices are specific to each platform. To improve compatibility, it is suggested to develop drivers for devices that are not yet supported on different platforms and validate the infrastructure on these platforms. Additionally, it is proposed to implement the VirtIO infrastructure and corresponding devices for Reduced Instruction Set Computer (RISC)-V architectures, as it is ready to be integrated into these architectures.

6.1.4 FreeRTOS system

The Service Guest was initially developed for a baremetal system, as the focus was on testing the functionality of the VirtIO infrastructure and corresponding devices. However, when the system needs to support a large number of devices and handle multiple guests making requests simultaneously, this system is not suitable as it cannot perform multiple tasks simultaneously. One viable solution is to use multiple Service Guests created on separate baremetal systems to divide the devices. However, a more efficient alternative is to implement the Service Guest on a FreeRTOS, which is specifically designed for multitasking execution.

6.1.5 Performance evaluation

The VirtIO protocol has a lower performance compared to the pass-through method due to the additional delays caused by the communication process. The pass-through method allows the guest direct

access to the device, eliminating these delays. To understand how much performance was lost, it is suggested to conduct a performance analysis of the protocol, so that when implementing an application that requires peripheral sharing, the trade-offs between using the protocol and using direct pass-through can be evaluated.

Bibliography

- [1] M. Barr. “Programming embedded systems: with C and GNU development tools”. In: *Introduction*. 2006 (cit. on p. 1).
- [2] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. “Virtualization on TrustZone-Enabled Microcontrollers? Voilà!” In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2019. doi: 10.1109/RTAS.2019.00032 (cit. on p. 1).
- [3] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto. “BAO: A lightweight static partitioning hypervisor for modern multi-core embedded systems”. In: *Workshop on Next Generation Real-Time Embedded Systems*. 2020. doi: 10.4230/OASICS.NG-RES.2020.3 (cit. on pp. 1–3, 5).
- [4] B. Sá, J. Martins, and S. Pinto. “A First Look at RISC-V Virtualization From an Embedded Systems Perspective”. In: *IEEE Transactions on Computers*. 2022. doi: 10.1109/TC.2021.3124320 (cit. on p. 1).
- [5] D. Cerdeira, J. Martins, N. Santos, and S. Pinto. “ReZone: Disarming TrustZone with TEE Privilege Reduction”. In: *31st USENIX Security Symposium*. 2022. url: <https://www.usenix.org/conference/usenixsecurity22/presentation/cerdeira> (cit. on pp. 1, 2).
- [6] S. Pinto and N. Santos. “Demystifying arm trustzone: A comprehensive survey”. In: *ACM Computing Surveys*. 2019. doi: 10.1145/3291047 (cit. on p. 1).
- [7] M. Costa, R. Moreira, J. Cabral, J. Dias, and S. Pinto. “Wall Screen: An Ultra-High Definition Video-Card for the Internet of Things”. In: *IEEE MultiMedia*. 2020. doi: 10.1109/MMUL.2020.3011595 (cit. on pp. 1, 3).
- [8] “Armv8-A virtualization”. In: 2019. url: <http://www.arm.com> (cit. on p. 2).
- [9] C. Dall. “The Design, Implementation, and Evaluation of Software and Architectural Support for ARM Virtualization”. In: 2018 (cit. on pp. 2, 5).
- [10] A. Kivity, U. Lublin, A. Liguori, Y. Kamay, and D. Laor. “kvm: the Linux virtual machine monitor”. In: *Proceedings of the Linux Symposium*. 2007 (cit. on p. 2).

- [11] J. Y. Hwang, S. B. Suh, S. K. Heo, C. J. Park, J. M. Ryu, S. Y. Park, and C. R. Kim. "Xen on ARM: System virtualization using xen hypervisor for ARM-based secure mobile phones". In: *5th IEEE Consumer Communications and Networking Conference*. 2008. doi: 10.1109/ccnc08.2007.64 (cit. on p. 2).
- [12] S. Xi, J. Wilson, C. Lu, and C. Gill. "RT-Xen: Towards real-time hypervisor scheduling in Xen". In: *9th ACM International Conference on Embedded Software*. 2011. doi: 10.1145/2038642.2038651 (cit. on p. 2).
- [13] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer. "Look Mum, no VM Exits! (Almost)". In: (2017). url: <http://arxiv.org/abs/1705.06932> (cit. on p. 2).
- [14] E. Kou. "Virtualization for embedded industrial systems". In: 2018. url: <https://www.ti.com/cn/lit/wp/spry317b/spry317b.pdf> (cit. on p. 2).
- [15] J. Martins. "Ontology-Driven Metamodeling Towards Hypervisor Design Automation: Microkernel Infrastructure". In: 2018 (cit. on pp. 5, 7).
- [16] C. Dall, S. W. Li, J. T. Lim, and J. Nieh. "ARM Virtualization". In: *Operating Systems Review (ACM)*. 2018. doi: 10.1109/ISCA.2016.35 (cit. on p. 5).
- [17] B. Kauer. "Improving System Security Through TCB Reduction". In: 2014 (cit. on p. 5).
- [18] P. Burgio, M. Bertogna, N. Capodiecici, R. Cavicchioli, M. Sojka, P. Houdek, A. Marongiu, P. Gai, C. Scordino, and B. Morelli. "A software stack for next-generation automotive systems on many-core heterogeneous platforms". In: *Microprocessors and Microsystems*. 2017. doi: 10.1016/j.micpro.2017.06.016 (cit. on p. 5).
- [19] G. Heiser. "The role of virtualization in embedded systems". In: *1st Workshop on Isolation and Integration in Embedded Systems*. 2008. doi: 10.1145/1435458.1435461 (cit. on p. 6).
- [20] VMware. "Understanding Full Virtualization, ParaVirtualization, and Hardware Assist". In: *Memory*. 2007 (cit. on p. 7).
- [21] M. T. Jones. "Virtio : An I / O virtualization framework for Linux Paravirtualized I / O with KVM and lguest". In: *IBM Developer Works*. 2010 (cit. on pp. 7, 14).
- [22] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. "LTZVisor: TrustZone is the key". In: *Leibniz International Proceedings in Informatics*. 2017. doi: 10.4230/LIPIcs.ECRTS.2017.4 (cit. on p. 9).
- [23] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares. "Towards a TrustZone-assisted hypervisor for real-time embedded systems". In: (2017). doi: 10.1109/LCA.2016.2617308 (cit. on p. 9).
- [24] B. Gamsa, O. Krieger, and M. Stumm. "Optimizing IPC performance for shared-memory multi-processors". In: *Proceedings of the International Conference on Parallel Processing*. 1994. doi: 10.1109/ICPP.1994.144 (cit. on p. 11).

- [25] S. Patni, J. George, P. Lahoti, and J. Abraham. "A zero-copy fast channel for inter-guest and guest-host communication using VirtIO-serial". In: *1st International Conference on Next Generation Computing Technologies*. 2016. doi: 10.1109/NGCT.2015.7375072 (cit. on pp. 11, 14).
- [26] A. Oliveira, J. Martins, J. Cabral, A. Tavares, and S. Pinto. "TZ- VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor". In: *IEEE International Symposium on Industrial Electronics*. 2018. doi: 10.1109/ISIE.2018.8433781 (cit. on pp. 11, 14).
- [27] F. Diakhaté, M. Perache, R. Namyst, and H. Jourden. "Efficient shared memory message passing for Inter-VM communications". In: *Lecture Notes in Computer Science*. 2009. doi: 10.1007/978-3-642-00955-6_7 (cit. on p. 11).
- [28] M. Kleidermacher and D. Kleidermacher. "Embedded Systems Security". In: *Embedded Systems Security*. 2012. doi: 10.1016/C2010-0-67275-0 (cit. on p. 11).
- [29] R. Russell. "Virtio: Towards a de-facto standard for virtual I/O devices". In: *Operating Systems Review (ACM)*. 2008. doi: 10.1145/1400097.1400108 (cit. on p. 13).
- [30] J. H. Kim and H. W. Jin. "Virtio Front-End Network Driver for RTEMS Operating System". In: *IEEE Embedded Systems Letters*. 2020. doi: 10.1109/LES.2019.2957570 (cit. on p. 13).
- [31] G. Motika and S. Weiss. "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard". In: *Computer Standards and Interfaces*. 2012. doi: 10.1016/j.csi.2011.05.002 (cit. on p. 14).
- [32] "Virtual I/O Device (VIRTIO) Version 1.1 Specification URIs". In: 2019. url: <https://docs.oasis-open.org/virtio/virtio/v1.1/cs01/virtio-v1.1-cs01.pdf> (cit. on pp. 15, 24–28).
- [33] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi. "Embedded hypervisor xvior: A comparative analysis". In: *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015. doi: 10.1109/PDP.2015.108 (cit. on pp. 16–18).
- [34] M. J. D. PAGARE and D. N. A. KOLI. "A technical review on comparison of XEN and KVM hypervisors an analysis of virtualization technologies". In: *IJARCCCE*. 2014. doi: 10.17148/ijarcce.2014.31234 (cit. on pp. 16, 17).
- [35] "Virtio devices high-level design — Project ACRN™ v 2.1-unstable documentation". In: url: <https://projectacrn.github.io/2.1/developer-guides/hld/hld-virtio-devices.html> (cit. on p. 19).
- [36] E. P. Martín. "Packed virtqueue: How to reduce overhead with virtio". In: 2020. url: <https://www.redhat.com/en/blog/packed-virtqueue-how-reduce-overhead-virtio> (cit. on p. 27).

