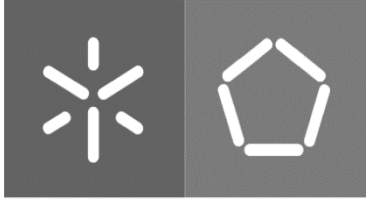
Universidade do Minho
Escola de Engenharia

Pedro Miguel Silvestre Machado

Self-Secured Devices: Securing shared device access on TrustZone-based systems

Outubro de 2018



Universidade do Minho
Escola de Engenharia

Pedro Miguel Silvestre Machado

Self-Secured Devices: Securing shared device access on TrustZone-based systems

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efectuado sob a orientação do
Professor Doutor Sandro Pinto

Outubro de 2018

Declaração do Autor

Nome: Pedro Miguel Silvestre Machado
Correio Eletrónico: a68526@alunos.uminho.pt
Cartão de Cidadão: 14390510 4ZY0

Titulo da dissertação: Self-Secured Devices: Securing shared device access on TrustZone-based systems
Ano de conclusão: 2018
Orientador: Professor Doutor Sandro Pinto

Designação do Mestrado: Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e Computadores
Área de Especialização: Sistemas Embebidos e Computadores
Escola de Engenharia
Departamento de Eletrónica Industrial

Universidade do Minho, 29/10/2018

Assinatura: Pedro Miguel Silvestre Machado

Acknowledgements

It has been a challenging journey, but there is no easy path to success. And in the end, I'm really grateful for this journey becoming into a personally rewarding one. Nothing would be possible without the people that surrounded me throughout it, and I will hereby try to express my gratitude towards them.

Firstly, I would like to thank my advisor Dr. Sandro Pinto, who challenged me with his pioneer ideas, always made himself available when I most needed, provided me opportunities to improve myself, for all the inexhaustible support that gave me, and for treating me not only as a student but also as a friend. Thanks to Dr. Adriano Tavares for also giving me advice and insight at our meetings and for sharing his extensive knowledge during my master's.

I would also like to thank all my lab colleagues, my companions throughout this journey: Ailton Lopes, Ângelo Ribeiro, David Cerdeira, Franciso Petrucci, Hugo Araújo, José Martins, José Ribeiro, José Silva, Nuno Silva, Ricardo Roriz, and Sérgio Pereira. That proved that the ESRG is more than just a group of extremely competent people, it's a family. Where we all support each other and overcome obstacles as one, with each of these individuals' skill set combined great things can be accomplished.

Thanks to my long-term friends: Álvaro Silvestre, Diogo Pinto, João Peixoto, João Santos, João Coutinho, Jorge Carvalho, Jorge Pereira, Miguel Rego, Paulo Pontes, Ricardo Marques, and Tiago do Val with whom, lately, I was not able to spend as much time as I would like, but were always there for me.

Special thanks to my family for being my biggest supporters and to my girlfriend, Carolina Guimarães, who unconditionally supported me all along and gave me strength to carry on, especially in those moments I was feeling down, sorry for my absense during this period.

Abstract

With the advent of the Internet of Things (IoT), security emerged as a significant requirement in the embedded systems development. Attacks against embedded systems infrastructures have been increasing, because security is being misconstrued as the addition of features to the system in a later stage of the system development. A new change in the way that systems are being developed is needed, to start guaranteeing security from the outset.

ARM Trustzone is a hardware technology that adds significant value to the security picture. TrustZone promotes hardware as the initial root of trust and has been gaining particular attention in the embedded space due to the massive presence of ARM processors into the market. TrustZone technology splits the hardware and software resources into two worlds - the *secure world*, dedicated to the secure processing, and the *non-secure world* for everything else. A lot of research has been done around TrustZone technology, ranging from efficient and secure virtualization solutions to trusted execution environments (TEE). Both cases, despite targeting different applications with different requirements, consolidate multiple virtual environments into the same platform and necessarily need to share resources among them. Currently, hardware devices on TrustZone-enabled system-on-chips (SoC) can only be configured as secure or non-secure, which means the dual-world concept of TrustZone is not spread to the devices itself. With this direct assignment method both worlds are unable to use the same device unless it is entirely duplicated, significantly increasing overall hardware costs. Existing shared device access on TrustZone-based architectures have been shown to negatively impact the overall system in terms of security and performance, besides often come with associated engineering effort or substantial hardware costs.

This thesis proposes the concept of self-secured devices, a novel approach for shared device access in TrustZone-based architectures. Self-secured devices extend the TrustZone dual-world concept to the inner logic of the device by splitting the device's hardware logic into a secure and non-secure interface. The implemented solution was deployed on LTZVisor, an open-source and in-house lightweight TrustZone-assisted hypervisor, and the achieved results are encouraging, demonstrating that we increase the security properties of the system for an acceptable cost in terms of hardware.

Resumo

Com o advento da Internet das Coisas (IoT), começaram a surgir mais preocupações relativas à segurança no desenvolvimento de sistemas embebidos. Os ataques contra infraestruturas deste tipo de sistemas têm vindo a aumentar exponencialmente, dado que a segurança tem vindo a ser reforçada através da adição de várias funcionalidades ao invés de ser considerada desde a fase inicial de desenvolvimento do sistema.

ARM TrustZone, é um exemplo de uma tecnologia de *hardware* que veio contribuir significativamente para o panorama de segurança. A tecnologia TrustZone promove o *hardware* como base inicial de segurança, tendo vindo a ganhar particular relevância em soluções de sistemas embebidos devido à presença massiva dos processadores ARM no mercado. A tecnologia TrustZone separa todos os recursos de *software* e *hardware* em dois ambientes de execução diferentes, os quais são denominados de mundo seguro, onde é realizado todo o processamento seguro, e o mundo não seguro para tudo o resto. Esta tecnologia já foi alvo de bastante investigação e tem sido explorada na implementação de soluções seguras de virtualização ou até mesmo ambientes seguros de execução (TEE). Apesar de ambos os casos visarem diferentes aplicações com diferentes requisitos, ambos consistem em consolidar vários ambientes virtuais numa só plataforma e inerentemente necessitam de partilhar recursos entre os mesmos. Contudo, atualmente, os dispositivos em *system-on-chips* (SoC) habilitados com TrustZone podem somente ser configurados como seguros ou não seguros, o que significa que o conceito de duplo ambiente de execução da TrustZone não está estendido aos próprios dispositivos. Com este método de atribuição direta, ambos os mundos não podem utilizar simultaneamente o mesmo dispositivo a não ser que o mesmo seja duplicado, aumentando significativamente os custos de *hardware*. Atualmente, os métodos existentes de acesso a dispositivos partilhados em sistemas com TrustZone demonstram ter um impacto negativo no sistema em termos de segurança, desempenho e por vezes requerem um grande esforço de engenharia ou custos de *hardware* excessivos.

Esta tese propõe desenvolver o conceito de dispositivos *self-secured*, um novo método de acesso a dispositivos partilhados em sistemas com TrustZone. Estes dispositivos estendem o conceito da TrustZone à logica interna dos dispositivos, dividindo a sua lógica numa interface segura e não segura. A solução implementada foi integrada no LTZVisor, um hipervisor em código aberto e de baixo *overhead* assistido por TrustZone, demonstrando que a segurança do dispositivo partilhado é assegurada com reduzidos custos de *hardware*.

Contents

List of Figures

List of Tables

List of Listings

Glossary

ABI	Application binary interface
ACP	Accelerator Coherency Port
ACTLR	Auxiliary Control Register
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
API	Application Programming Interface
APU	Accelerated Processing Unit
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
blob	Binary Large OBject
BRAM	Block RAM
BUFG	Global buffer
CLB	Bonfigurable Logic Blocks
CPU	Central Processing Unit
DECERR	Decode error
DMA	Direct memory access
DRAM	Dynamic random-access memory
DSP	Digital signal processing
DTS	Device tree source
FF	Flip-flop
FIFO	First In, First Out
FIQ	Fast Interrupt Requests
FPGA	Field Programmable Gate Array
GIC	Generic Interrupt Controller
GPOS	General-purpose operating system
HDL	Hardware description language
I/O	Input/Output
IDE	Integrated Development Environment
IoT	Internet of Things

IP	Intelectual Propertie
IPC	Inter process communication
IPI	Inter-processor interrupt
IRQ	Interrupt Request Line
ISA	Instruction set architecture
ISR	Interrupt Status Register
KB	Kilobyte
LoC	Lines-of-Code
LTZVisor	Lightweight TrustZone-assisted Hypervisor
LUT	Look up Table
MB	Megabyte
MMU	Memory Managment Unit
NS	Non-Secure
NSW	Non-Secure World
OCM	On-chip memory
OS	Operating System
PC	Program counter
PL	Programmable logic
PLL	Phase-locked loop
PMU	Performance Monitor Unit
PS	Processing system
QSPI	Quad serial peripheral interface
RAM	Random-access memory
RO	Read only
ROM	Read only memory
RTOS	Real Time Operating System
RW	Read/Write
SCR	Secure Configuration Register
SCTLR	System Control Register
SCU	Snoop control unit
SDIO	Secure Digital Input Output
SLCR	System level control register
SLVERR	slave error
SMC	Secure Monitor Call
SoC	System on a chip
SRAM	Static Random Access Memory
SW	Secure World

TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TLB	Translation lookaside buffer
TTC	Triple Timer Counter
TXT	Intel Trusted Execution Technology
TZ	TrustZone
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller
UART	Universal asynchronous receiver-transmitter
UCB	Untrusted Computing Base
UI	User interface
VIF	Virtual Interface
VM	Virtual Machine
VMCB	Virtual Machine Control Block.
VMM	Virtual Machine Monitor
WO	Write only
XSDK	Xilinx Software Development Kit

1. Introduction

With the rise of embedded system's complexity, the demand for solutions to fulfill security and real-time requirements has been escalating [SYKS14]. For decades virtualization technology has been used by the scientific community to efficiently exploit hardware resources between multiple virtual environments [PBB13]. While the advantages of virtualization are quite clear in embedded systems where size, weight, power, and cost (SWaP-C) are important considerations, it also enforces the overall system's security [MJNH16].

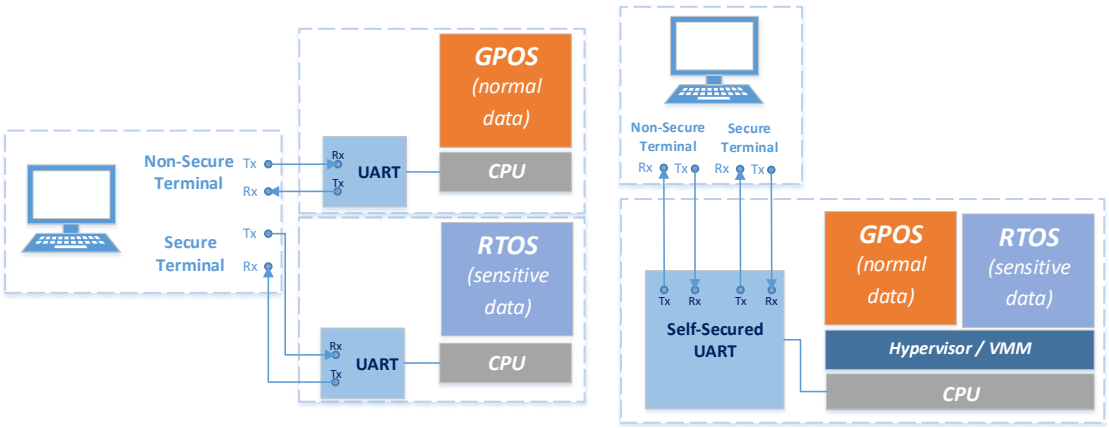
Upon the advent of the Internet of Things (IoT), security emerged even further as a significant requirement in the embedded systems development. Ensuring security in such systems is extremely crucial, as they play an important role in many mission and safety-critical systems (aviation, medical, transportation, military), and prior attacks on cyber systems have proven that can even cause physical harm [Lan11]. However, attacks against embedded systems infrastructures have been increasing because security is being misconstrued as the addition of features (e.g., cryptographic algorithms and security protocols) to the system [USK11, PMB15]. A new change in the way that systems are being developed is needed, to start guaranteeing security from the outset [OGP18].

ARM Trustzone [ARM09] and Intel TXT (Intel Trusted Execution Technology) are examples of security-oriented technologies which promote hardware as the initial root of trust. The former is gaining particular attention in the embedded space due to the massive presence of ARM processors in the market. TrustZone technology splits the hardware and software resources into two worlds - the *secure world* (SW), dedicated to the secure processing, and the *non-secure world* (NSW) for everything else. A lot of research has been done around TrustZone technology, ranging from efficient and secure virtualization solutions [FLWH10, PPG+17b, MAC+17, POP+17, PPG+17a, PTM16] to trusted execution environments (TEE) [SRSW14, PGP+17, POP+15]. Both cases, despite targeting different applications with different requirements, consolidate multiple virtual environments in the same platform and necessarily need to share resources among them. Ideally, hardware

devices should also be capable of being shared between this virtual environments. However, in TrustZone-enabled SoCs, hardware devices can only be configured as secure or non-secure, which means the dual-world concept of TrustZone is not spread to the devices itself. Consequently, with this direct assignment method, if both worlds require a certain device it needs to be completely duplicated, significantly increasing the overall hardware cost.

1.1 Motivation

Figure 1.1 illustrates a motivation example with a shared universal asynchronous receiver-transmitter (UART). In this scenario, a computer terminal requires simultaneously to communicate and exchange data with two separate operating systems: a Real-Time Operating System (GPOS), and a General purpose operating system (RTOS). Given the security-critical nature of an RTOS [YBW10], sensitive data exchanged between the computer terminal cannot be accessed or compromised in any way. With the former presented solution shown in Figure 1.1a, the computer terminal exchanges data between two separate terminal interfaces and two individual UART devices. Although this approach can satisfy the design requirements, it requires duplication of the hardware device, considerably increasing the hardware costs. In contrast, the latter solution presented in Figure 1.1b, is based on a virtualized system with device sharing capabilities. This solutions allows consolidating both operations systems in the same platform and avoids duplicating the device.



(a) Solution with device duplication. **(b)** Solution with shared device access.

Figure 1.1: Motivational example for shared device access.

Herewith, the RTOS can exchange its sensitive data through the secure terminal interface, while simultaneously allowing the GPOS also exchanging its data

through the non-secure interface, using the exact same device. With this approach, the GPOS cannot compromise the device, neither can access or compromise the sensitive data being exchanged by the RTOS.

Currently, shared device access on TrustZone-based architectures can follow different approaches:(i) Proxy task [LMH+14]; (ii) Device emulation [SVL01]; (iii) Device Para-virtualization [FLWH10, KLJ+13]; (iv) Device Para-TrustZone [Pin17]; (v) Device repartitioning [SHT12b]; and (vi) Self-virtualizing devices [RS07, WSC+07]. Among all these methods, some bring platform independence and flexibility, with an expense in the TCB size and execution overhead. Others, although presenting less execution overhead, require a considerable engineering-effort or/and hardware costs and may present limitations in the number of functionalities. Most importantly, several aforementioned approaches allow the device to be intentionally manipulated and cause failure or do not even take security into consideration, compromising the secure state of a device. Therefore, there is a need for a new shared device method that can fully and simultaneously address security while not compromising performance.

1.2 Objectives

This thesis proposes the concept of self-secured devices, a novel approach for shared device access in TrustZone-based architectures. Self-secured devices extend the TrustZone concept to the device itself by separating the device's hardware logic into a secure and non-secure interface.

Under the light of the above arguments, this thesis proposes to achieve the following goals:

- Study and in-depth analysis of Virtualization and ARM TrustZone technology;
- In-depth review and comparison of existing state-of-the-art methods;
- Familiarization with the platform and tools employed throughout this thesis development, including the Zynq7000 SoC, LTZVisor, as well as a deep understanding of the used hardware mechanisms;
- Implementation of the Self-Secured approach on devices of different complexity levels;

- Integration of the implemented solution on LTZVisor, an open-source and in-house lightweight TrustZone-assisted hypervisor, which will be deployed on the Xilinx Zybo board featuring the ARM TrustZone technology;
- Conducting quantitative studies about the hardware costs introduced by the Self-Secured approach, relatively to the device complexity level;
- Extensive evaluation of the Self-Secured devices comparatively to relevant existing state-of-the-art methods in terms of engineering effort, memory footprint, achieved speedup, and security enhancements.

1.3 Document Structure

This thesis is structured as follows:

- **Chapter 1:** The first chapter presents the motivation, goals and the structure of the thesis.
- **Chapter 2:** The second chapter is divided into two sub-chapters, the first part covers up theoretical concepts of virtualization and ARM TrustZone technology, highlighting the relevant features for this thesis implementation. The second sub-chapter, the state-of-art, describes and compares existing shared device access methods on TrustZone-based systems.
- **Chapter 3:** The third chapter describes the platform and tools. Firstly, the platform requirements are identified and the development platform chosen according to those requirements. Thereafter, the Advanced eXtensible Interface (AXI) bus protocol is discussed, given its relevance for the development process. Then, the Zybo platform and main features are laid out, highlighting the platform features that were used throughout the development process. Lastly, the LTZVisor, the hypervisor where the developed work is integrated and tested is carefully analyzed and OSes used as guest OSes are justified.
- **Chapter 4:** The fourth chapter proposes the development of the self-secured approach and the approach application to devices with different complexity level. This chapter also encapsulates the developed device drivers for managing the devices. Finally, it describes the performed modifications to the LTZVisor and hosted guests OSs, in order to integrate and test the developed devices in this hypervisor.

- **Chapter 5:** The fifth chapter addresses the evaluation. It describes the performed experiments comparing different state-of-the-art solutions.
- **Chapter 6:** The sixth chapter concludes this thesis, presenting the obtained conclusions derived from this research and tangible results, identifying the limitations, and suggesting future work towards further development on identified limitations.

2. Background, Context, and State of the Art

2.1 Background

This chapter is divided into two sub-chapters. Firstly, some background regarding virtualization, ARM TrustZone and shared device access is described. Then, the state of art, presents up-to-date methods for sharing devices on TrustZone-based systems, while comparing them among each other, considering their pros and cons.

2.1.1 Virtualization

A computer system is usually represented as consisting of several abstraction levels arranged in a hierarchy that allows the separation of concerns and ease platform independence. The lower layers are implemented in hardware and relate to the operating system and hardware platforms providing the application binary interface (ABI) and instruction set architecture (ISA), interfaces that applications depend on to run. Virtualization technology allows conceiving multiple emulated environments from a single, physical hardware system. This layers can be virtualized and their available resources and interfaces are simulated and mapped onto the interface and resources of the real system [SN05]. This concept of virtualization can be applied not only to subsystems but to an entire machine. By adding a software layer to the real system to support the desired architecture it appears as an emulated different machine, or even a set of multiple machines, duplicates of the original one. These machines, so-called virtual machines can be classified as: Process VMs, providing a virtual ABI or API environment for user applications delivering replication, emulation, and optimization; or System VMs [Hei08], the relevant type for this thesis context, providing a complete environment in which a single-host hardware platform is partitioned into several virtual

machines supporting the concurrent execution of multiple, isolated guest operating system simultaneously. In Figure 2.1, it is demonstrated the basic software stack for system virtualization. The Virtual Machine Monitor (VMM) or hypervisor component is the software layer that provides the VM environment. The hypervisor normally runs with full privileges, having access to, and managing all the hardware resources. While the guest operating system and its application processes run with lower privilege and are managed under control of the VMM, which in some cases can verify and perform privileged operations in behalf of the unprivileged guest, upon a request for shared hardware resources.

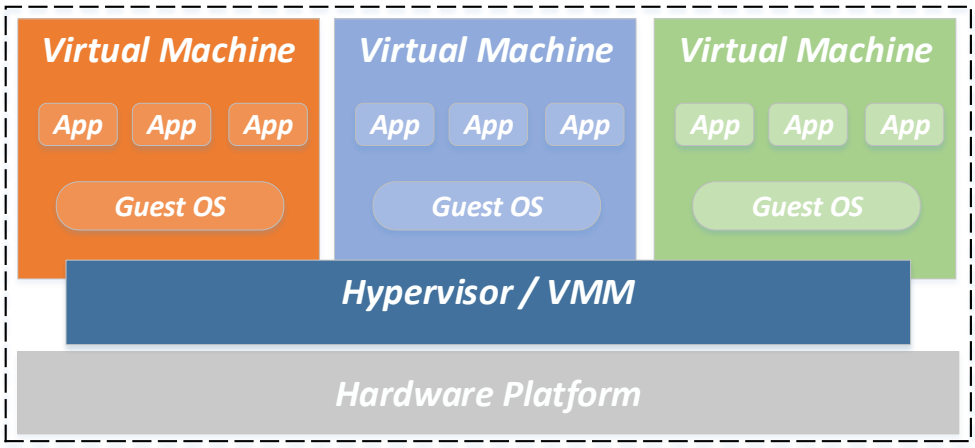


Figure 2.1: System Virtualization Stack.

The classical Popek and Goldberg virtualization requirements [PG74], derived under simplifying assumptions, identify the three essential properties of virtual machines:

- **Equivalence**: The virtualized environment running under the VMM should have identical behavior to that demonstrated when running on the original machine. This also minimizes the engineering effort of porting guest software to the VM.
- **Resource Control**: It must be impossible to an arbitrary guest to affect other guests resources. The VMM must be in complete control of all the virtualized resources.
- **Efficiency**: All harmless instructions must be executed by the hardware directly without VMM intervention, minimizing the overhead mechanisms of the VMM. Guest software must show none or only a slight deterioration from their native performance.

In the Popek and Goldberg's terminology, a VMM must present all of the above mentioned properties. This guarantees an acceptable performance without considerable deterioration, mutual isolation among guests while minimizing porting efforts.

2.1.1.1 Classical Software Virtualization Techniques

To achieve the necessary technical requirements for virtualization, the target processor must have several modes with different privilege levels that implicate different access rights to the system resources, also known as protective rings in a ring model, illustrated in Figure 2.2. For instance, the well known x86 architecture has four execution modes (or four rings). Typically, the OS running in a physical machine is executed in kernel-mode and the user applications in user-mode. The OS kernel runs with the highest privilege and has access to the full set of instructions (privileged and unprivileged) of the physical processor. Differently, applications running in the unprivileged user mode have no direct access to the privileged instructions, [RHFN+12]. The hypervisor must run on an additional privileged mode, in order to be protected from the guest OSs, and guest OSs from their applications, guaranteeing the second property of virtual machines. When a guest OS executes a privileged instruction and does not have the required privilege it creates traps, also known as processor exceptions, which are events that turn control over to the privileged software (hypervisor) by changing the processor mode and setting the program counter (PC) to a known entry point. This extra layer that has an extra privileged mode is provided by CPU extensions for virtualization support. When this extra layer is not provided, implementations often recur to a technique named ring decompression or ring deprivileging [UNR+05]. In such approach, the hypervisor is placed in kernel mode and both the OS kernel and user code are pushed back to the unprivileged mode. However, in many processors, only two privilege levels are provided. In that scenario, through a simple technique called trap-and-emulate, an extra virtual privileged level is created, subdividing the OS and user code into two separate privileged layers. Upon the occurrence of a trap, the hypervisor checks the current state of the VM: if it is running on kernel mode it performs the required action, otherwise, it emulates a trap in the VM by changing its state and forwarding the exception to the guest OS. Therefore, both ring compression and trap-and-emulate techniques incur considerable performance costs due to every crossing of modes or execution of a sensitive instruction having to go through the hypervisor, breaking one of the essential properties of virtual machines [PG74].

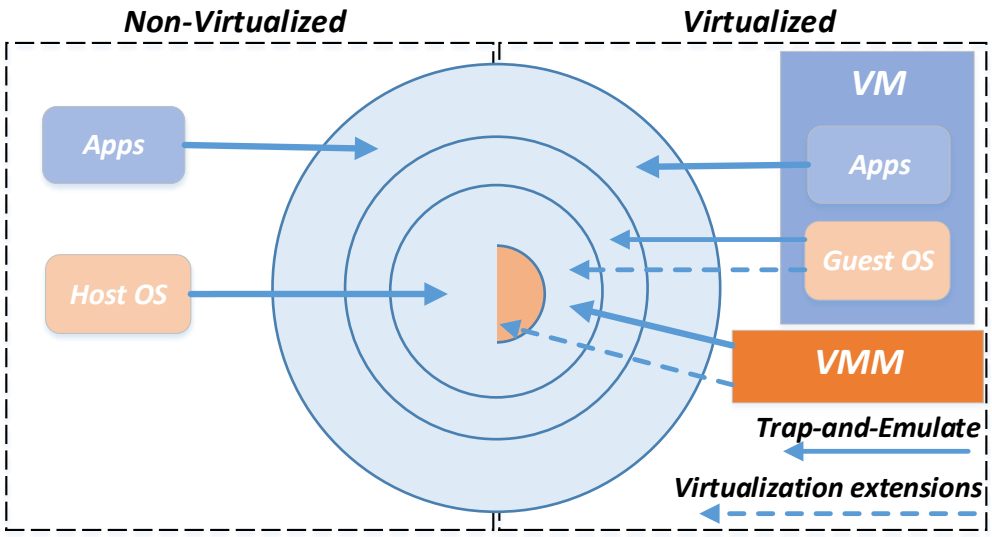


Figure 2.2: CPU protection ring levels. Adapted from [RHFN+12].

Popek and Goldberg [PG74], introduced a classification of instructions of an ISA into different groups, according to their behavior when executed in different processor modes:

- **Privileged**: Instructions that trap if the machine is in user mode, and do not trap if it is in kernel mode.
- **Control sensitive**: Instructions that attempt to change the configuration of resources in the system.
- **Behavior sensitive**: Instructions whose behavior or result depends on the configuration of the system's resources.
- **Innocuous**: Instructions which are not sensitive, do not require a privileged processor mode to run, and cannot change hardware resources context.

That said, to apply classic virtualization to an architecture all sensitive instructions in its ISA must be privileged. The rest of the sensitive instructions which are not privileged are considered critical instructions. These requirements, guarantee the resource control property, above mentioned, by running the hypervisor with a higher privilege level than the guests. Consequently, in a trap-and-emulate approach, depicted in Figure 2.3, whenever a guest executes a sensitive instruction (in user mode), the following actions are performed in the stated order: the hypervisor takes full control of that action; the instruction is decoded; the trap originating instruction is interpreted; the instruction is emulated; the VM state is updated.

To comply with the efficiency property, most of the hosted virtual machines ISA instruction should be identical to the physical underlying hardware in order to be considered innocuous, consequently enabling their execution directly by the hardware.

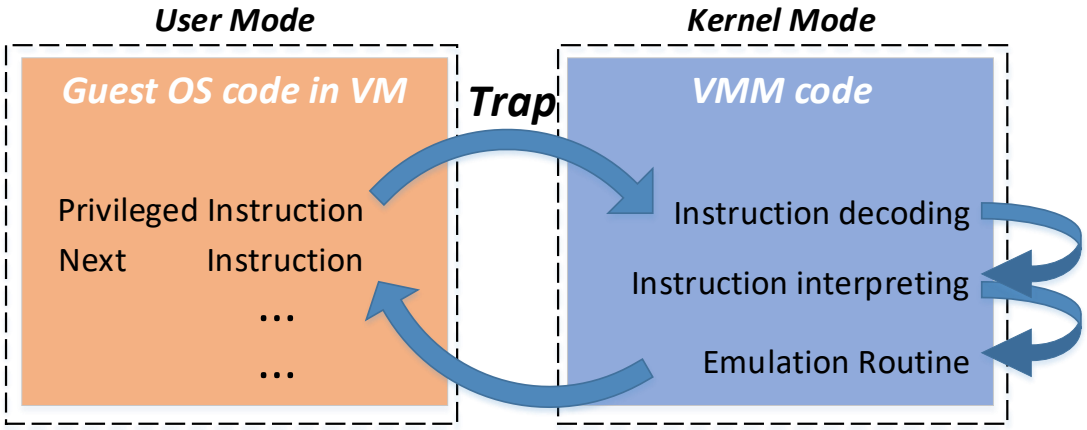


Figure 2.3: Trap and emulate technique.

2.1.1.2 Para-Virtualization

Until this point, the term virtualization has been used to refer to the classic virtualization, known as full-virtualization [RHFN+12]. With full-virtualization guest run unmodified and not aware of their virtualization, which may cause performance deterioration for not being able to easily take advantage of virtualization features. Nonetheless, full-virtualization has to possess the strict requirements of classic virtualization (previously described) or hardware virtualization support. Para-virtualization [Chi07, Kai09, VMw06, WSG02] offers potential performance benefits, a consequence of modifying the guest by exploiting its virtualization awareness. Modifications to the guest OS code consist in adding a special set of instructions (named hypercalls) for the execution of critical instructions, replacing instructions of the real machine's ISA. These hypercalls are conceptually equal to a system call. Figure 2.4 shows the difference between system calls and hypercalls and the ring transitions when a system call from an application is issued. An extra transition between layers allows applications to run without modification with the cost of a small speed penalty.

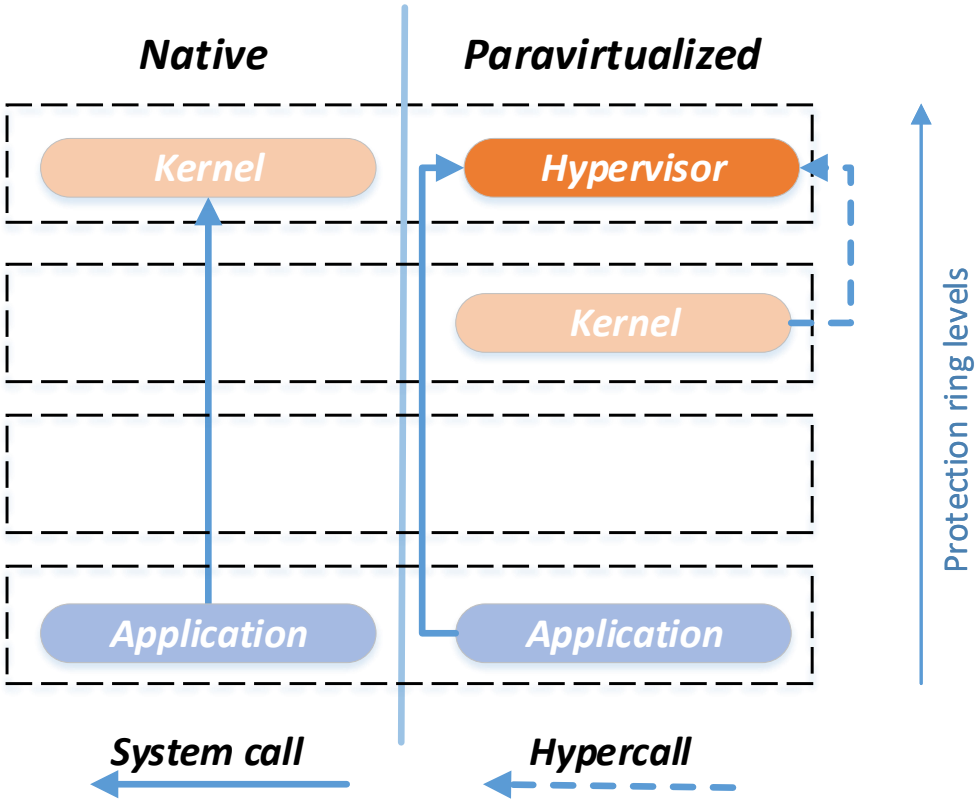


Figure 2.4: System calls in native and para-virtualized systems. Adapted from [Chi07].

A single hypercall can replace a set of many sensitive instructions [Chi07], reducing the frequency of switches between modes and consequently the overhead incurred by the decoding of those instructions and hardware emulation. Overall, para-virtualization can provide a solution for non-virtualizable architectures that do not provide any hardware virtualization support and performance enhancements. Also, due to guest awareness, efficient communication and synchronization mechanisms (IPC) among guests can be implemented. One of the biggest bottlenecks in many full-virtualized systems is device emulation. Device para-virtualization, later scrutinized, replaces the device driver for the emulated device with a front-end device driver removing the emulation associated overhead. Despite all advantages, para-virtualization implicates high engineering effort and can break the resource control property of virtual machines, violating guest isolation.

2.1.1.3 Hardware-Assisted Virtualization

All of the virtualization methods formerly referred are still too inefficient due to the high overhead associated with processor exception mechanisms and context save and restore operations necessary whenever the hypervisor takes control.

At some point, different manufacturers have extended the architecture in different ways to overcome the overhead introduced by these mechanisms and decrease VMM complexity, providing virtualization hardware support extensions to aid virtualization adding a set of instructions that makes virtualization considerably easier. These extensions target a wide range of architectures, such as server, desktop architectures (Intel's VT-x [UNR+05]) and embedded architectures as AMR's VE [VH11] or Imagination Technologies' MIPS VZ [ZMH15]. Conceptually, virtualization extensions can be thought as adding a new privilege processor mode, in which, the hypervisor is expected to run and can trap-and-emulate operations that previously, would have failed silently. This added mode, also allows the OS to stay at the same level it expects to be without virtualization and catching attempts to access the hardware directly. However, these extensions also provide other different features, such as replicating and multiplexing important hardware configuration registers, two-level address translation, and virtual interrupt support. In the context of this thesis, is also worth mentioning ARM TrustZone hardware extensions. Even though ARM TrustZone is a security extension and not a virtualization extension, some of the provided features are very similar, for instance, much of the critical hardware is replicated and also adds an extra privileged mode. ARM TrustZone is widely spread and available in low-end and mid-end range microprocessors, contrariwise to virtualization hardware extensions. Hence, it has been exploited to also enable embedded virtualization [FLWH10, HGX+17]. Compared to para-virtualization, hardware assisted virtualization allows running unmodified OSs. However, given that the guest is not aware that is running in a virtual environment it cannot take easy advantage of virtualization features which makes it more likely to be slower. Nevertheless, a hybrid approach might be advantageous, for instance, taking advantage of hardware assisted faster system calls and support for nested page tables, and para-virtualization better I/O performance due to its lightweight interfaces to devices.

2.1.1.4 Hypervisor Architectures

Based on the location of the virtualization layer in the system stack and on the permission of the VMM accesses to the hardware resources, hypervisor topologies [RHFN+12, SGB+16], despited in Figure 2.5, can be categorized as:

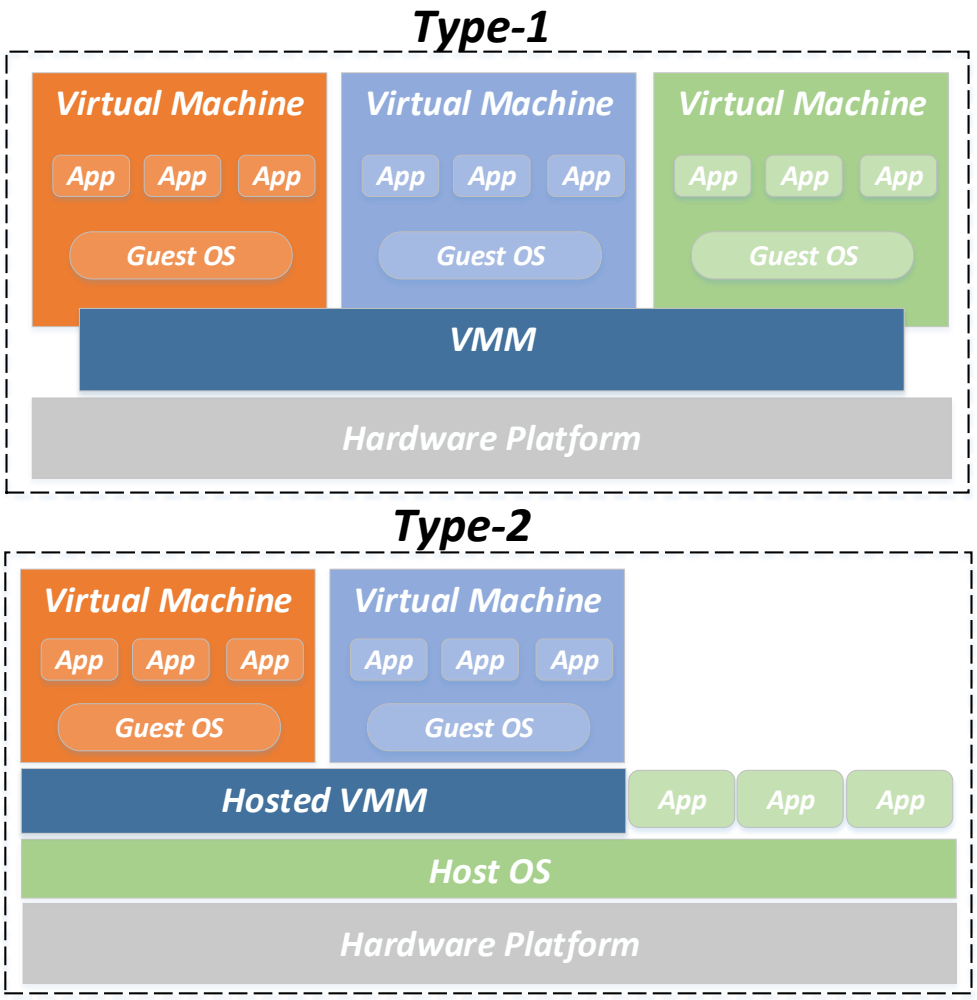


Figure 2.5: Virtualization Topologies.

- **Type-1**, or bare-metal Hypervisors, run in a higher privileged mode than the guest OSs, directly over the hardware where it can manage and access every hardware resource of the system.
- **Type-2**, or hosted Hypervisors, run in an unprivileged mode over the privileged host OS that is already executing, with no direct permissions to the hardware.

Bare-metal hypervisors also referred to as native virtualization, are more suitable for time-critical systems due to the performance degradation of guest OSs only depending on the hypervisor performance. In addition, hosted hypervisors lack of privilege and security is not adequate for embedded devices where critical applications will run, and usually exhibit a poorer performance compared to type-1 hypervisors.

2.1.2 ARM TrustZone

TrustZone technology [ARM09, ARM16] refers to security extensions available in all ARM Application-processors (Cortex-A) for several years, since the ARMv6 architecture. This hardware security extension splits all hardware resources and virtualizes a physical core into two virtual cores, providing two completely separated execution environments: the *secure* and the *non-secure* worlds, as illustrated in Figure 2.6. Hardware mechanisms are provided, to ensure secure world resources are not accessible by the non-secure world, while the secure world can access any resource. This strong isolation is important in scenarios where a trusted OS with a small TCB, executing time-critical and security-critical applications, runs in the secure side, alongside a rich untrusted GPOS running in the non-secure world. With the addition of a new architectural feature at the processor level, the 33rd bit, also referred as the NS (Non-Secure) bit, provides separation between these two worlds, indicating in which world the processor is currently executing. This bit is accessible through the added Secure Configuration Register (SCR) present in the System Control Co-processor (CP15) and exclusively accessible by the secure world. Some of the System Control Co-processor (CP15) registers and other critical processor bits are banked in both worlds. The remaining registers which are not duplicated, are either non-accessible by the non-secure world or kept under close supervision of the secure world.

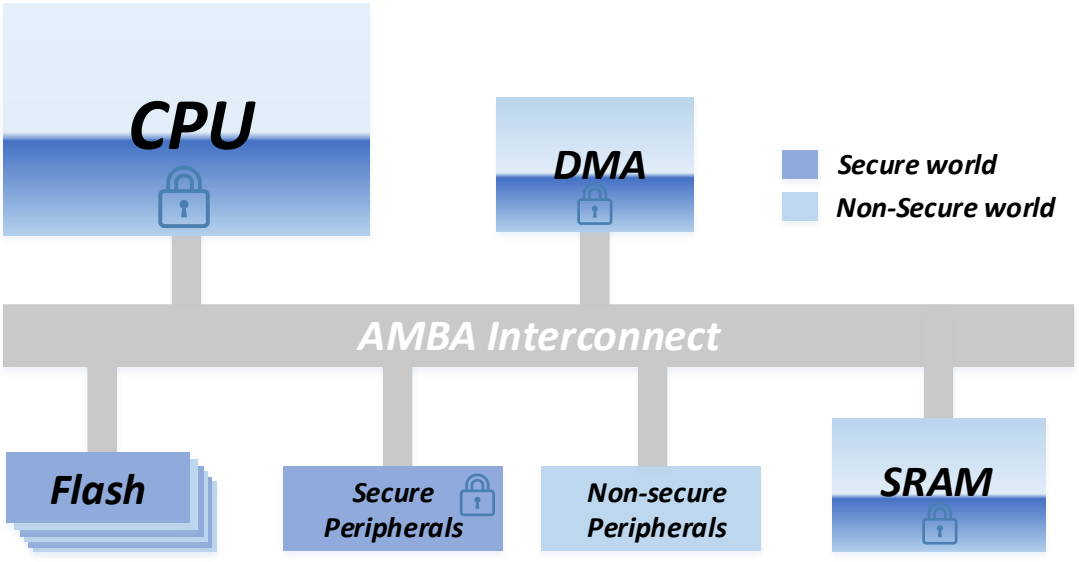


Figure 2.6: Arm TrustZone hardware Architecture. Adapted from [ARM15].

TrustZone adds a special new secure processor mode called *monitor mode*. This

new mode is used for bridging transactions between both worlds, while preserving the processor state. Unlike other processor modes, monitor mode is only present in the secure world, hence always considered secure. A new privileged instruction was also specified, SMC (*Secure Monitor Call*) analogous to system calls, through this instruction the non-secure world is able to enter monitor mode. The monitor mode can also be enabled by configuring it to handle interrupts (IRQs, FIQs) and exceptions in the secure side.

Secure and non-secure world partitioning is not only restricted to the processor, but also propagated to other system resources such as memory, peripherals and buses. Memory infrastructure can also be partitioned into distinct memory regions, which can be configured to be used by both worlds or exclusively by the secure world. If the non-secure world tries to access the secure address space, an abort routed to the monitor mode is triggered. The processor also provides two virtual Memory Management Units (MMUs), delivering separate virtual-to-physical memory address translation tables to each world. Even though the NS bit is still available in the non-secure side, from the non-secure world perspective is transparent, since accesses are always performed with with NS set. This memory isolation is extended and still available at cache level, with the NS bit tagging each entry with the processor state upon the access. At the cache-level, entries from both worlds can coexist removing the need for duplication and cache flushing, consequently accelerating world switching.

To provide the aforementioned memory infrastructure isolation, TrustZone features hardware peripherals such as the TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA). The TZASC hardware controller provides a programming interface, only accessible by the secure world, that can configure specific memory regions of the DRAM, after being partitioned into different memory segments, whose granularity depends on the SoC implementation. If the TZASC configures one of these memory regions as secure, non-secure attempts to access it will be denied. TZMA provides the same functionalities, but targeted at the on-chip memory, such as ROM or SRAM. However, TZMA cannot be used for partitioning dynamic memories or memories that require multiple secure regions, unlike TZASC. Additionally, through TrustZone Protection Controller (TZPC), which is a configurable signal control block placed on Advanced Peripheral Bus (APB), system devices can be dynamically configured as secure or non-secure. Moreover, to avoid overloading the processor, direct memory access (DMA) controller can be used for moving data around physical memory. TrustZone is also extended to this engine, featuring both a secure and

non-secure concurrent channels with independent interrupts and controlled by a dedicated APB interface. TZASC, TZMA and TZPC components are all optional and implementation-specific.

The TrustZone-enabled AMBA Advanced eXtensible Interface (AXI) system bus, carries extra control signals to restrict access on the main system bus, including an additional control bit, the non-secure bit, for each of the read and write channels on the main system interconnect. This enables TrustZone architecture to also secure peripherals (e.g. interrupt controllers, timers, and user I/O devices) through this additional non-secure bit.

To support the robust management of secure and non-secure interrupts, the Generic Interrupt Controller (GIC) provides both secure and non-secure prioritized interrupt sources. Through the Interrupt Security Register of the GIC an interrupt can be configured as either secure or non-secure. Interrupt prioritization is available, allowing to configure secure interrupts as higher priory than the non-secure interrupts. An important feature that allows secure interrupts to be handled with higher priority than non-secure interrupts, preventing potential denial-of-service attacks. Among other possible interrupt models the GIC allows to configure FIQs as secure and IRQs as non-secure interrupt sources, as suggested by ARM and adopted by LTZVisor.

2.1.3 TrustZone-assisted Virtualization

Even though ARM TrustZone hardware extension is security-oriented, some of its features are very familiar to other hardware-assisted virtualization extensions. In particular, the existence of an extra higher privileged mode (monitor mode) where the hypervisor can run; the ability to have full control over the exception system; and being able to execute OSs in the other remaining processor modes. However, classical hardware virtualization depends on two-level address translation which, in fact, is not provided by TrustZone. Instead, the TZASC enables memory segmentation for isolation. Nevertheless, unmodified guest OSes need to cooperate in order to be executed in the respective preassigned segments achieving memory isolation. TrustZone can be efficiently exploited to assist virtualization with the enormous advantage of being widely spread to most low-end and mid-range microprocessors used in embedded devices. The segmented memory model should not impose any problem, given the reduced and fixed number of VMs normally deployed in embedded use-cases.

TrustZone-assisted virtualization can support systems with different OS configurations [PS18], such as single-guest, dual-guest, and multi-guest systems. The

simplest architectures use a single-guest setup [FLWH10], in which the single guest OS and its applications run in the non-secure side and the hypervisor in monitor mode. The hypervisor has full access to the whole system, composing the entire system's TCB, and has the responsibility of keeping the secure resources under its close supervision. In contrast, the non-secure guest may only manage and access resources (i.e. devices, memory, and interrupts) configured as non-secure. However, if the non-secure guest requires accessing secure resources, it is able to perform those accesses through para-virtualized drivers, under the hypervisor's supervision.

Most of the existing solutions commonly implement a dual-OS configuration in order to have one processor virtual state (non-secure and secure state) dedicated to each guest OS. In this manner, the hypervisor also runs in the monitor mode, while each guest OS runs individually in each world. This type of configuration is ideal in scenarios where an RTOS with real-time functionalities and requirements runs in the secure world, isolated from the GPOS, which runs in the non-secure world, usually during the RTOS idle periods to ensure timing requirements are meet. The LTZVisor [PPG+17b], is an example of an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems, and supports the coexistence of two OSs, one secure RTOS side by side with an untrusted, rich GPOS.

The number of supported guest OSs on TrustZone-enabled systems has been a setback for perceiving TrustZone has a viable virtualization solution. However, recent solutions [MAC+17, PPG+17a] were able to demonstrate how TrustZone-enabled platforms are capable of hosting several guest OSs without compromising isolation between guest OSs, neither interfering with their proper execution, by attentively managing shared resources and the security configurations of memory, devices, and interrupts at runtime.

2.2 Related Work

Currently shared device access can essentially follow the approaches: (i) Proxy task; (ii) Device emulation; (iii) Device Dara-virtualization; (iv) Device Para-TrustZone; (v) Device re-partitioning; (vi) Self-virtualizing devices. In this section, the existing shared access methods will be thoroughly analyzed and compared.

2.2.1 Devices Access in TrustZone

When multiple virtual environments are consolidated in the same platform there is a need to share resources among them. Ideally, hardware devices should also be capable of being shared between these virtual environments. However, in TrustZone-enabled SoCs, hardware devices can only be configured as secure or non-secure, which means the dual-world concept of TrustZone is not extended to the devices itself. So, if both worlds require a certain device, it needs to be completely duplicated. Likewise, typically in dual-OSs systems, devices are usually duplicated and assigned exclusively to each guest OS: devices that are critical for the reliability of the system are assigned to the RTOS and the remainder devices are assigned to the GPOS. This method, denominated direct assignment or pass-through access only allows the non-secure world to access devices configured as non-secure and the secure world to access devices configured as secure, as illustrated in Figure 2.7.

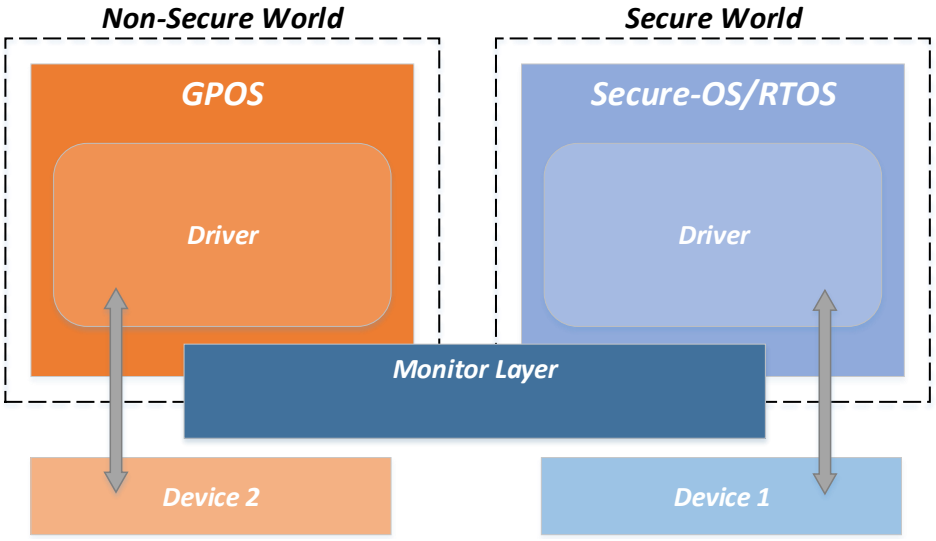


Figure 2.7: Direct assignment access method.

When both OSes need to make use of the exact same device it becomes necessary duplicating the device, which is useful for ensuring the RTOS reliability, and maximizing the system performance. However, it also adds a significant increase in the total hardware cost.

For achieving a reliable device sharing mechanism the following requirements must be satisfied:

- **Real-time:** Timing requirements are really critical for a RTOS. Device sharing mechanisms must ensure that the SW has full control over the shared

device and the successful completion of its operations. Meaning that an operation performed by SW must be completed, regardless of operation requests coming from the NSW. Any action from the NSW that may prevent the SW use of the shared device for an unbounded amount of time should not be possible.

- **Security:** Secure resources and logic must be protected against both malicious and accidental accesses from the non-secure world. The shared device cannot be, in any way, compromised by the NSW.
- **Overhead:** The introduced overhead of a device sharing mechanism (e.g., due to data copies, data exchange, access policies or context switches) must be minimized. The device's native performance should not be affected by the method application and introduce performance degradation.
- **Device latency:** Access to the device when a valid request is issued must be performed as fast as possible.
- **Modifications:** Modifications to the TCB software and monitor must be minimized or ideally, avoided. Otherwise, it could incur overhead and require considerable engineering effort. Moreover, device drivers should be generic and should not require modifications, regardless of the OS that is using them.
- **Hardware Costs:** Hardware modifications to the device's logic must also be minimized, to prevent a significant increase in terms of hardware costs when comparing with the native device.

2.2.2 Proxy Task

Proxy Task [LMH+14] illustrated in Figure 2.8, is the most basic shared device method, which consists in a SW OS client task that can send a request to a proxy task in the NSW OS, through a communication channel [SHT12a, OMC+18], in order to take advantage of the GPOS libraries and drivers richness. The introduced overhead is fairly low due to the high level of abstraction of the request. Despite this method's performance being ideal, it is completely unreliable. The SW OS requests might be ignored or act differently than expected due to the GPOS software being untrusted, excluding this method as a viable secure shared device access approach.

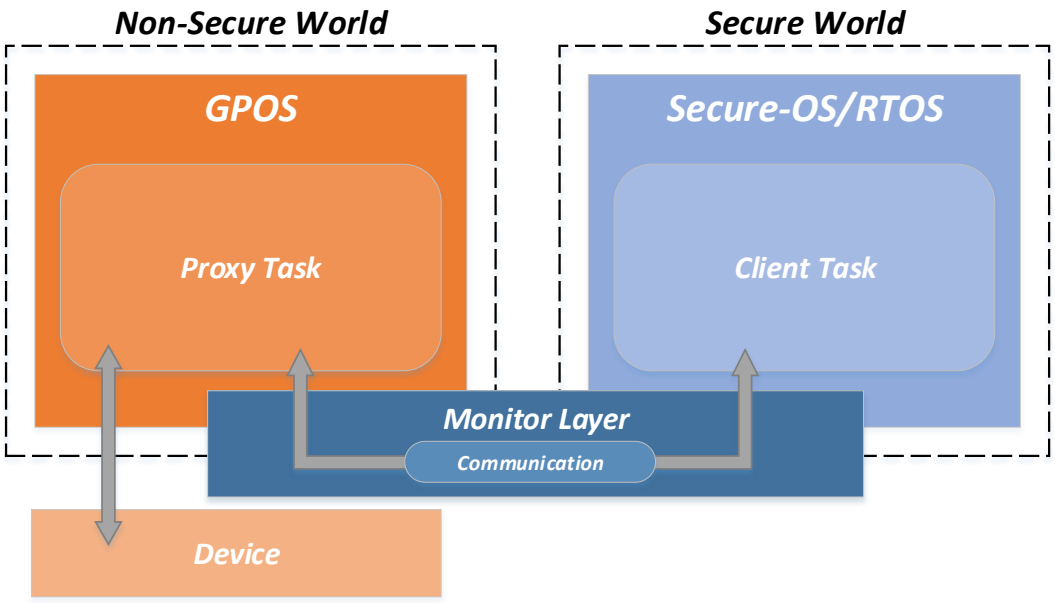


Figure 2.8: Proxy Task method.

2.2.3 Device Emulation

Device emulation, depicted in Figure 2.9, is typically used in full-virtualization solutions [SVL01] and follows the classical Popek and Goldberg's trap-and-emulate approach for virtualization [PG74]. From the GPOS point of view, it owns all the devices, tricked to think there is a provided legacy driver. However, when the GPOS tries to make any access to the virtual device, it results in being trapped by the monitor layer. After being trapped, the hypervisor is responsible for emulating the functionality that the GPOS was intended to perform on the device. Once the GPOS access is trapped in the monitor layer where security and access permissions issues will be accounted for, it is forwarded to the SW OS (RTOS), where a driver will handle the physical device. This method delivers the GPOS platform independence and flexibility and does not require any changes to guest OSs. Nonetheless, it comes associated with significant execution overhead and TCB size expense, due to the required complex extensions in order to implement the trap mechanism. Moreover, traps are typically delivered to the SW OS as interrupts, and the interrupt rate must be limited in order to prevent frequent accesses from the GPOS to the device, that may result in a performance bottleneck.

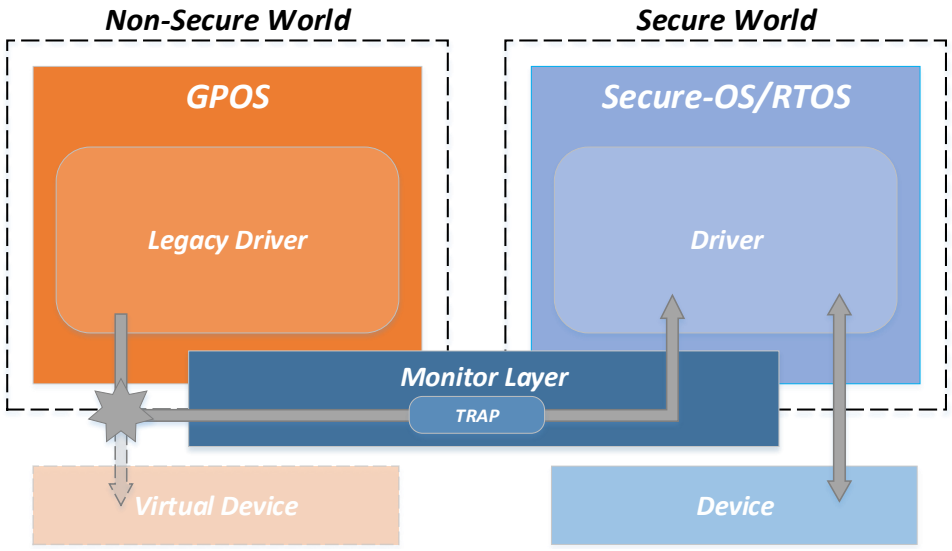


Figure 2.9: Device emulation method.

As shown in Figure 2.10, the NSW OS can trigger external aborts, which are exceptions generated by accessing secure or invalid device memory. Such aborts trap directly in the monitor if the secure world OS has this feature enabled. This mechanism could be used to emulate access to devices.

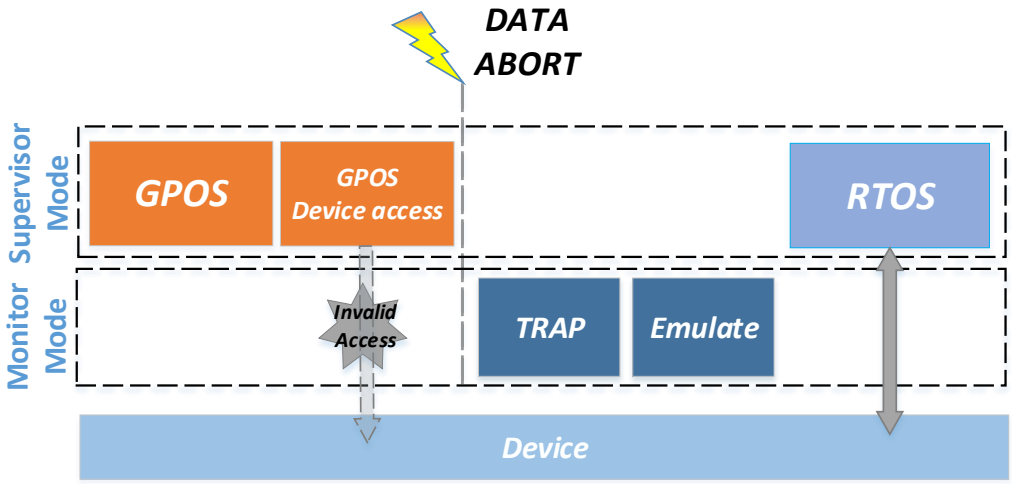


Figure 2.10: Ideal device emulation flow control.

However, using these trap-and-emulate techniques is not possible, due to TrustZone protection mechanisms [Kal14]. Consequently, device emulation on TrustZone-based systems cannot be implemented, as a result of most sensitive instruction in the non-secure world not having the necessary security privilege to be detected by the hypervisor. Even so, for those that manage to get detected, such as load/store instruction to access secure device registers, the CPU will not immediately enter

the hypervisor when the fault occurs. Instead, the bus transaction will still be attempted, triggering an external data abort, similar to a device interrupt. Even though the violation can be detected and can generate an exception, this exception is imprecise and not always immediate. This leaves no chance to reconstruct what happened in between the invalid access and the reception of the external abort exception in the hypervisor, neither can the hypervisor restore the non-secure world to a useful state.

2.2.4 Device Para-Virtualization

The para-virtualization approach [FLWH10, KLJ+13, Chi07] shown in Figure 2.11, follows the para-virtualization concept, consists in slight modifying the GPOS driver (i.e., front-end driver) to send requests to the SW OS driver (back-end), through hardware-like interfaces that exchange data between NSW and SW, like a communication channel [SHT12a, OMC+18].

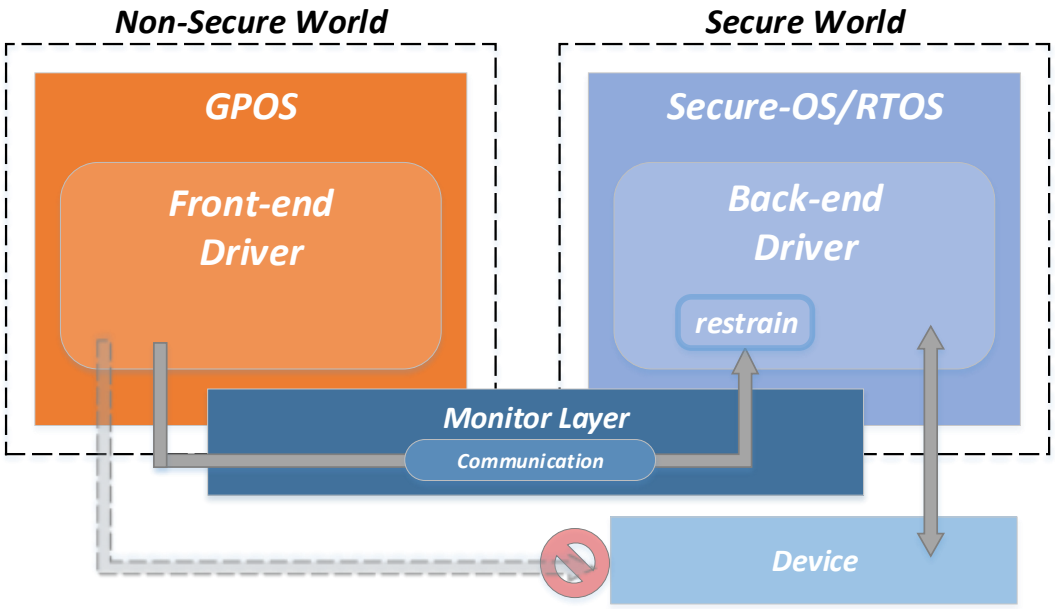


Figure 2.11: Device Para-Virtualization method.

Both the GPOS and the SW OS (normally a RTOS) are modified in order to support devices allocated to the secure world. If a device is configured as non-secure both the NSW and SW can access it directly without requiring SW intervention except for the interrupt delivery. However, if the device is configured as secure, an access driver (i.e., front-end driver) that sends requests to the secure world is required. The SW request will be interpreted and the correspondent action executed or mediated by the secure driver (back-end). The request from

the NSW is validated and verified according to the specified security policy. Access to a secure device is not as fast as access to non-secure devices because it evolves communication between NSW OS and SW OS. This access generally involves the following steps:

- The access driver in NSW issues a request to the SW;
- The monitor saves the NSW processor state and sends a message to the SW;
- The SW selects the device and handles the request;
- After finishing the request the SW sends a reply message to the monitor;
- The hypervisor restores NSW processor state and initiates the NSW entry.

This method presents less hardware costs relatively to device duplication, less execution overhead than emulation, and protects shared devices from potential bugs which might arise from software modifications or extensions, unlike re-partitioning. Although, still requires a considerable engineering-effort like modifying the OSs (the effort required to modify a kernel can be high), presents limitations in the number of functionalities (the GPOS is limited to the functionality supported by the RTOS driver) and its performance is still far from native. Also, this approach comes with associated overhead, which might cause real-time performance issues on the trusted domain, and also a considerable increase in the TCB complexity.

2.2.5 Device Para-TrustZone

The para-TrustZone approach [Pin17] illustrated in Figure 2.14, is based on the para-virtualization approach, which consists in slight modifying the GPOS driver to send requests for the secure device. However, with this approach, instead of sending the request over to the SW OS, the requests are directly sent to the hypervisor itself, which handles the requests and carries them out. It is implemented through the added TrustZone privileged instruction SMC (*Secure Monitor Call*), which enables the non-secure world entering monitor mode. The SMC instruction requires kernel privilege to be executed, hence the GPOS driver must be modified to add the supported instructions.

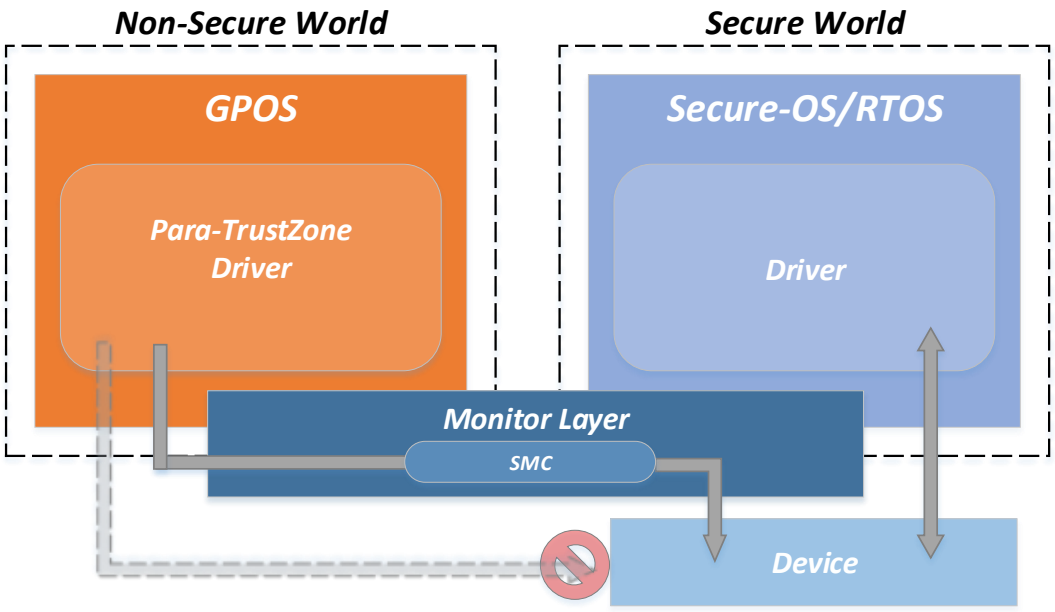


Figure 2.12: Device Para-TrustZone method.

This method can out-perform the para-virtualization by removing the overhead incurred by the communication mechanisms and OSs context switches. However, a considerable engineering-effort is still required due to the required modifications at the OS and hypervisor level. The number of functionalities is limited to the ones supported by the SMC handler within the hypervisor. Most importantly, this method has security issues, due to not protecting the device against the GPOS misbehavior and not limiting the performed accesses in any way. Consequently, once the GPOS is running it can perform massive SMCs, intentionally causing device failure.

2.2.6 Device re-Partitiong

Figure 2.13 depicts the re-partitioning approach [SHT12b], implemented in SafeG, with this approach a device that has been already assigned to a certain OS can be dynamically re-assigned at run-time. Devices can be configured and re-configured as part of the secure or non-secure world through the TrustZone Protection Controller (TZPC). This re-assignment occurs with a trigger condition from the SW OS.

As a result, devices can be directly accessed by both the SW and NSW considerably reducing overhead. Additionally, the monitor requires few or even none modifications at all. The re-partitioning approach can be implemented in a *pure* and *hybrid* form. A module named Re-partition manager present in both OSs

is responsible for managing the device sharing through a communication channel [SHT12a]. The SW OS Re-partition manager is activated whenever a condition is triggered.

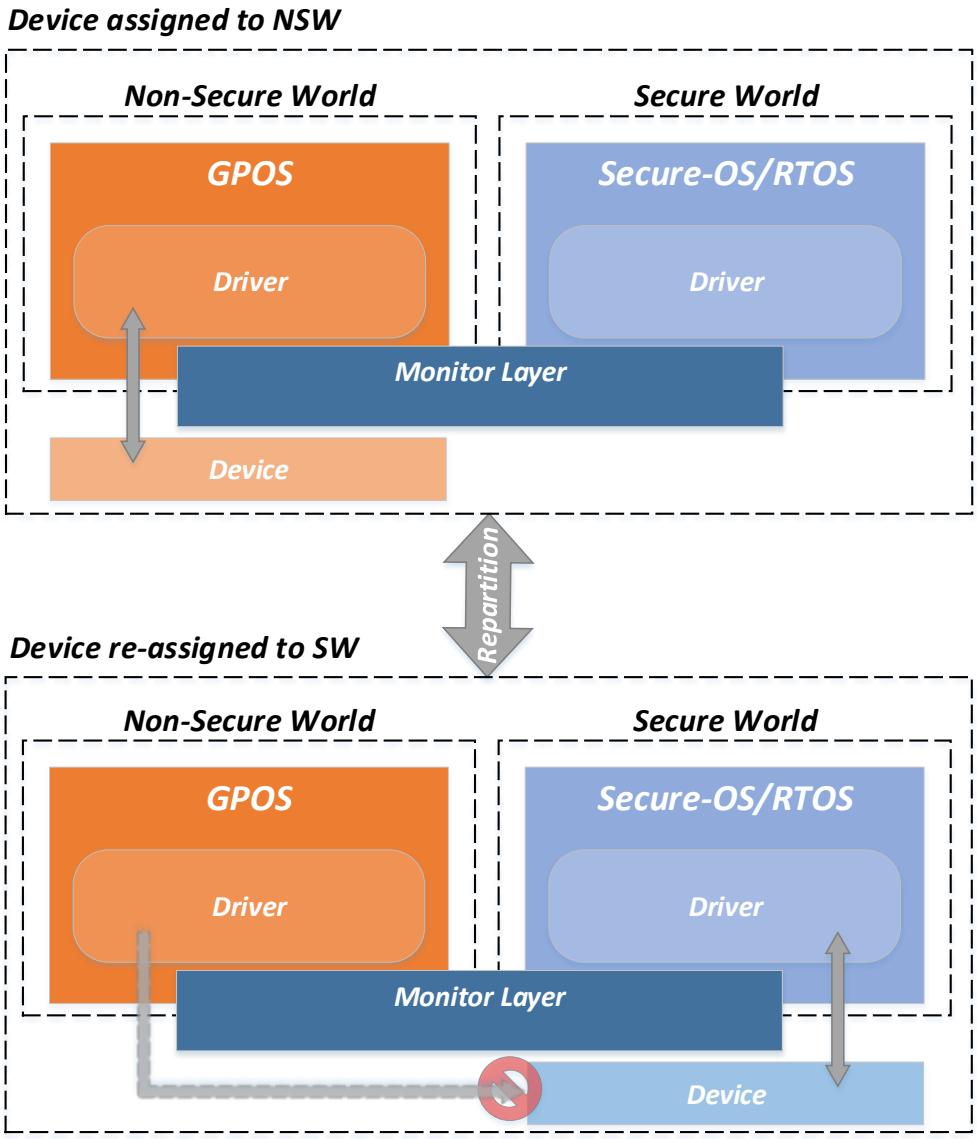


Figure 2.13: Device Re-partitioning method.

In the pure form, when a device must be re-partitioned to the TCB, the RTOS sends an "UNPLUG" event to the GPOS manager. To guarantee that the SW OS usage of the shared device cannot be compromised and is reliable, the RTOS manager is completely independent of the GPOS manager state. Following the trigger event, the RTOS manager needs to perform a full reset on the device to a predefined state, and then the RTOS re-partition manager configures the device as part of the TCB, finishing the re-partition process and making the device now able to be reliably used by the RTOS. On the other hand, when the RTOS no longer

needs the device, it must re-partition the device back to the UCB, the RTOS manager flushes all the secure and sensitive data from the device, and configures it as part of the UCB. Then, a "PLUG" event must be sent to the GPOS, where the respective manager will be responsible for restoring the device state and restoring the processes that were previously stopped by the "UNPLUG" event, sent when the RTOS needed the device. Despite the pure mechanism allows maximizing performance with direct access to the device, it is necessary to perform a full reset on the device which incurs a considerable device latency that may not satisfy time-critical applications needs.

In behalf of the incurred high device latency, an alternative hybrid repartitioning method was introduced based on the para-virtualization approach. With this method, the interface of the shared device is split into two separate interfaces: an interface with the operations that are performed at boot time and another with the remaining operations performed at run-time. In order to reduce the resetting time of the device, the initialization interface is only accessible by the RTOS guaranteeing that certain time-critical conditions are satisfied. This avoids a full reset when the device is being re-partitioned to the TCB. By denying the GPOS access to these device initialization operations, a shorter device latency is achieved. When the GPOS wants to access the run-time interface a monitor call must be performed (similarly to para-virtualization). Even though, this approach (if entirely implemented in SW) still requires some changes to the monitor layer and introduces overhead. As a consequence, the choice lies with a trade-off between high performance or lower device latency. Despite all benefits of both mechanism, these solutions still present a huge setback: security. Once the device is assigned to the GPOS, the GPOS has complete access to the device, meaning that if it is compromised, the device can be intentionally manipulated to cause a failure. This mechanism only reclaims access permission from the GPOS when the RTOS requires the device. Thus, the shared device is not protected against the SW misbehavior or faults, and incapable to prevent device failure caused by GPOS accesses (e.g. massive requests from the GPOS).

2.2.7 Self-virtualizing

A self-virtualizing device [RS07, WSC+07] has additional computational resources that support I/O virtualization functionalities. With these resources the device is capable of: multiplexing/demultiplexing a large number of virtual devices mapped to a single physical device; managing virtual devices through an API on the hypervisor; using APIs for accessing virtual devices and interacting with guest

domains and taking maximum advantage of the computing power of the hardware platform (e.g., multiple processing cores).

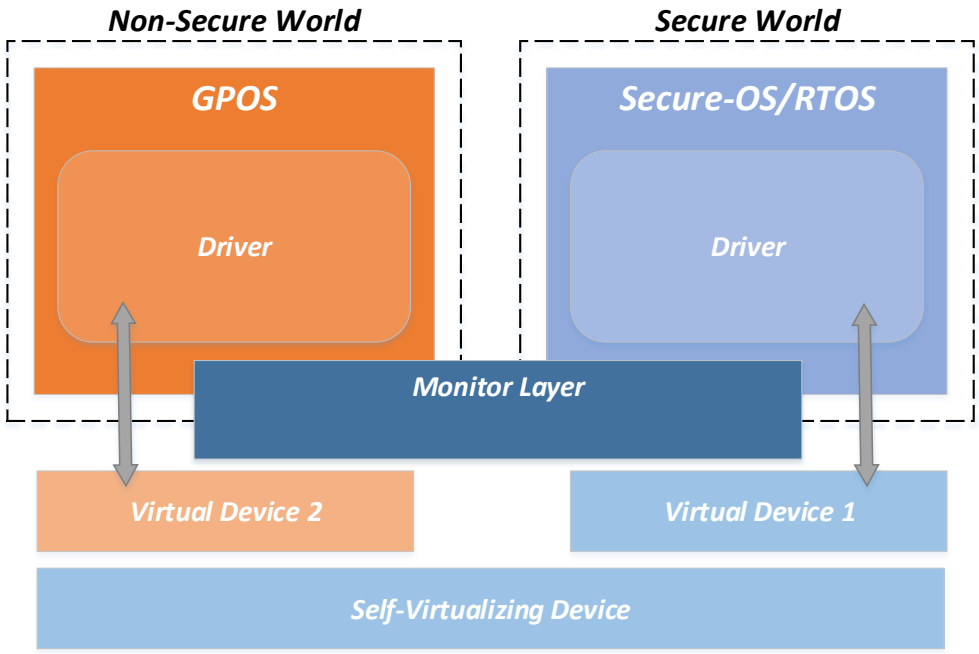


Figure 2.14: Self-virtualized devices method.

Each device is represented by a virtual interface (VIF) which is accessed from the guest OS through a device driver. A simple light-weighted API allows the guest to send and receive messages through two different queues. The self-virtualized device is responsible for managing VIFs by creating and destroying them or reconfigure parameters that define their performance characteristics. When a physical device wants to send data to the processing system it uses a communication channel to send data which is then demultiplexed into one of the existing VIF and sent over to the respective guest through the VIF's receive queue. The ability to multiplex/demultiplex various VIFs on a single physical I/O is the key task of a self-virtualized device. These scheduling decisions made as part as this task must enforce performance isolation among different VIFs and with built-in support for real-time reservations could be shared consistently by both the RTOS and the GPOS through separated interfaces, achieving near-native performance.

Unfortunately, the current availability of these virtual devices that support this approach, such as, virtual network interfaces, virtual block devices (disk), virtual camera devices, and others is limited in practice. Also, even though the API is lightweight, it stills incurs considerable overhead due to the required communication mechanisms, similarly to the Para-Virtualization approach. Therefore, unless it is targeted to systems where a large number of OSs require using the same

physical device, it does not deliver benefits comparatively to previously mentioned approaches. Most importantly, usually this approach does not address device's security properly, not protecting the device from possible exploits. Untrusted device drivers of the non-secure guest OSs may instruct the VIF to perform a malicious action to susceptible parts of the device logic, potentially compromising the entire device. Even though additional protection mechanisms could be implemented in the hypervisor to improve security, it would entail huge engineering efforts and possibly still not ensure security.

2.3 Gap Analysis

Given all the previously mentioned advantages and disadvantages of existing shared device access methods, Table 2.1 presents a gap analysis between all approaches.

Table 2.1: Existing shared device access methods comparison

	Shared device access methods					
Device sharing Requirements	Proxy Task	Device Emulation	Para-Virtualization	Para-Trustzone	Repartitioning Pure / Hybrid	Self-Virtualizing
Real-time	✗	✓	✓	✓	✓/ ✓	✓
Security	✗	✗	✗	✗	✗/ ✗	✗
Overhead	✓	✗	✗	✗	✓/ ✗	✓
Device latency	✗	✓	✓	✓	✗/ ✓	✓
Modifications	✓	✗	✗	✗	✓/ ✗	✗
Hardware Cost	✓	✓	✓	✓	✓/ ✓	✗

According to Table 2.1 and taking the reliable device sharing mechanism requirements into consideration, the existing methods fail to fully meet the following requirements:

- **Proxy Task**: This approach does not fulfill real-time requirements and does not take security into consideration, since the device is by default assigned to the non-secure world which performs the secure world requests.
- **Emulation**: Device emulation is not implementable in TrustZone-enabled systems. Even though, theoretically, its implementation has associated overhead and requires complex TCB modifications.
- **Para-Virtualization**: This method's performance is still far from native, requires considerable engineering effort, and also increases the TCB complexity.

- **Para-TrustZone**: Compared to para-virtualization, the Para-TrustZone approach achieves better performance through SMCs, although it incurs additional modifications to the monitor layer. Additionally, such as para-virtualization, it requires considerable engineering effort and some changes to the hypervisor code.

- **Pure/ Hybrid Re-partitioning**: The pure approach offers performance at the cost of higher device latency, and the hybrid approach offers lower device latency but introduces overhead, hence worst performance. Nonetheless, the device can also be compromised by the NS world for the same previous reason (when assigned to the GPOS can intentionally cause device failure).

- **Self-Virtualizing**: Even though this approach has great performance and takes into account real-time considerations, it requires adding an API to the hypervisor (TCB expense and modifications), considerable engineering effort, and most importantly not ensuring security.

Among the existing methods, the most prevailing and major issue is that the device's security is not properly addressed, not protecting the device from many possible exploits. Untrusted device drivers of the non-secure guest OSs may perform a malicious action to susceptible parts of the device logic, potentially compromising the entire secure device, and consequently the secure tasks that depend on the device. Even though many of these methods actions are performed under the hypervisor supervision and provide protection mechanisms, these incur huge engineering efforts and are still not sufficiently effective.

3. Platform and Tools

In this chapter the research platform and tools used during this thesis development are described. Firstly, the platform requirements are identified, and the development platform was chosen accordingly. Thereafter, the Advanced eXtendible Interface (AXI) bus protocol is discussed, given its relevance for the development process. Then, the Zynq platform device where the system is deployed and its most relevant and required features for the development process are scrutinized. Lastly, the LTZVisor, is carefully analyzed and its hosted OSs choice justified.

3.1 Platform Requirements

The main goal of this thesis is to develop a new approach for shared device access in TrustZone-based architectures and integrate it within the LTZVisor. For this goal to be feasible a set of requirements on the selected platform must be met:

- The selected platform must be able to host the LTZVisor, more precisely:
    - must include at least one ARM processor;
    - the ARM processor must feature a memory management unit (MMU), in order to run general purpose operating systems (GPOSs).
    - must feature ARM TrustZone security extension;
- The selected platform must provide a FPGA with enough resources to deploy the system design.

ARM Cortex-A9 is a mid-range cost-effectively processor, widely deployed, providing hardware-assisted virtualization features, and a memory management unit (MMU) as required. The Xilinx Zynq-7000 [Xil18, Pal14] is a TrustZone-enabled SoC featuring two ARM Cortex-A9, multiple useful security features and a FPGA, hence an adequate choice for this thesis. However, there are still several

development boards featuring a Zynq-7000 SoC. The Zybo was the selected platform due to its characteristic low cost, while still providing the required features and FPGA resources for this thesis implementation.

3.2 AMBA Advanced eXtensible Interface

ARM AMBA AXI [Xil11, LI04] bus, was introduced in 1996 and is already on the fifth version (AXI5). Since then, it has been used by ARM as the standard protocol for SoC communication. AXI can be categorized into three interface types, targeting applications with different specificities: AXI-Lite, AXI-Full, and AXI-Stream.

These interfaces can be configured as master or slave. An interface configured as master is responsible for starting and managing the performed transaction and its direction. In contrast, an interface configured as slave only answers the transaction requests from the master, and acknowledges write requests (if memory-mapped). The AXI BUS provides separate address/control and data channels. Firstly, control signals are set, then the data is transferred through the respective channel. Support is provided for unaligned data transfers using byte strobes, and Burst mode transactions.

AXI interfaces can be further categorized according to the addressing type: memory-mapped or point-to-point. Memory-mapped interfaces (AXI-Lite and AXI-Full), are accessible through memory addresses and feature five different channels: two for read operations, one for the address/control channel, and the other for the data channel; two for write operations, one for the address/control channel, and the other for the data channel; and the remaining channel to acknowledge write operations, if configured as slave. Read operations are not acknowledged because the master can always retry the read transaction whenever it fails. Point-to-point interfaces (AXI stream) only features one unidirectional channel for both control and data signals.

3.2.1 AXI-Lite

From the above mentioned interfaces, the AXI-lite [Xil12] memory mapped interface deserves to be highlighted for its low level complexity and small logic footprint. It requires fewer control signals, less hardware implementation effort, and less complexity to manage it from the user-side. Although, only one transaction can be performed each time. To perform an AXI-lite transaction, the following steps are required:

- **Handshake**: The master sets the respective control signals for the transaction. Then, if the issued transaction is a read operation the ARVALID is asserted, otherwise, if it is a write operation the AW-VALID is asserted instead. Lastly, to start the transaction the slave asserts the ARREADY or AWREADY signal accordingly.
- **Data Transfer**: When the master and slave AR/AW-VALID and AR/AW-READY signals are both set to high the data transfer is performed.
- **Acknowledge**: After the data transfer is done, the BRESP signal indicates the status of the transaction(BREADY and BVALID handshake), if successful is set to LOW.

Table 3.1 shows every AXI-lite signal name, source, channel, and description.

Table 3.1: AXI-lite signals. Adapted from [LI04].

Signal	Channel	Source	Descripton
ACLK	Global	Clock	Global clock signal.
ARESETn	Global	Reset	Global reset signal, active LOW.
AWVALID	Write address	Master	Indicates that valid write address and control information are available.
ARVALID	Read address	Master	Indicates, when HIGH, that the read address and control information is valid.
AWREADY	Write address	Slave	Indicates that the slave is ready to accept an address and associated control signals:
ARREADY	Read address	Slave	Indicates that the slave is ready to accept an address and associated control signals.
AWADDR[31:0]	Write address	Master	Gives the address of the first transfer in a write burst transaction.
ARADDR[31:0]	Read address	Master	Gives the initial address of a read burst transaction.
AWPROT[2:0]	Write address	Master	Indicates the normal, privileged, or secure protection level and whether the transaction is a data access or an instruction access.
ARPROT[2:0]	Read address	Master	Indicates the normal, privileged, or secure protection level and whether the transaction is a data access or an instruction access.
WVALID	Write data	Master	Indicates that valid write data and strobes are available.
WREADY	Write data	Slave	Indicates that the slave can accept the write data.

WSTRB[3:0]	Write data	Slave	Indicates which byte lanes to update in memory.
RVALID	Read data	Slave	Indicates that the required read data is available and the read transfer can complete.
RREADY	Read data	Master	Indicates that the master can accept the read data and response information.
RDATA[31:0]	Read data	Slave	Read from data bus can be 8, 16, 32, 64, 128, 256, 512, or 1024 bits wide.
RRESP[1:0]	Read data	Slave	Indicates the status of the read transfer. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.
BVALID	Write response	Slave	Indicates that a valid write response is available.
BREADY	Write response	Master	Indicates that the master can accept the response information.
BRESP[1:0]	Write response	Slave	Indicates the status of the write transaction. The allowable responses are OKAY, EXOKAY, SLVERR, and DECERR.

3.3 ZYBO Zynq-7000 SoC

Zybo (ZYnq BOard) [Pul16] is a low cost, feature-rich, ready-to-use development board featuring the Z-7010, a member of Xilinx Zynq-7000 family, based on the Xilinx All Programmable System-on-Chip (AP SoC) architecture, which integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA). The Zybo platform can host a complete system design by providing a wide variety of multimedia and connectivity peripherals. The platform features onboard memories, video and audio I/O, dual-role USB, Ethernet, and SD slot. These above mentioned characteristics are optimal for a Zynq developer beginner whose system design does not depend on the high density of I/O, among other large hardware resources capabilities present in mid-level and above boards. Another advantage is the compatibility with Xilinx's high-performance Vivado Design Suite as well as the ISE/XSDK toolset. This toolset provides an intuitive, facilitated design flow, by melding FPGA design with embedded software development. Targeting a wide range of systems with different design complexity, from a hypervisor running multiple OSs, down to a bare-metal application simply controlling LEDs. Figure 3.1 shows the Zybo board, managing to encompass all these features in a compact board.

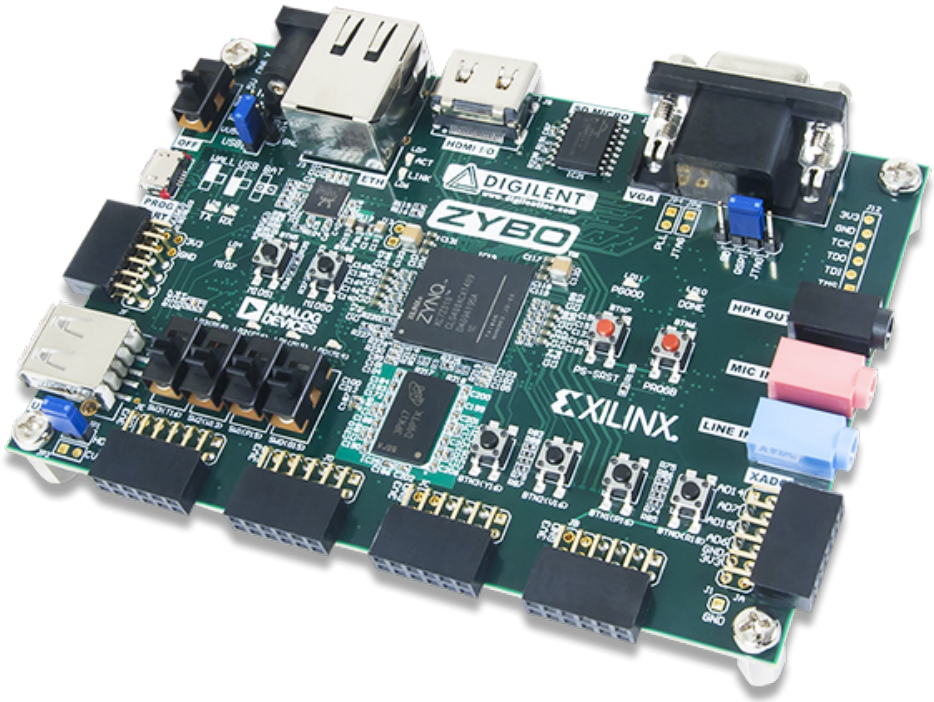


Figure 3.1: The ZYBO Zynq-7000 development board.

3.3.1 Zynq-7000 family

Figure 3.2 depicts the Zynq-7000 [Xil18] All Programmable SoC (AP SoC) architecture. Similarly to all Zynq devices, this architecture contains at least one ARM Cortex-A9 processor, the core component of the processing system (PS). Furthermore, the processing system encompasses an application processor unit (APU), memory interfaces (with multiple memory technologies), and I/O peripherals (interfaces for external data communication). The APU offers multiple high-performance features: a single/dual ARM Cortex-A9, with associated computational units, such as the FPU and NEON engine; an MMU; a 32 KB Level 1 data and instruction cache; 512 KB Level 2 data and instruction cache; a Snoop control unit (SCU) to maintaining L1 and L2 coherency; an Accelerator coherency port (ACP); timers for the watchdog and time-tracking; a General interrupt controller (GIC); a DMA controller; and 256 KB of on-chip memory (OCM).

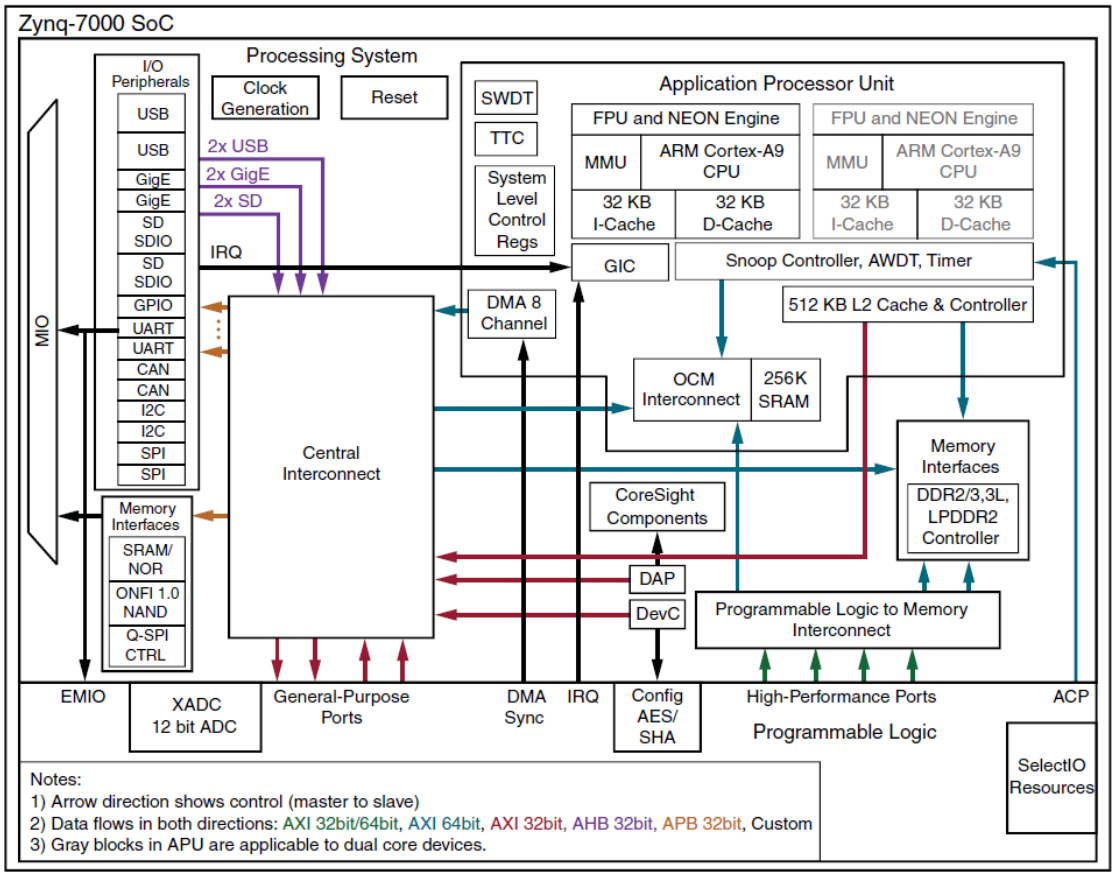


Figure 3.2: Zynq-7000 SoC overview [Xil18].

The programmable logic, based on the Artix-7 and Kintex-7 field-programmable gate array (FPGA) fabric with user-configurable capabilities provides Configurable logic blocks (with LUTs, flip-flops, other logic), block RAMs, clock management, digital signal processing, and I/O configurable blocks for interfacing.

PS-PL interactions are performed through advanced extensible interface (AXI). The nine existing PL AXI interfaces have multiple channels and encompass thousands os signals, and are divided into the following types:

- **General Purpose (AXI_GP)**: Interface with a 32-bit data bus, suited for general-purpose applications without high performance need, and are connected directly to the ports of the master/slave interconnect. Four general purposes interfaces are provided, two where the PS is the master and the PL the slave, and the other two where the PL is the master and the PS the slave.

- **High Performance (AXI_HP)**: Four high performance interfaces with a 32 or 64-bit data bus, includes high bandwidth datapaths do the DDR

and OCM memories, and two FIFO buffers to support burst transactions for read and write traffic.

- **Accelerator Coherency Port (AXI_ACP)**: A single port with low-latency that provides access to the PL (master), with optional cache coherency (through the SCU).

3.3.2 TrustZone technology Support in Zynq-7000 AP SoC

Zynq-7000 AP SoC includes FPGA programmable logic (PL) that enables designers to program the PL with custom and Xilinx IPs (hardware design language (HDL) modules). These IP cores are normally connected through a memory-mapped AXI interface.

3.3.2.1 AMBA Advanced eXtensible Interface

The most relevant feature in the context of this thesis is the extended AMBA AXI design on TrustZone-enabled SoCs, which provides an extra control signal for each of the read and write channels on the main system bus. These controls signals are called the Non-Secure, or bits, and are defined in the public AMBA3 Advanced eXtensble Interface (AXI) bus protocol specification, [Pal14, ARM09, LI04]. These mentioned bits are among others in the AWPROT or ARPROT signals, that provide three levels of access protection, [Pal14, Figure3.2]:

Table 3.2: AXI access protection levels.

ARPROT[2:0] **AWPROT[2:0]**	**Protection Level**
[0]	1 = Privileged Access
	0 = Normal Access
[1]	1 = Non-secure Access
	0 = Secure Access
[2]	1 = Instruction Access
	0 = Data Access

- AWPROT[0] and ARPROT[0], used by a master to indicate their processing mode, if high indicates a privileged acces, and low indicates a normal access.

- AWPROT[1] and ARPROT[1], provided by TrustZone enabled systems where a greater degree of differentiation between processing modes is required. If this bit is high indicates a non-secure access, and low a secure access.
- AWPROT[2] and ARPROT[2], differentiates an a data access from an instruction access. Low indicates a data access, and high an instruction access.

When a new transaction takes place these signals are set by the bus masters, and the bus or slave decoding logic interprets them ensuring that the required security separation is met. Masters configured as non-secure must set their NS to high in the hardware, which doesn't allow them to access secure slaves due to the decoded address not matching any secure slave. Whenever this happens, its implementation defined weather if a transaction is supposed to fail silently or generate an error, which in the latter case may be raised by the slave (SLVERR) or the bus (decode error, DECERR), depending o the on the hardware peripheral design and bus configuration [ARM09].

However, both secure and non-secure operating states might be supported by an AXI master, and also extend this concept of security to memory access. As shown in Figure 3.3, the bit AxPROT[1] identifies an access as secure or non-Secure, defined so that when it is asserted the transaction is identified as Non-Secure.

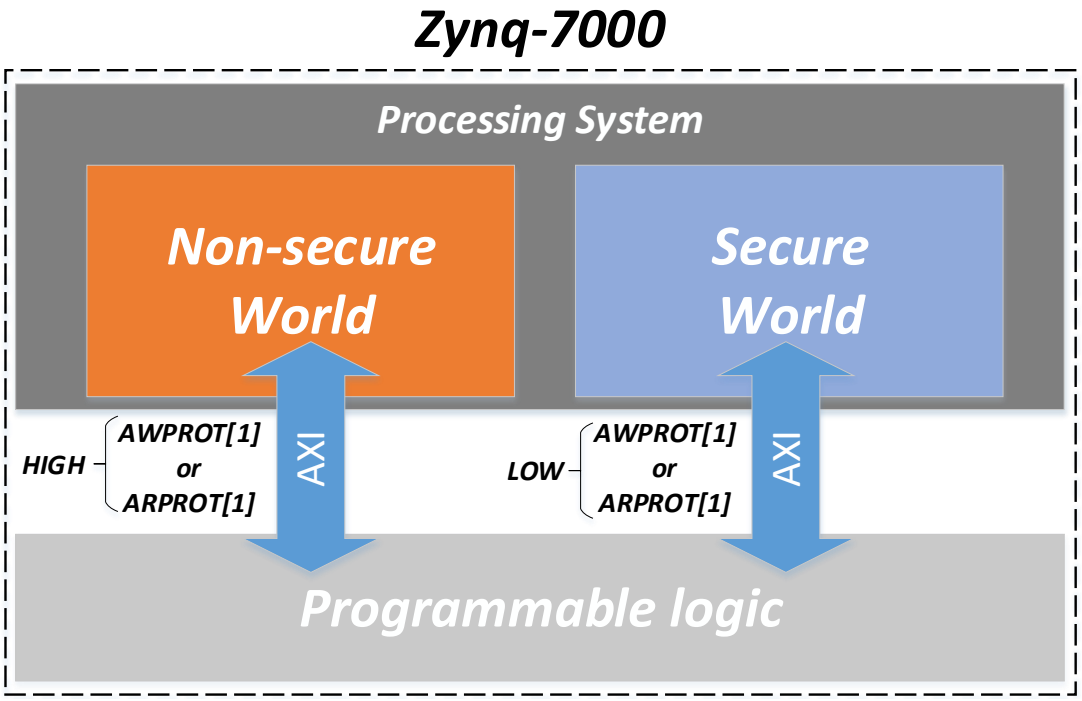


Figure 3.3: Advanced eXtensble Interface (AXI) non-secure control signals.

3.3.2.2 Xilinx AXI Interconnect IP Support

Zynq-7000 SoC AXI interfaces between the PS and the PL are AXI3 compliant, differently from Xilix IP cores which are AXI4 compliant. Therefore, the system designer must instantiate an AXI Interconnect IP core along with IP cores in the programmable logic to connect them to the processing system. The instantiated interconnect in the PL provides an additional secure bit checking feature (optional, and disabled by default). If the system designer enables this features on the AXI master interface, and a non-secure read/write transaction is attempted the DECERR will be issued and the transaction will not propagate any further. On the other hand, if disabled the non-secure transaction will also propagate to the slave.

3.4 Development Toolchain

This section addresses the development tools from Xilinx used during this thesis development: Vivado Design Suite, and Xilinx software development kit (XSDK).

3.4.1 Vivado Design Suite

Xilinx Vivado Design Suite [Xil16a], is a set of tool-chains created to aid development challenges throughout system's design, integration and implementation incurred by Xilinx devices complexity.

The Vivado IP integrator, allows developers to easily integrate Xilinx IPs from the IP library into their design and configure them through an user-friendly interface, also lets the user create custom IPs and add them to the library. The UI enables the user to easily connect the IP blocks and the rest of the system modules. Vivado enables developers to perform synthesis and implementations of their designs, perform timing, power, hardware utilization, verification analysis. Moreover, enables the design behavioral, post-synthesis, and post-implementation simulation through test benches, testing the system reaction to different stimulus.

3.4.2 Xilinx SDK

The Xilinx Software Development Kit (XSDK)[Xil16b] is the Integrated Development Environment(IDE) based on Eclipse, that directly interfaces to the Vivado embedded hardware design environment, for creating embedded applications on Xilinx's microprocessors. XSDK provides a feature-rich C/C++ code editor and

compilation environment for software development alongside the hardware development on Vivado.

The XSDK provides tools to easily access the hardware design previously created and to program the FPGA with the hardware generated and exported in Vivado. The required files for the PS and target processor initialization, accessing hardware devices, templates, and so on, are automatically generated, facilitating the entire development process. An integrated debugger supporting Zynq-7000 SoC is also provided delivering useful features such as: setting breakpoints, stepping through the program execution, checking the program variables and stack, and viewing the system's memory contents.

3.5 LTZVisor

The LTZVisor [PPG+17b] is an open-source lightweight TrustZone-assisted hypervisor. TrustZone provides, the already mentioned, two execution environments: the secure world, responsible for hosting the privileged and trusted software, while the non-secure world is responsible for hosting the non-privileged and untrusted software. This virtualization architecture is composed of two different VMs (the secure an non-secure VM) and the hypervisor itself, as depicted in Figure 3.4.

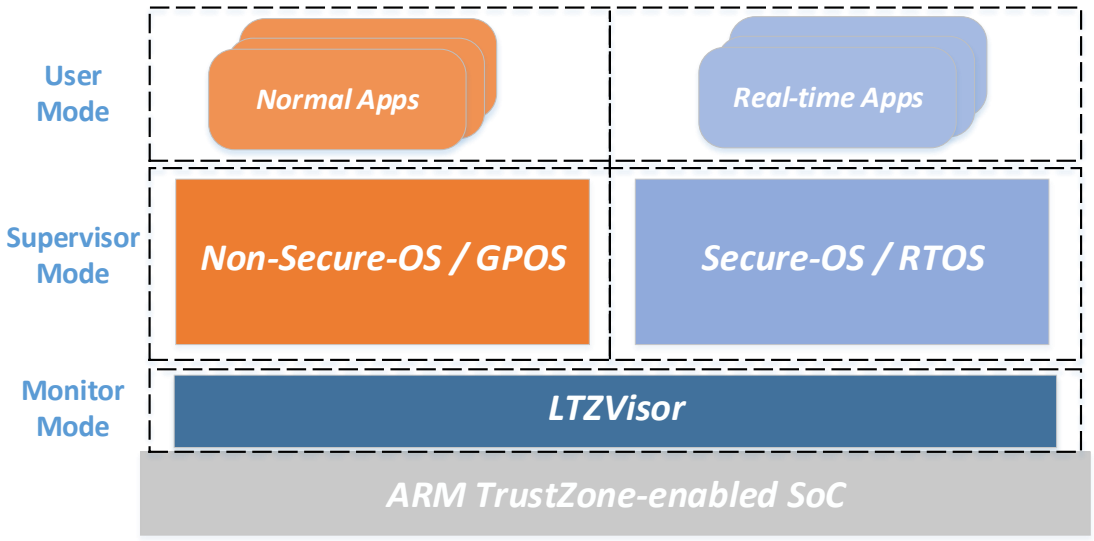


Figure 3.4: LTZVisor architecture overview.

LTZVisor runs in monitor mode with the highest privileged processor mode, thus, is always considered secure. By running in monitor mode, the hypervisor has full control of all hardware and software resources and is in charge of configuring memory, interrupts and devices assigned to each VM, as well as managing the

Virtual Machine Control Block (VMCB) of each VM during partition switches. Whenever a VM is about to be executed, the hypervisor is responsible for transferring the VM state from its respective VMCB to the physical processor context. Upon a new partition switch, the state of the active VM is saved back by the hypervisor into its respectiveVMCB and the same former procedure for the VM execution process is repeated. Both VMs run in the supervisor mode. The secure VM, running on the secure side runs privileged code that can access or modify any of the non-secure VM resources, such as its memory and associated devices. Therefore, the OS hosted on the secure VM must be aware of its virtualization and considered part of the TCB, hence must keep a small TCB. An RTOS is ideal to run on the secure side due to its characteristic small footprint and strict time constraints, which can be met because of the higher execution privilege. The non-secure VM, running on the non-secure side is ideal to host a GPOS, useful for running human-machine interfaces and containing rich libraries and drivers. Software in the non-secure world is completely isolated from privileged software in the secure world. An exception to the hypervisor is triggered whenever an attempt from the non-secure world to access secure world resources is performed.

3.5.1 CPU virtualization

TrustZone hardware security extensions virtualizes a physical core into two virtual cores, providing two completely separated execution environments, the secure and non-secure world. Each world contains an individual copy of banked registers. On the non-secure side the VMCB is composed by 25 registers: General Purpose Registers (R0-R12), the Stack Pointer (SP), the Linker Register (LR) and Saved Program Status Register (SPSR) for the Supervisor, System, Abort and Undef modes. However, for the IRQ and FIQ modes the General Purpose Registers (R8-R12), as well as the SP, LR and SPSR registers are not replicated and included due to being mutually exclusive for each world. The monitor mode is only dedicated to the secure world. On the secure side, the VMCB is composed only by 16 registers: General Purpose Registers (R0-R12), the SP, the LR and SPSR for the System mode. The reduced size of the secure VMCB promotes faster partition switches from the non-secure to the secure world, reducing the secure interrupts latency when the non-secure is executing.

Most of the co-processor register are banked, however, some must be preserved, such as the System Control Register (SCTLR) and the Auxiliary Control Register (ACTLR) which are responsible for configuring and controlling memory, cache, MMU (enabling or disabling), AXI accesses, and so on. To promote system's

security, TrustZone denies any attempt from the non-secure world to change any of these registers. Is the hypervisor's responsibility to initialize the non-secure VMCB registers before the boot process, enabling MMU, Level1 cache and other functionalities required for the GPOS to run. Otherwise, if attempted to modify these registers during the non-secure boot, the GPOS will get stuck.

3.5.2 Scheduler

LTZVisor implements an asymmetric or idle scheduler where the hypervisor behaves in a passive way. The scheduling process is carried out by the secure guest OS itself, with higher scheduling priority than the non-secure guest OS. The asymmetric design principle ensures that the non-secure guest is only scheduled during the idle periods of the secure guest OS and the secure guest is able to preempt the execution of the non-secure guest.

3.5.3 Memory isolation

As aforementioned, on TrustZone enabled SoCs without virtualization extensions, only MMU single-level address translation is provided, instead of the traditional 2-level address translation offered by virtualization extensions that allow the execution and spatial isolation of unmodified guests. However, through TZASC, memory can be configured and partitioned into different memory segments with different security privileges. Memory regions can be configured with a specific, implementation-defined, granularity which in Zybo Zynq-7000 platform is 64MB. The non-secure VM must have its respective memory segments configured as non-secure and the remaining memory as secure. An exception is automatically triggered and the execution control redirected to the hypervisor whenever the non-secure guest attempts to access any of the secure memory segments. The security status of each particular memory segment is defined by a system level control register named TZ_DDR_RAM. Figure 3.5 shows the LTZVisor memory security configurations on the Zybo platform, in which the non-secure memory region was reduced from the original memory configuration due to Zybo smaller memory resources.

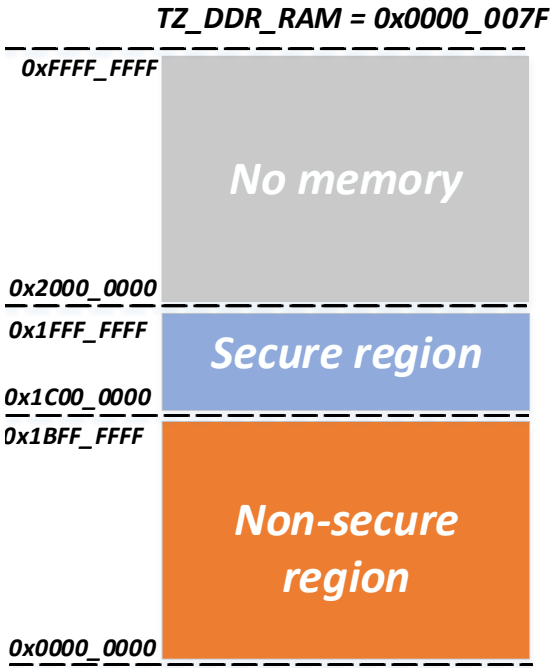


Figure 3.5: LTZVisor memory configuration.

3.5.4 MMU and Cache

With TrustZone security extension, the processor provides two virtual Memory Management Units (MMUs), delivering separate virtual-to-physical memory address translation tables to each world. Therefore, each world has an individual copy of the TTBR register set and an independent MMU configuration, accelerating world switching due to removing the need to invalidate translation lookaside buffer (TLB) entries. This memory isolation is extended and still available at cache level, again with the NS bit, tagging each entry with the processor security state upon the access. At the cache-level, entries from both worlds can coexist removing the need for duplication and cache flushing, consequently accelerating world switching and improving LTZVisor performance. On Zybo Zynq-7000 platform the L2 cache can only be enabled/disabled from the secure world side, hence is the hypervisor responsibility to manage the L2 Control register (reg1_control), enabling it before non-secure guest boot (similarly to L1 cache) so that the non-secure world can, thereafter, manage its non-secure entries.

3.5.5 Device partitioning

Devices can be either dynamically or statically configured as secure or non-secure in TrustZone-based systems. Isolation at the device level is ensured when devices are partitioned between both worlds and not shared among them. Device

virtualization on the LTZVisor is divided into three phases: Firstly, at design time, devices are assigned to a specific partition; then, during boot time, devices are configured; thereafter, devices are managed directly by the guest partition, i.e. following the so-called pass-through policy. Consequently, devices assigned to the RTOS are configured as secure, and devices assigned to the GPOS configured as non-secure. When an access from the GPOS to a device that has been configured has secure is attempted an exception will be triggered and immediately handled by the hypervisor. To configure the device security settings, on Zybo Zynq-7000 platform its provided a set of registers accessible from the secure side, such as, Secure Digital Input Output (SDIO) slave security registers (TZ_SDIO), APB slave security registers (security_apb), AXI GP0 master port (fssw_s1) security setting register, and so on.

3.5.6 Interrupt managment

The Generic Interrupt Controller (GIC) in TrustZone-enabled SoCs, provides both secure and non-secure prioritized interrupt sources. The GIC enables an interrupt model where secure interrupts are configured with higher priority as FIQs and non-secure interrupts with lower priority assigned to IRQs. This interrupt model is suggested by ARM and adopted by the LTZVisor, and prevents a denial-of-service attack against the secure world (from the GPOS). Every implemented interrupt security must be configured by setting the respective bit in the Interrupt Security Registers set (ICDISRn) accordingly. To assign a secure interrupt source to the processor FIQ interrupt mechanism, the FIQen bit in the CPU Interface Control Register (ICPICR) must be set. With this implementation shown in Figure 3.6, when the secure world is executing, if a secure interrupt (FIQ), is triggered it is handled by the RTOS itself, in order to avoid adding overhead to the RTOS interrupt latency, hence not requiring hypervisor's intervention. This behavior is achieved by disabling the FIQ bit in the Secure Configuration Register SCR. Also when the RTOS is running, if a non-secure interrupt (IRQ), arises (using the processor IRQ mechanism) it does not affect the guest behavior and the interrupt is only attended as soon as the non-secure guest becomes active again.

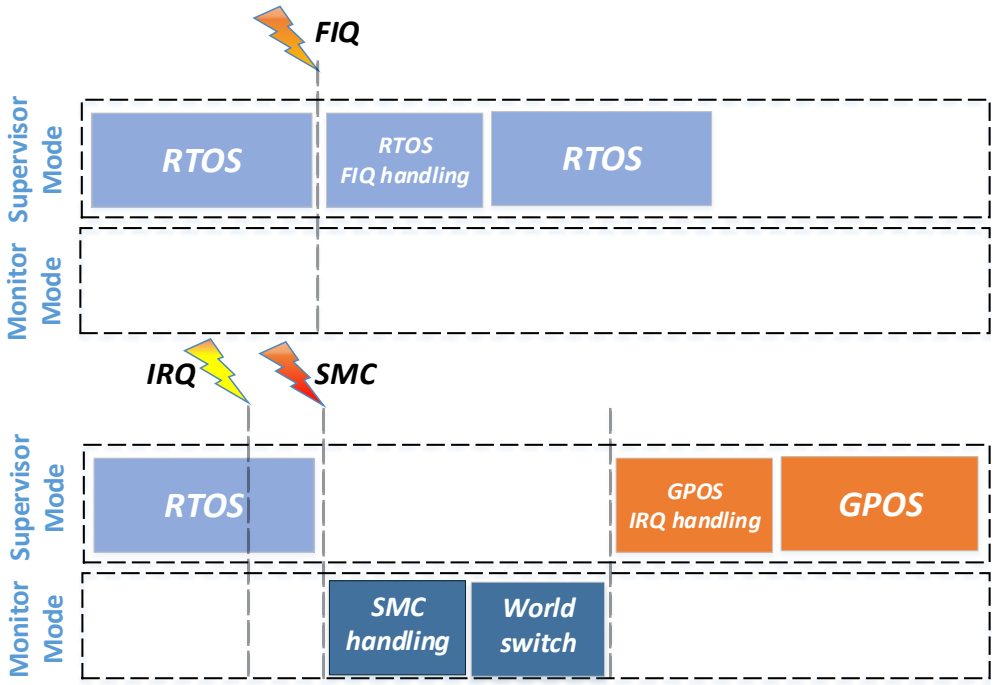


Figure 3.6: LTZVisor interrupt management when RTOS is runnning.

Otherwise, if the non-secure guest (GPOS) is running and a FIQ arises, the hypervisor takes immediate control and handles the secure interrupt directly in monitor mode. If an IRQ arises instead, the interrupt is directly handled by the non-secure guest itself. Figure 3.7 shows the described interrupt behavior in the mentioned scenario.

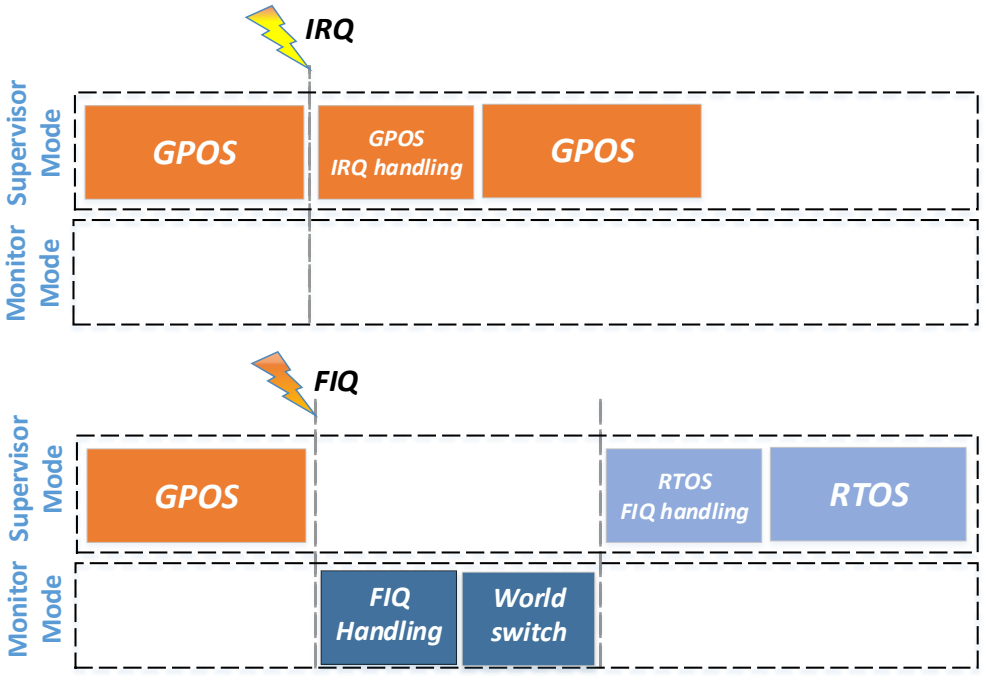


Figure 3.7: LTZVisor interrupt management when GPOS is running.

3.5.7 Time management

Due to its asymmetric scheduler and dual-OS configuration, time management on the LTZVisor is performed using two independent timers, one for each guest OS. The hypervisor dedicates the Triple Timer Counter (TTC) 0 to the secure VM and the Triple Timer Counter (TTC) 1 to the non-secure VM, thus it must be configured as non-secure at boot time to prevent undesired exceptions. This time management implementation ensures: all timing structures are always updated correctly; The GPOS, even though tick-less, has notion of the real-time; The RTOS does not miss any system-tick interrupt.

3.5.8 Execution Flow

A set of initialization configurations must be performed at the beginning of the LTZVisor execution, in the boot process, for the system to act as expected. These include the previously mentioned, processor and co-processor registers, memory (partition and security), stack, peripherals (partition and security) and the interrupt controller (GIC, interrupt model setup). Following the system boot process, the RTOS is booted and its real-time tasks are scheduled. As soon as the RTOS tasks become idle, a system call is performed to enter the monitor mode (hypervisor), through a SMC instruction and prepare a world switch. Then, the hypervisor handles the SMC instruction, preparing the transition to the non-secure world. During this preparation phase the hypervisor's responsibility is to: save the processor state of the secure world, RTOS, in the respective VMCB, and restore the GPOS context saved in its VMCB; enable the FIQ and NS bits of the SCR register; set supervisor mode; update the linker register with the start of the NS OS address space; and jump to the restored non-secure address. Thereafter, the GPOS will run until a FIQ arises, switching the processor to monitor mode and entering the FIQ handler on the monitor vector table. The hypervisor, handles the FIQ, disables FIQ and NS bits of the SCR register, saves the current processor state into the non-secure VMCB and restores the context from the secure VMCB. When the processor is in supervisor mode running the RTOS, its real-time tasks are dispatched until the moment the idle task is re-scheduled, and all previous steps need to be repeatedly performed. Figure 3.8 depicts and summarizes the described execution flow.

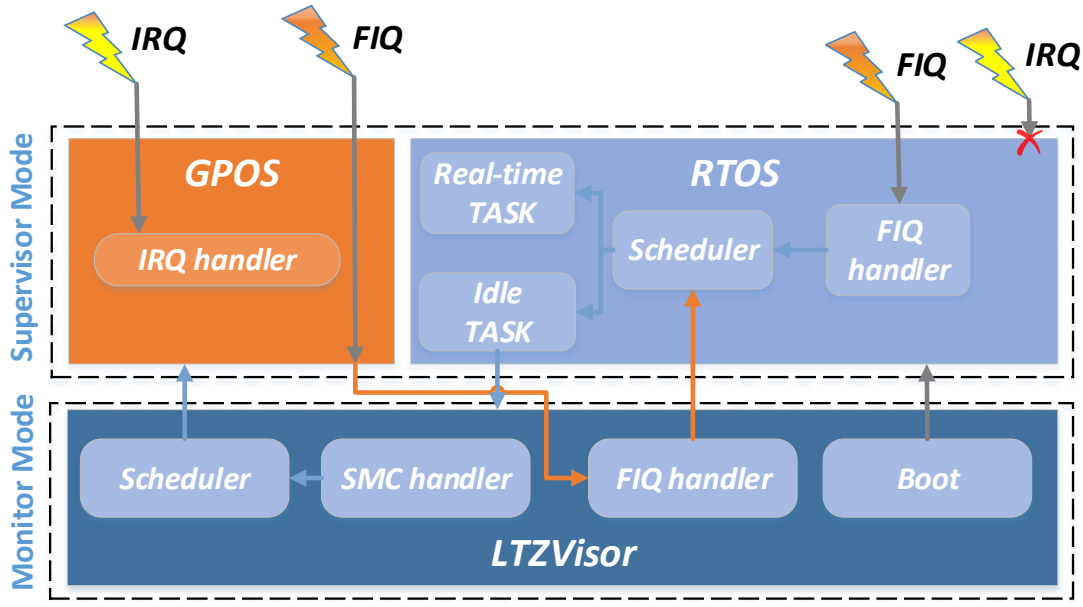


Figure 3.8: LTZVisor execution flow.

3.6 Operating System stacks

LTZVisor, as a virtualization infrastructure, is able to consolidate and run two guest OSes on the same platform. This section describes the OSes which were used as guest OSes to run on top of LTZVisor.

3.6.1 FreeRTOS

FreeRTOS [Fre, Rea16] is a real-time operating system(RTOS) offering a completely free, smaller and easier real-time processing alternative for applications where GPOSs are not sufficient enough to meet the design security and timing requirements. FreeRTOS core kernel is very simple and kept minimal, with most of its source code written in 'C' language. FreeRTOS software architecture can be separated in two different layers [WW09], illustrated in Figure 3.9. The hardware independent layer contains most of the OS functions and remains intact for all architectures. This layer is composed of two mandatory files, 'list.c' and 'task.c' that provide task management and scheduling functionalities and a list data structure for maintaining task queues, respectively. Task scheduling follows a priority-based policy, prioritizing the execution of highest priority level tasks, although if two equally privileged tasks are set to be executed the scheduler uses a round-robin model. In this same layers, three other optional files are available: 'queue.c', implements priority queues data structures for inter-task communication and synchronization purposes; 'timers.c', provides functions facilitating software

timers implementation, for the application tasks; 'croutine.c', offers support for co-routines, a special type of task with additional memory efficiency.

The portable layer is responsible for architecture-specific processing (e.g. context switching) and should at least contain the 'port.c' file, with hardware-specific code and the standard API for the hardware independent layer. And, the 'heap.c' file, with the architecture-specific memory allocation and deallocation functionalities.

The main reason of the FreeRTOS selection over other RTOS are: the ability for scalability and internal design modifications due to its open-source code, the acceptable engineering effort to perform modifications given its minimal kernel core, and the large support availability on different architectures, application and configurations [PPO+14].

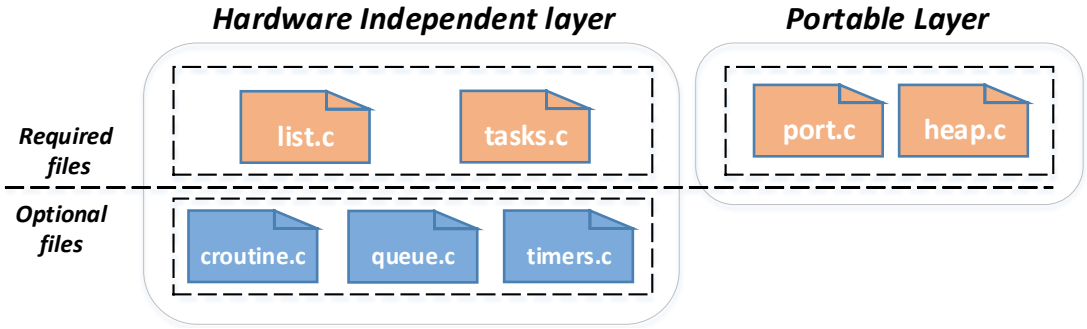


Figure 3.9: FreeRTOS software layers.

3.6.2 Linux

Linux is a free and open-source GPOS, initially developed for personal computers use. Currently, Linux is ported to more platforms than any other OS, mainly due to the large presence of Linux kernel-based Android OS, wide use in embedded devices, and many smartphones and tablets running on Linux kernel-based OSs. Linux is one of the OSs with most community support and largest user base, an extremely advantageous characteristic delivering support to new features, technical issues, or even to manage added hardware resources that require device drivers, which might be already developed by someone among the open-source community. Another huge advantage, especially in embedded devices, is the possibility to compile a custom Linux kernel.

Given the massive prevailing presence of Linux in embedded systems, many commercial specialists focused on porting Linux to embedded systems. Xilinx company already provides support for various Linux distributions ported to their

platforms, including platforms from the Zynq family, such as the chosen Zybo development board.

Xilinx Zynq Linux [Xil] is based on the original Linux kernel with additional Xilinx features (board support packages, and Xilinx drivers), and is frequently updated to the last Linux Kernel versions. The building and running processes for ARM Linux are very similar in Xilinx Zynq Linux. To perform the Linux boot process on Zynq platforms, the processing system (PS) and custom developed hardware information must be jointly provided.

4. Self-Secured Devices

This chapter addresses the implementation of the self-secured approach and its application to devices with different complexity levels. This chapter also describes the developed device drivers for managing the devices, and the performed modifications to the LTZVisor and hosted guests OSs, required to integrate and test the developed devices in this hypervisor.

4.1 Overview

Figure 4.1 illustrates the proposed generic architecture of a self-secured device. Self-secured devices extend the TrustZone dual-world concept to the inner logic of the hardware device, aiming to enable security improvements to shared devices access in TrustZone-based architectures.

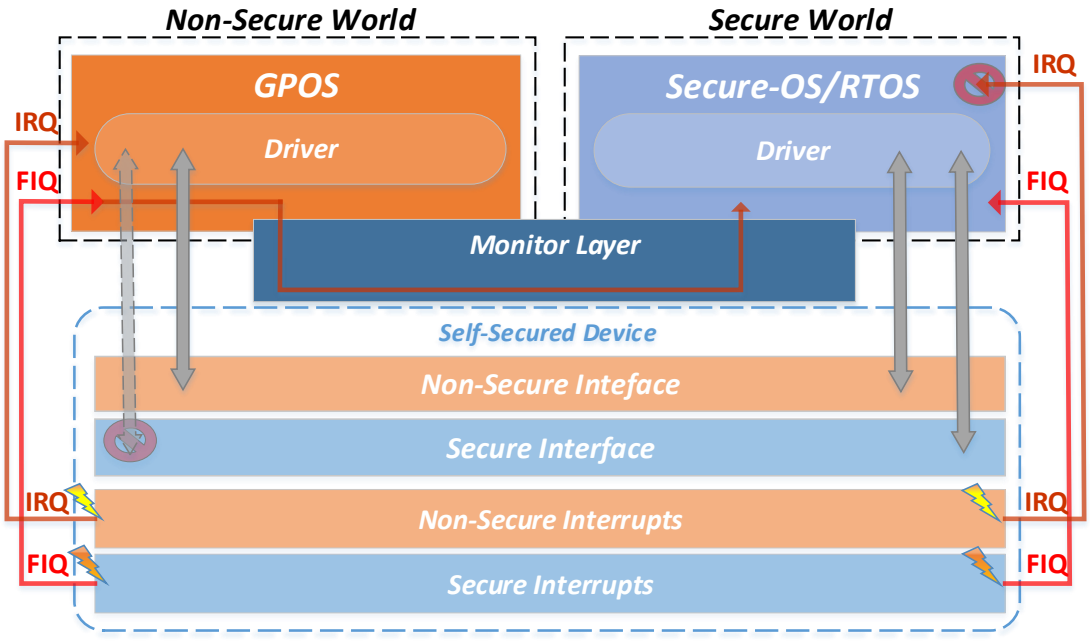


Figure 4.1: Self-Secured Device Generic Architecture

Certain accesses and actions performed by the non-secure world to a device can be potentially exploited to compromise the device and consequently, the secure world itself and its execution. In reconfigurable SoCs (e.g., Zynq-7000) access to the hardware device, in the programmable logic, is performed through the Advanced eXtensible Interface (AXI). In TrustZone-enabled SoCs this interface includes an additional control bit for each of the read and write channels. The AXI security state can be checked through the AWPROT and ARPROT signals by reading the non-secure bit that indicates which world is accessing the device. This allows checking if the non-secure world is trying to access some sensitive registers or configurations of the device and denying access when requested. However, it is imperative to split these sensitive registers and configurations from the rest of the device's logic and eventually deny non-secure world access to the vulnerable logic part of the device.

A possible approach to solve non-secure world inaccessibility to the vulnerable logic is by replicating the vulnerable logic part of the device, that could be potentially exploited. By creating a separate non-secure interface with an extra copy of this sensitive registers and isolating them into two different interfaces, with different banked registers in both interfaces, it enables both the non-secure and secure world to perform concurrent accesses to the device while reassuring its security cannot be compromised. Following this approach, the secure world is granted access to both the secure and non-secure logic interfaces, while the non-secure world access is restricted to the non-secure logic interface. Access from the non-secure world to the device's configurations is performed under the supervision of the secure side.

Another important condition to self-secure a device is to discriminate non-secure from secure interrupts and route them accordingly, from the programmable logic to the processing system. Therefore, the system GIC should also be configured to route FIQs to the secure world, and IRQs to the non-secure world. Likewise, secure interrupt sources from the device must be routed as FIQs and non-secure interrupts as IRQs. Following the same interrupt model as the LTZVisor and suggested by ARM, if the secure world is executing and a FIQ arises, it is handled by the RTOS itself in order to avoid adding overhead to the RTOS interrupt latency. In contrast, if a non-secure interrupt, IRQ, arises it does not affect the guest behavior and the interrupt is only handled as soon as the non-secure guest becomes active. Otherwise, if the non-secure guest (GPOS) is running and a FIQ arises, the hypervisor takes immediate control and handles the secure interrupt directly in monitor mode. If an IRQ arises instead, the interrupt is directly

handled by the non-secure guest.

4.2 Self-Secured Private Timer

The ARM's Cortex-A9 private timer [ARM12] was selected as an example of a low-complexity device for implementing the self-secured approach. This implementation consists on a replica of the original private timer based on ARM's provided documentation. Table 4.1 depicts the private timer register address map, which is composed of four main registers: Load value, Counter Value, Control (i.e., prescaler, auto-reload, enable), and Interrupt Status.

Table 4.1: Private Timer register map.

Baseaddress (0x43C00000) + **Offset**	**Type**	**Name**
0	RW	Private Timer Load Register
4	RW	Private Timer Counter Register
8	RW	Private Timer Control Register
12	RW	Private Timer Interrupt Status

The device supports the following features:

- 32-bit Counter that triggers an interrupt when reaching zero.
- Two configuration modes: Single-shot or auto-reload.
- Load value that can be used to configure counter starting values.

The timer configuration and control is performed through the control register, detailed in Figure 4.2.

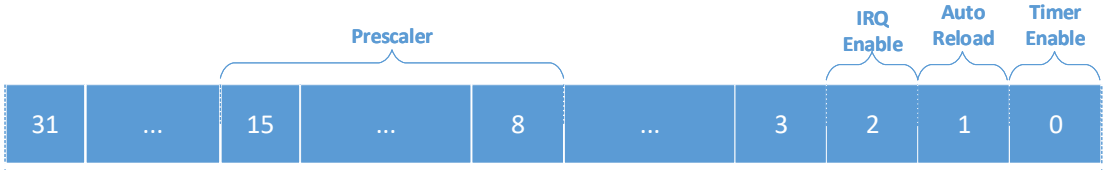


Figure 4.2: Private Timer: Control Register.

The Counter register keeps the counter value, decremented every clock tick while its value is greater than zero and the timer enable bit is set, in the Timer

Control register. When the counter value reaches zero, if the auto reload bit in the control register is set, the value on the load register is used to reload the counter which will be again decremented until zero. Otherwise, if in single-shot mode (auto-reload bit disabled), when the counter reaches zero, the timer stops until it gets re-enabled on the control register. Moreover, in both cases, whenever the counter reaches zero and the IRQ enable bit is set the event flag is set and the overflow interrupt is generated. This described behavior is shown in Figure 4.3. The interrupt event flag is a sticky bit set in the interrupt status registers, that must be cleared by the interrupt handler.

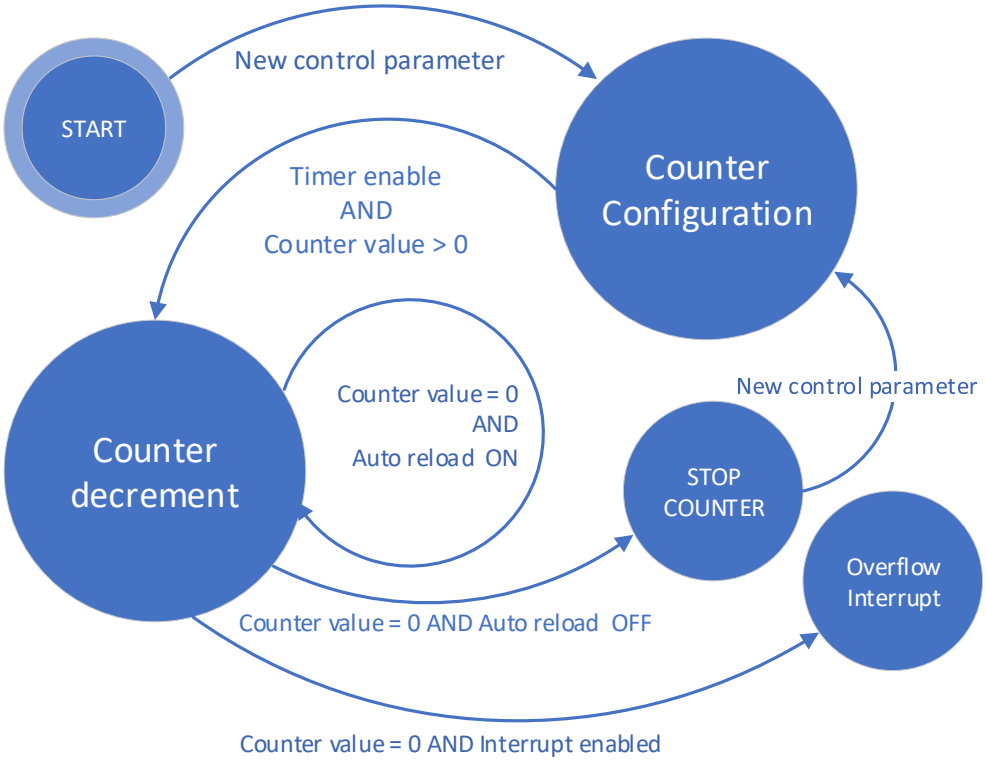


Figure 4.3: Private Timer Counter finite state machine.

Figure 4.4 illustrates how the counter, control parameters and interrupt status are updated, through control signals set upon any AXI register modification. The counter value can be updated by changing the AXI load or counter register, which will replace the current counter value. The counter is also updated with the load register value whenever the counter reaches zero and auto-reload is set. After generating the overflow interrupt, it can be cleared by setting the respective bit on the interrupt status register. Whenever the AXI control register is modified, its associated parameters (i.e. timer emable, auto-reload, prescaler, interrupt control) are all updated.

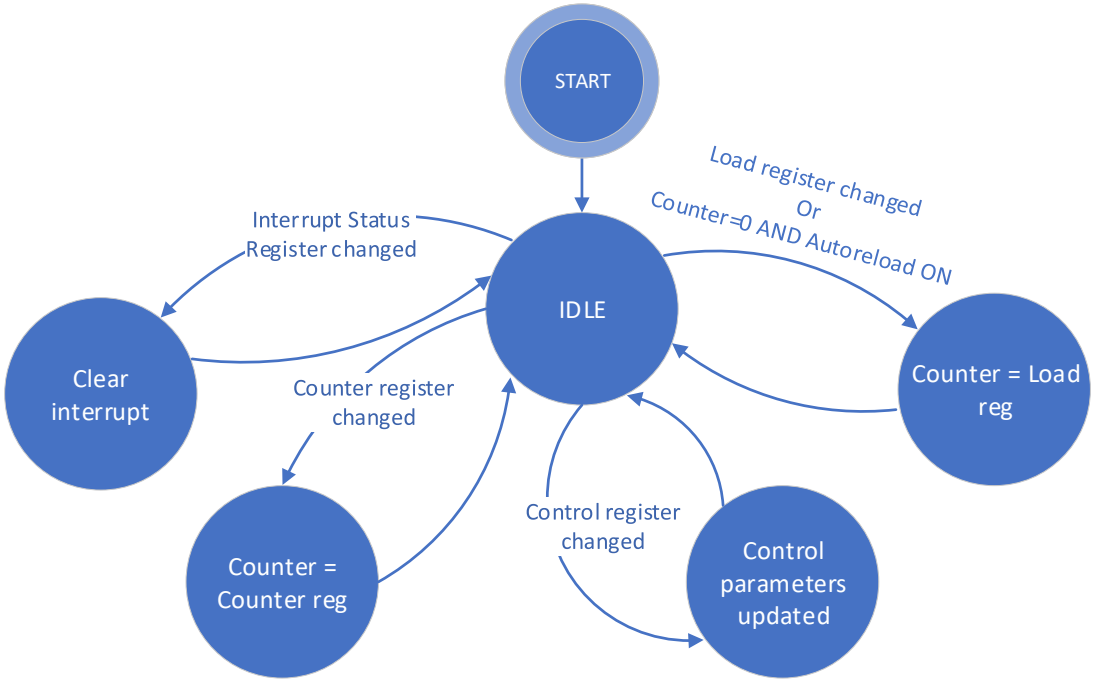


Figure 4.4: Private Timer control signals finite state machine.

The prescaler allows the timer clock tick to be set at the desired rate, which value is defined through the control register. The value between two timer ticks can be calculated by the following equation:

$$\frac{(\text{prescaler} + 1) \times (\text{Load\_value} + 1)}{\text{CLK}}$$

The block design of the implemented timer is shown in Figure 4.5. The hardware logic is composed by the Zynq processing system, a reset module, the AXI interconnect, and the timer itself, driven by the clock divider module which divides the user-defined prescaler by the original reference clock source.

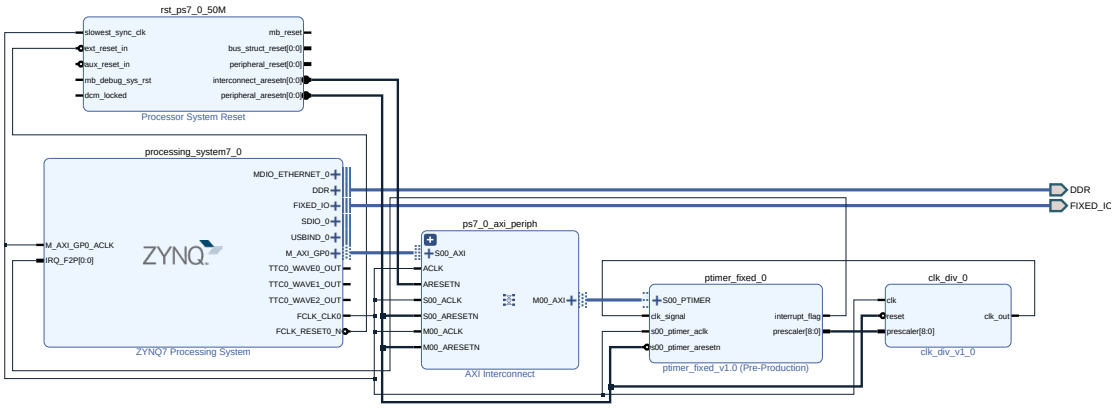


Figure 4.5: Private Timer Block design.

Firstly, the Zynq processing system must be instantiated and configured through an user interface, enabling the following features: (i) PL-PS shared interrupt ports, (ii) the general purpose AXI master interface 0 to connect with AXI-lite private timer slave interface, (iii) the UART for debug and test purposes, and the rest of the configurations left as default. Then, as illustrated in Figure 4.6, the private timer is instantiated based on an AXI-lite IP, the prescaler is connected to the clock divider module, and the overflow interrupt connected to the PL interrupt ports. An AXI interconnect is required since the interfaces of the PL need to be AXI3 compliant, differently from the instantiated IP cores, which are AXI4 compliant.

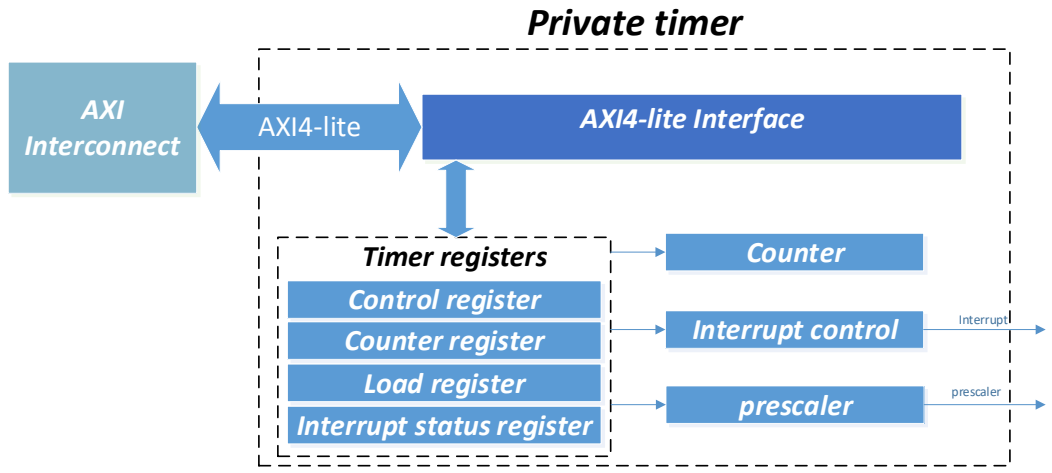


Figure 4.6: Private Timer Block diagram.

4.2.1 Device driver

To access and manage the device, a software interface to the hardware device must be provided, enabling both the GPOS and the RTOS access to hardware functions through an abstraction layer. The device driver contains the private timer registers physical addresses shown in Figure 4.1, and provides the following functions to manage the timer, by accessing its respective AXI registers:

- PrivateTimer_CnfgInitialize: Initializes the device structure with device's physical base address, ID, and status;
- PrivateTimer_Start: Sets the enable bit in the control register, enabling the timer;
- PrivateTimer_Stop: Disables the enable bit in the control register, disabling the timer;

- PrivateTimer_SetPrescaler: Sets the prescaler bits in the control register with the respective value;
- PrivateTimer_GetPrescaler: Reads the prescaler bits in the control register;
- PrivateTimer_Set_Load_register: Sets the load register with the passed argument value, also updating the counter value.
- PrivateTimer_Get_Load_register: Reads the load register.
- PrivateTimer_Set_Counter_register: Sets the counter register with the passed argument value, updating the counter value.
- PrivateTimer_Get_Counter_register: Reads the counter register.
- PrivateTimer_Set_Control_register: Sets the control register value with the passed argument value.
- PrivateTimer_Get_Control_register: Reads the control register value.
- PrivateTimer_Set_Interrupt_status_register: Allows the interrupt handler to clear the interrupt flag by setting the interrupt status register.
- PrivateTimer_Get_Interrupt_status_register: Reads the interrupt status from the interrupt status register.

4.2.2 Self-Securing the Private Timer: Minimal Approach

Following the self-secured concept, the device logic is divided into two separate interfaces with different banked registers. To self-secure the private timer two different approaches can be adopted. The choice lies in a trade-off between less engineering effort and additional support for the extra provided interface. The vulnerable logic part of the device must be isolated from the non-secure world, which encompasses registers that may: (i) compromise the counter value of the secure world; (ii) trigger unexpected and unintended interrupts by changing interrupt configurations; (iii) change device configurations that are normally performed at boot time (e.g. prescaler); (iv) tamper with device's normal flow by changing the device secure configuration (e.g. auto-reload).

With the former approach, the AXI registers remain intact but the main internal counter and load registers are entirely duplicated as they compose the functional part of the counter infrastructure. These registers are distinct from the controller and interrupt status registers which are only extended and partially

replicated. This allows each world to have its own configurable counter interface. However, the secure interface is exclusively accessible by the secure world and the non-secure interface is exclusively accessible by the non-secure world. This spares adding additional AXI registers and making any kind of modification to the device drivers. Although, it does not allow the secure world to take advantage of the extra provided counter interface. Nonetheless, an additional interrupt is provided in order to assign different interrupts for each counter overflow. In this sense, whenever the counter value from the non-secure register bank overflows, an IRQ is triggered, and whenever the counter value from the secure register bank overflows, a FIQ is triggered instead. The described architecture is illustrated in Figure 4.7.

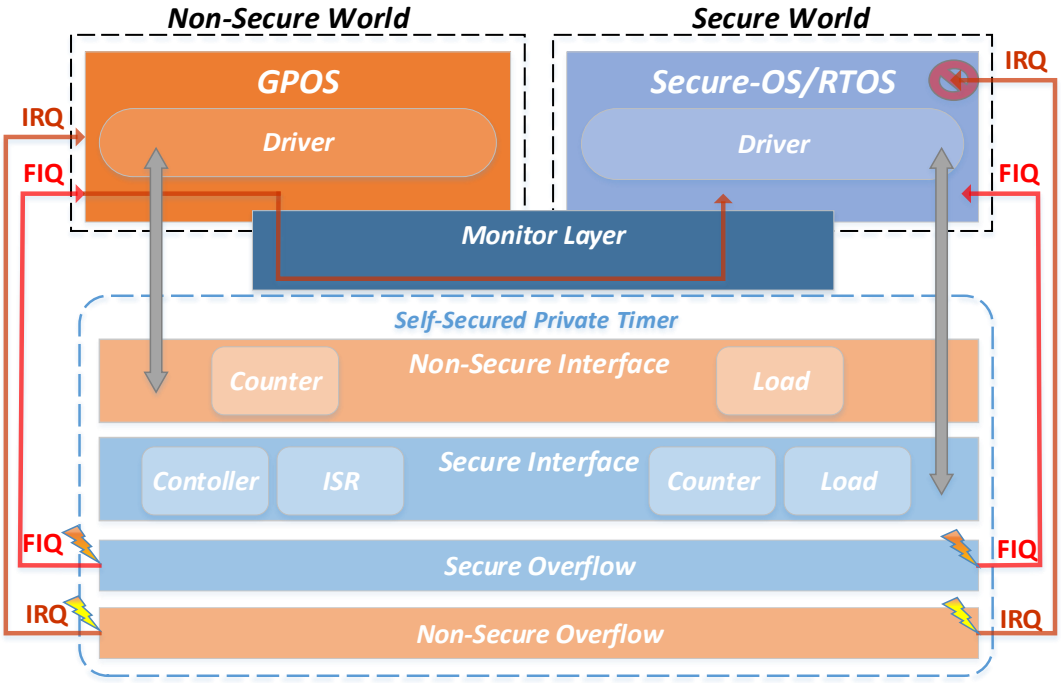


Figure 4.7: Self-Secured Private Timer: Minimal Approach architecture.

The extended control and interrupt status registers shown in Figure 4.8 do not provide non-secure world access to the entire registers, and do not require a banked copy of the entire register in the non-secure interface. Instead, they provide additional bits for the interrupt configuration and interrupts event flags for the non-secure world. Besides, the secure parameters in the control register cannot be accessed by the non-secure world. For instance, the prescaler value predetermines the counter tick and influences the whole timer, thus the non-secure

world cannot modify the respective registers bit, which is configured by the secure world, typically at boot time. Likewise, the interrupts configuration is also exclusively configured by the secure world, avoiding the generation of unintended interrupts. Similarly, the secure interrupt FIQ event flag in the interrupt status register should only be modified by the secure world. Therefore, non-secure world accesses to these mentioned secure parameters and events flags are restricted through the TrustZone protection signals. Moreover, the auto-reload and timer enable bit parameters are replicated and banked in both worlds, allowing each world's interface to access the respective copy of its banked registers. Therefore, from the secure interface only the auto-reload and timer bits banked in the secure world are exclusively accessible, and from the non-secure interface only the non-secure banked bits are accessible.

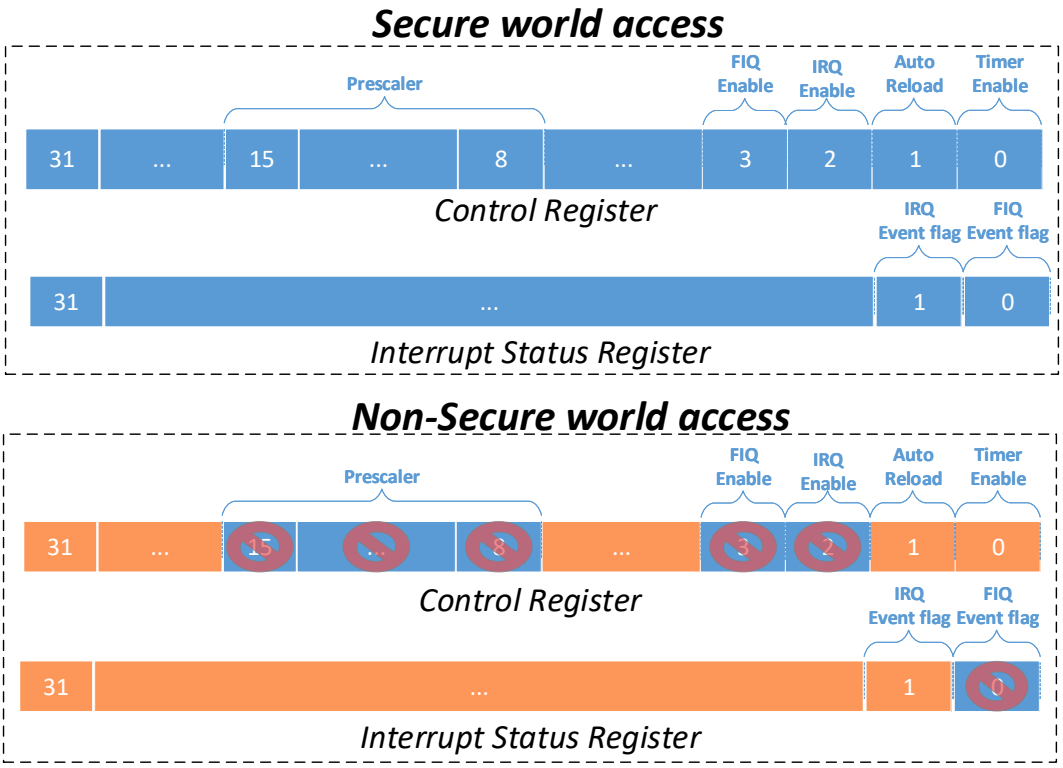


Figure 4.8: Minimal Approach control and interrupt status register.

The behavior of the AXI registers access upon read/write operations from both the secure and non-secure world is illustrated in Figure 4.9. From the secure world perspective when accessing the timer register only the secure banked registers can be accessed and from the non-secure world perspective only non-secure register. As previously explained, and illustrated, write accesses are restricted based on

the AWPROT TrustZone extended protection signal and read accesses restricted based on the ARPROT signal.

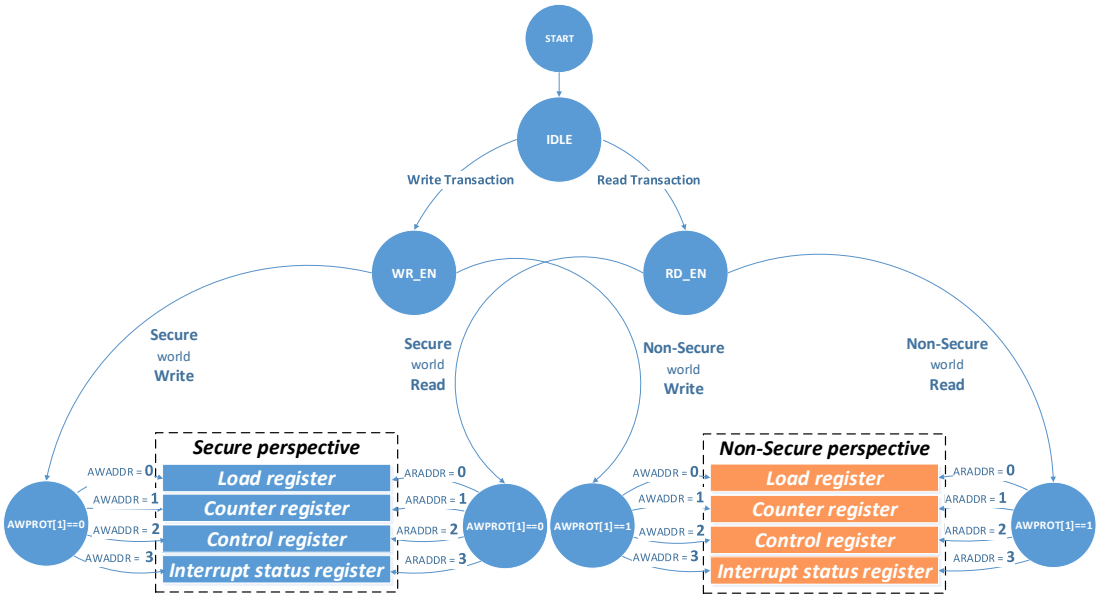


Figure 4.9: Self-Secured Private Timer: Minimal Approach register access flow.

4.2.3 Self-Securing the Private Timer: Default Approach

Differently, in the default approach, illustrated in Figure 4.10, the secure world can access both the secure and non-secure interfaces. Although the secure world is now able to use both virtual timer interfaces, the non-secure world is still unable to tamper with the timer's normal flow and values of the secure interface, used by secure applications. In contrast to the minimal approach, not only the main internal counter and load registers need to be replicated, but also their respective AXI registers. Differently, the ISR and Controller AXI registers are not replicated, similarly to the minimal approach, where these registers are only extended and partially replicated internally. The previously described behavior implies changes in the device AXI register map, as shown in Table 4.2, providing access to the non-secure extra interfaces.

Table 4.2: Self Secured Private Timer: Default Approach register map.

Baseaddress (0x43C00000) + Offset	Type	Name
0	RW	Private Timer Load Register
4	RW	Private Timer Counter Register
8	RW	Private Timer Control Register
12	RW	Private Timer Interrupt Status
16	RW	Non-Secure Private Timer Load Register
22	RW	Non-Secure Private Timer Counter Register

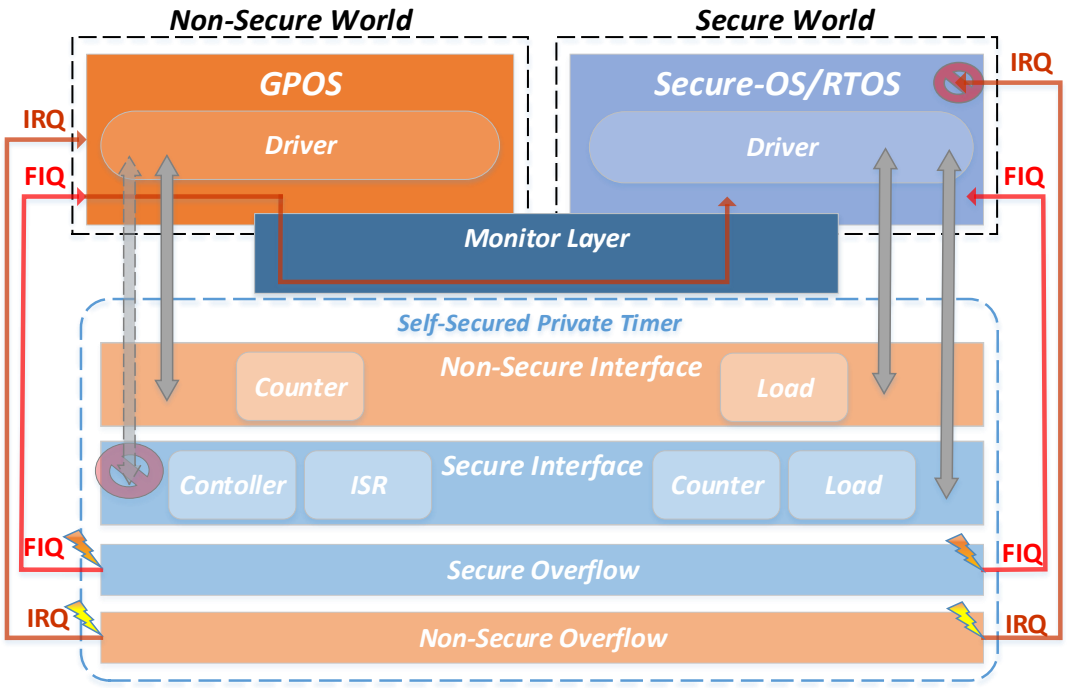


Figure 4.10: Self-Secured Private Timer: Default Approach architecture.

Since the secure world can now access both interfaces, a method to control them simultaneously was implemented, hence two additional control bits were added to the control register to configure and manage the non-secure interface (auto_reload and enable bit for non-secure), as depicted in Figure 4.11. This enables the secure world to configure and manage both the secure and non-secure timer interfaces, while still restricting the non-secure world to access the above mentioned secure control parameters and secure interface configurations. Interrupts and interrupt event flags are managed the exact same way as the minimal approach.

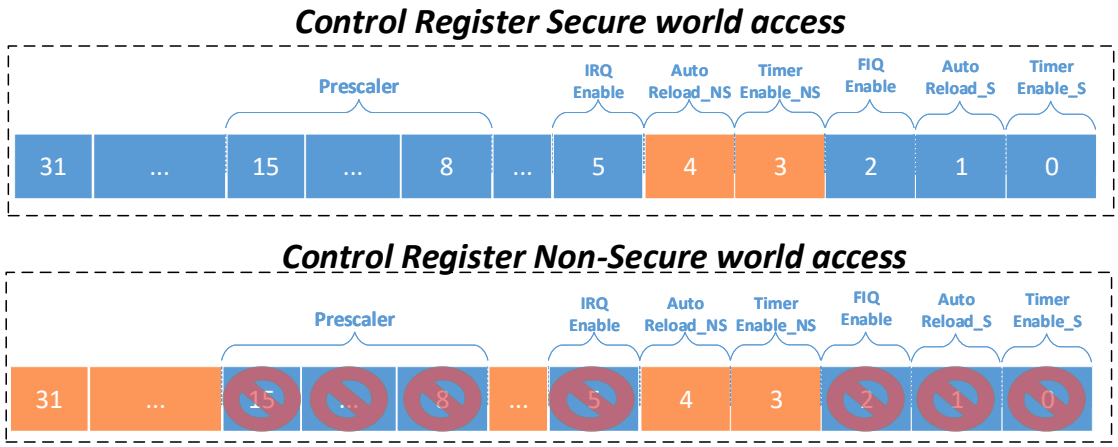


Figure 4.11: Default Approach control register.

Upon a read/write transaction, from the secure world perspective it can see/access the non-secure interface registers, and consequently take advantage of both interfaces. Nevertheless, the non-secure world can only access the secure interface registers, and any access to a secure interface register is denied through the AWPROT/ARPROT TrustZone extended protection signals. This behavior is illustrated in Figure 4.12.

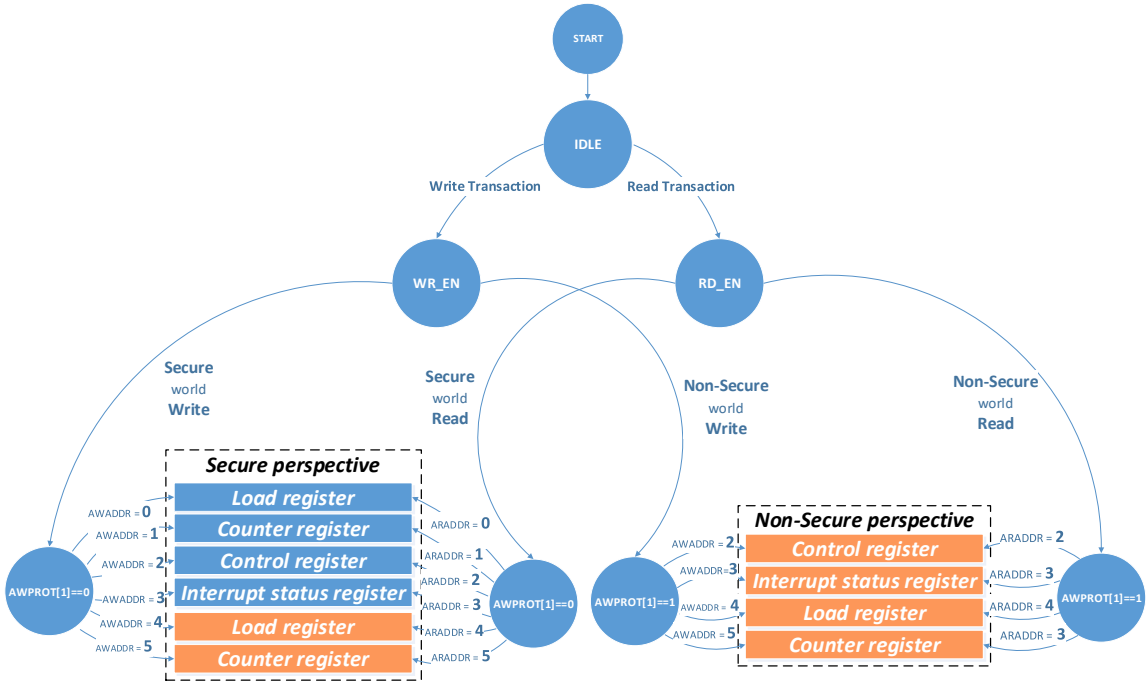


Figure 4.12: Self-Secured Private Timer: Default Approach register access flow.

4.2.3.1 Device driver modifications

To enable access from the secure world to both interfaces, and as a consequence of the implicated changes in the AXI registers addresses, some minor modifications to the device drivers must be performed. The changes encompass the following functions:

- PrivateTimer_Set_Load_register_S: Sets the secure banked Load register, exclusively from the secure world.
- PrivateTimer_Set_Load_register_NS: Sets the non-secure banked Load register.
- PrivateTimer_Get_Load_register_S: Reads the secure banked Load register, exclusively from the secure world.
- PrivateTimer_Get_Load_register_NS: Reads the non-secure banked Load register.
- PrivateTimer_Set_Counter_register_S: Sets the secure banked counter register, exclusively from the secure world.
- PrivateTimer_Set_Counter_register_NS: Sets the non-secure banked counter register.
- PrivateTimer_Get_Counter_register_S: Reads the secure banked counter register, exclusively from the secure world.
- PrivateTimer_Get_Counter_register_NS: Reads the non-secure banked counter register.

4.3 Self-Secured UART

The Cadence universal asynchronous receiver-transmitter [DD03], available in Zynq-7000 SoC, was chosen as a medium-complexity device for implementing the self-secured approach. Used for serial data communication the UART provides full-duplex asynchronous receiver and transmitter. The device is structured with separate Receiver (Rx) and Transmitter (Tx) data paths, each with a dedicated 64-byte FIFO, which contains the data that is serialized/de-serialized, within the provided following features:

- Programmable baud rate generator;

- Receive and transmit FIFOs with 64 bytes;
- Programmable protocol, 6/7/8 data bits, 1/1.5/2 stop bits, and odd/even/space/mark/no parity;
- Error detection for parity, framing and overrun;
- Line-break and interrupt generation;
- Multiple opertaion modes por RxD an TxD (e.g. normal, echo, diagnostic loopback).
- Model control for modem control signals.

The device was replicated in hardware, by implementing six main modules: (i) Control and Status; (ii) Baud-rate Generator; (iii) Transmitter and transmitter's FIFO; (iv) Receiver, and receiver's FIFO; (v) Mode switch; and (vi) Modem control. Figure 4.13 depicts the UART block diagram.

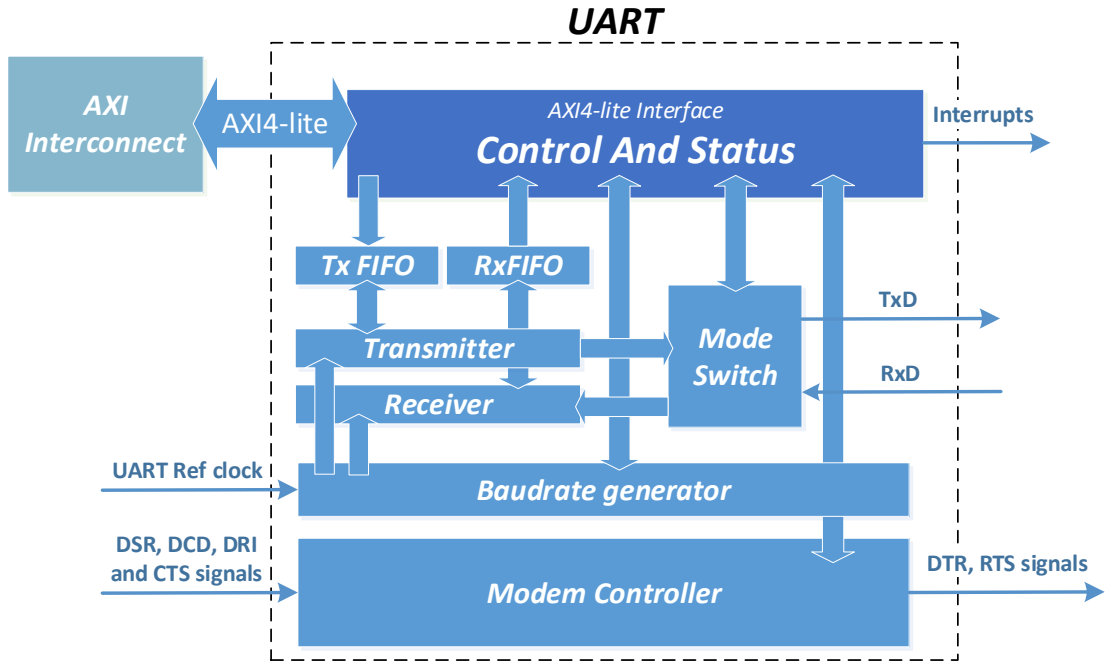


Figure 4.13: UART block diagram.

4.3.1 Control and Status Module

The Control and Status module is composed of the whole UART's seventeen AXI registers, shown in Table 4.3. This Module is the device's control unit, responsible for managing all the modules through the AXI-lite interface, some of its main tasks are: configure UART's operation mode, baud rate, interrupts, and

data format; store and read characters in the Tx/Rx FIFOs; enable, disable and issue soft resets to the receiver and transmitter; manage the generated interrupts; and configure user-defined parameters (e.g. FIFOs trigger level, baud rate divider).

Table 4.3: UART register map.

Base address (0x43C00000) + Offset	Type	Name
0	RW	Control register
4	RW	Mode register
8	RW	Interrupt enable register
12	RW	Interrupt disable register
16	RO	Interrupt mask register
20	RW	Interrupt status register
24	RW	Baudrate generator register
28	RW	Receiver timeout register
32	RW	RxFIFO Trigger value register
36	RW	Modem Control register
40	RW	Modem status register
44	RO	Channel status register
48	RW	TxFIFO register
52	RW	Baud rate divider register
56	RW	Flow delay register
60	RW	TxFIFO Trigger value register
64	RW	RxFIFO register

The control register (Figure 4.14), controls the transmitter and receiver modules through its eight less significant bits. Each bit represents a control signal that allows enabling, disabling, and resetting the transmitter and receiver modules. Furthermore, it allows to introduce transmission breaks in the transmitter module and to reset the timeout counter of the receiver module.

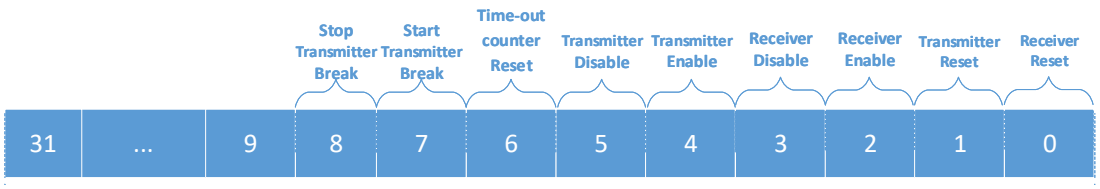


Figure 4.14: UART Control register layout.

The mode register illustrated in Figure 4.15 allows to configure: the UART operation mode; the data format, by defining the received/sent data number of stop bits, parity type, and number of data bits (i.e. character length); and the selected clock source of the baud rate generator module.

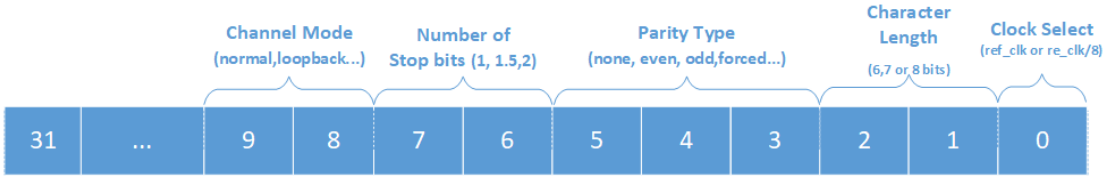


Figure 4.15: UART Mode register layout.

The enable/disable/mask interrupt registers (Figure 4.16), present the same bit disposition and are used to configure the desired interrupts. These include: interrupts generated by the receiver and transmitter modules; FIFOs indicating their status, such as full, nearly full, empty, and above the trigger value; and receiver interrupts issued upon (i) the received character not matching the configured parity (parity interrupt), (ii) the received character not matching the number of configured stop bits (framing interrupt), (iii) the reception of another character before the previous reception is over (overflow interrupt), and (iv) exceeding the programmed amount of time waiting for data reception (time-out interrupt). The timeout interrupt value is configured in the receiver timeout register, and the trigger level of the Tx and RxFIFOs are respectively configured through the TxFIFO trigger value and RxFIFO trigger value registers.

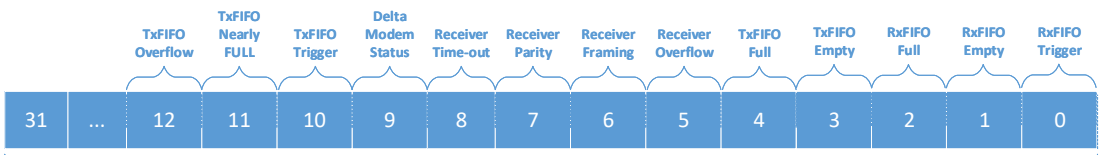


Figure 4.16: UART Interrupt enable/disable/mask registers layout.

The channel status register provides continuous monitoring of the TxFIFO and RxFIFO levels and trigger status, the transmitter/receiver state machines status (active or inactive), and the flow delay trigger status.

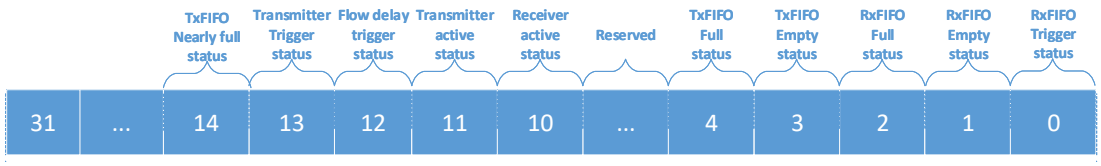


Figure 4.17: UART Channel status register layout.

One of the Control and status module main functionalities is to receive data from the AXI interconnect and store it into the TxFIFO, allowing the UART transmitter module to retrieve the data from the respective FIFO, and transmit it over to the terminal (Tx). This behavior, shown in Listing 4.1, is achieved by

connecting the AXI register into the TxFIFO and enabling the write enable whenever this register is written. The written enable should be cleared immediately in the following clock cycle in order to prevent the same value getting written into the FIFO more than once.

```
if (slv_reg_wren) begin
case (awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
5'h0C:
for(byte_index = 0;byte_index<=(C_S_AXI_DATA_WIDTH/8)-1;byte_index=byte_index+1)
// TxFIFO AXI register being written
  if ( S_AXI_WSTRB[byte_index] == 1 ) begin
      slv_reg12[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
      if(wr_en_flag ==  0 && byte_index <= (C_S_AXI_DATA_WIDTH/8)-1) begin
        Tx_fifo_wr_en <= 1; wr_en_flag <= 1;//write enable signal issued
      end
  end
```

Listing 4.1: TxFIFO write enable upon AXI TxFIFO register write. Verilog code extract.

Among the main functionalities is the reverse operation, reading the received characters from the UART, which were stored into the RxFIFO. Upon issuing a read of the respective AXI register (RxFIFO), the read enable signal is activated for one clock cycle and the returned FIFO value stored into the respective read AXI register, as depicted in Listing 4.2.

```
if(!rd_en_flag &araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB]==5'h10& slv_reg_rden)
// RxFIFO AXI register being read
     begin Rx_fifo_rd_en <= 1;rd_en_flag <= 1; end//read enable signal issued
if(rd_en_flag) begin Rx_fifo_rd_en <= 0;  rd_en_flag <= 0; end
if (slv_reg_rden) axi_rdata <= reg_data_out; //read selected axi register data
end
```

Listing 4.2: RxFIFO read enable upon AXI RxFIFO register read. Verilog code extract.

Interrupt management is another task of the control and status module, which checks the enable, disable, and mask registers of each interrupt and detects the raising edge of each generated interrupt by the other modules. If all this conditions are identified, the generated interrupt is let through to the PL interrupt ports until it is cleared in the respective bit of the interrupt status register, as demonstrated in Listing 4.3.

```
//Checks enable/disable/mask
assign ir_rxovr = (slv_reg2[0] && !slv_reg3[0] && slv_reg4[0]) ? 1 : 0;
//Checks for the interrupt rising edge
```

```
assign pos_rxovr = (prev_rxovr == 0 && i_slv_reg11[0]) ? 1 : 0;
...
if(slv_reg_wren & (awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB]==5'h05))begin
//interrupt status register modified, interrupt can be cleared
for(byte_index=0;byte_index<=(C_S_AXI_DATA_WIDTH/8)-1;byte_index=byte_index+1)
  if ( S_AXI_WSTRB[byte_index] == 1 ) begin ...
      slv_reg5[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
  end
end else begin
//updates interrupt status upon trigger and after clear
slv_reg5[0] <= (ir_rxovr & pos_rxovr)? 1 : slv_reg5[0]; ...
end
```

Listing 4.3: Example of overflow interrupt management on the Control and status module. Verilog code extract.

4.3.2 Baud rate generator Module

The baud rate generator provides the receiver and the transmitter modules with a clock source. This module uses the general clock source and based on the introduced user baud rate configurations parameters generates the baud rate with the appropriate frequency. The required user-defined parameters are set at the baud rate generator and baud rate divider registers, the reference clock frequency is divided by these parameters, as shown in Figure 4.18. Moreover, two independent baud rate signals are provided, for each the transmitter and receiver modules. This is required to perform the baud rate synchronization by the receiver module whenever the data transition begins, as explained later on.

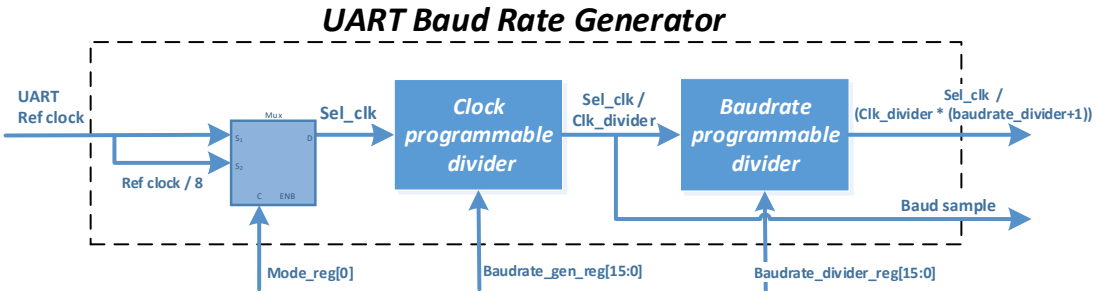


Figure 4.18: UART Baudrate generator.

4.3.3 Transmitter and transmitter FIFO modules

The transmit FIFO (TxFIFO) stores the data from the AXI TxFIFO register, up to eight bits upon a write transaction. When data is written, the empty flag

is cleared, remaining low until all the data is removed and sent by the transmitter. This FIFO provides event and interrupt flags such as, TxFIFO full interrupt status (TFULL), TxFIFO nearly-full flag (TNFULL), threshold trigger (TTRIG), that indicate if the FIFO is full, nearly full (one more write left), or reached the programmed fill level, respectively.

The Transmitter gets the data from the TxFIFO and serializes it, Figure 4.19 illustrates this module state machine. Firstly, the TxFIFO level is verified, and if it is not empty a read enable to the TxFIFO is issued and the start bit is sent. If the data was successfully read from the FIFO (TxFIFO valid signal high) the data bit transmission is started, otherwise the transmission is restarted. Once in the data bits transmission state, the character is serialized and each data bit is sent one by one at the corresponding baud rate. Also, high level data bits are accounted in order to send the parity bit afterwards, if enabled at the device's configurations. When the number of sent data bits reaches the value configured at the mode register, the transmitter state is updated. If the parity bit is enabled at the mode register, the parity of the sent data bit is verified and sent accordingly high/low, for one baud rate clock, otherwise this last step is disregarded. Then, for the stop bit state, the number of configured stop bits at the mode register is transmitted, also at the baud rate.

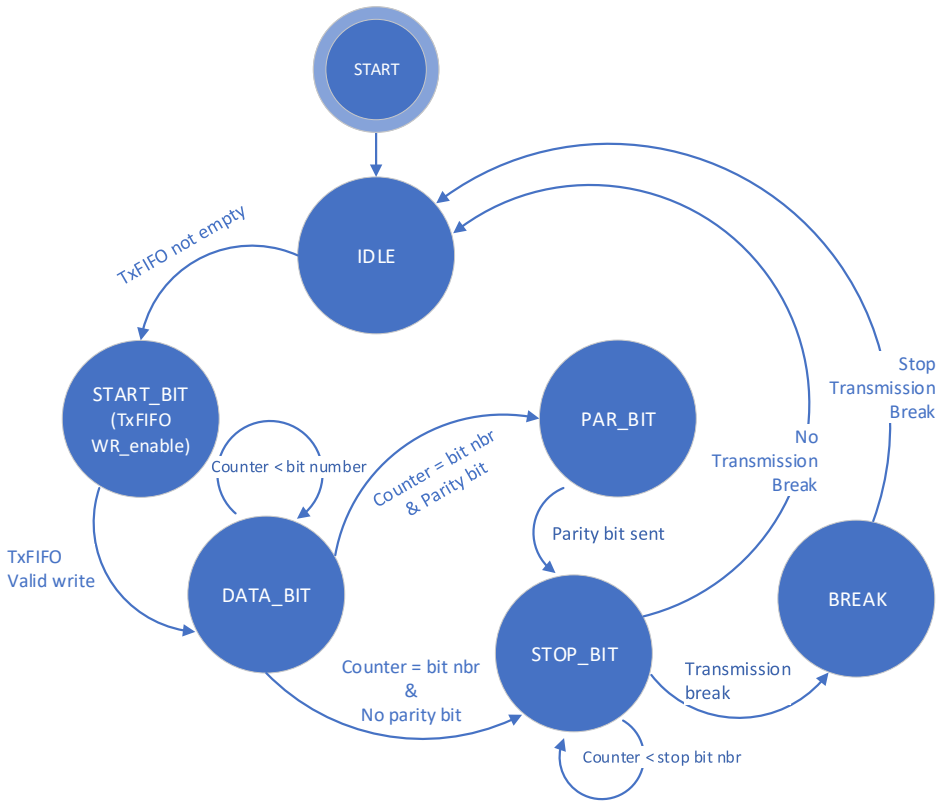


Figure 4.19: Transmitter finite state machine.

Lastly, before returning to idle state and ready to transmit other data, the transmitter checks if a transmission break was issued, and in such case, the transmitter will be locked in a "break" state until a stop transmission break is issued, thereupon returning to the idle state and ready for the next transmission. Otherwise, without receiving any transmitter break the transmitter returns directly to the idle state after the sending the stop bits. Throughout the various transmitter states, the Tx signal is set accordingly to the current state, parity configuration, and current data bit (if in data bit state), as shown in Listing 4.4.

```
1  assign Tx =
2  (state == IDLE)      ?  1'b1 :
3  (state == START_BIT)?  1'b0 :
4  (state == PAR_BIT && mr_par[0] == 0 && mr_par[1] == 0) ?
5  (nbr_of_ones_reg % 2 == 0) :
6  (state == PAR_BIT && mr_par[0] && mr_par[1] == 0) ?
7  (nbr_of_ones_reg % 2 != 0) :
8  (state == PAR_BIT && mr_par[0] == 0 && mr_par[1])  ?  0 :
9  (state == PAR_BIT && mr_par[0] && mr_par[1])  ?  1 :
10 (state == STOP_BIT) ?  1'b1 :
11 (state == BREAK)    ?  1'b0 :
12 (state == DATA_BIT) ?  data_reg[global_counter] : 1'b0;
```

Listing 4.4: Tx signal set according to current state. Verilog code extract.

Figure 4.20 illustrates a complete transmitted data stream synchronized with the respective baud rate, and configured with a eight data bit length, parity bit and one stop bit.

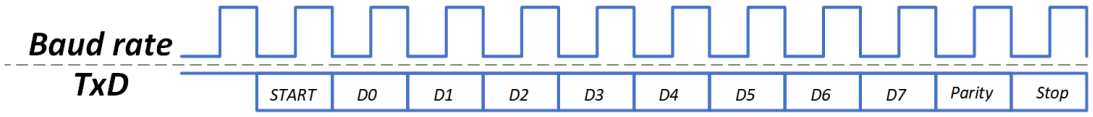


Figure 4.20: Transmitter data stream.

4.3.4 Receiver and receiver FIFO modules

The receiver FIFO (RxFIFO) stores data up to eight bits, from the receiver mode. This FIFO provides event and interrupt flags such as, RxFIFO full interrupt status (RFULL) and the threshold trigger (RTTRIG) that indicate if the FIFO is full or reached the programmed fill level, respectively.

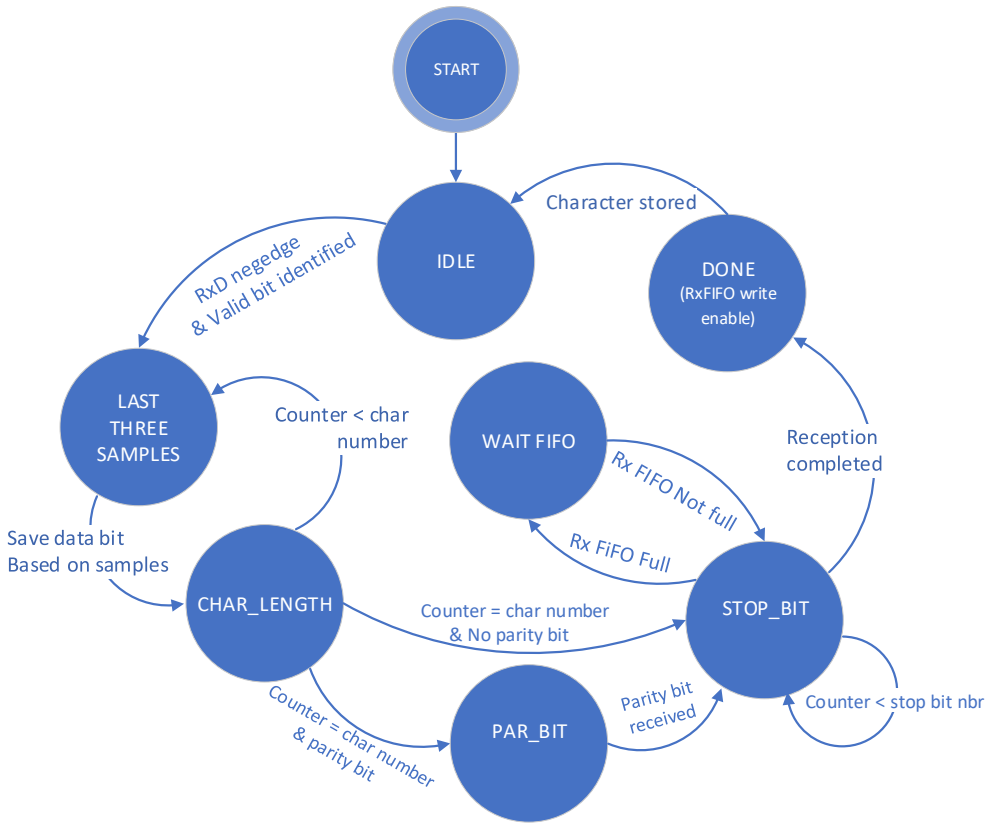


Figure 4.21: Receiver finite state machine.

The receiver module is responsible for continuously over-sampling the RxD signal, gathering the data which is being serialized, and storing the received UART data. This state machine is illustrated in Figure 4.21. When a sample detects the RxD transition to low level it waits for half of the configured baud rate divider value, collects three more samples and verifies if the RxD signal remained low. If the signal remained low until this point, the receiver considers it as a valid start bit, otherwise, the same process is performed in the next RxD negative edge. Upon a valid start bit detection, the receiver baud rate clock is resynchronized in order to collect three samples around the data bit mid-point, as shown in Figure 4.22.

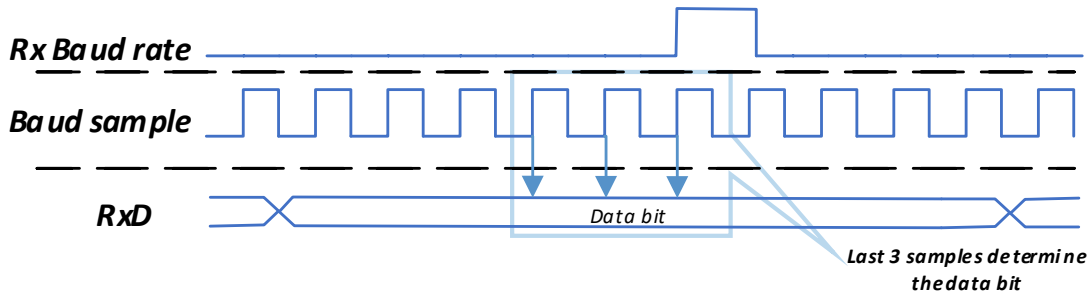


Figure 4.22: Resynchronized baud rate at data bit mid-point.

When the resynchronized baud rate is high, the last three samples are collected and the selected data bit is determined by majority voting, and the number of high level selected bits is recorded for parity checking purposes. The former process is repeated at a specific baud rate, until the number of selected bits meets the number of characters configured in the mode register. Then, the receiver enters the stop bit state, receiving the number of expected stop bits, also configured in the mode register. After receiving the stop bits, if the RxFIFO has space for the received data, the write enable signal is activated and the data stored in the FIFO. Otherwise, if the FIFO is full, the receiver waits for available space to store the data in the FIFO. However, if a new valid start bit arises when the receiver is waiting for FIFO space, the assembled character is dumped and the overflow interrupt issued.

In addition to the overflow interrupt, the receiver module also generates other already mentioned interrupts, such as: parity interrupt, when the parity of the received data bits is calculated, in accordance with the respective mode register bit field, and does not match the received parity bit; framing interrupt, when the received number of stop bits counter doesn't match the expected number (configured on mode register); and the timeout interrupt, when a global counter that keeps track of the amount of time since the receiver is in idle state waiting for a valid start bit exceeds the programmed value, configured in the respective AXI register.

4.3.5 Mode switch module

The mode switch module uses the mode register configuration bit to control the RxD and TxD signals routing. This module enables the UART to operate in several modes show in Figure 4.23, such as: (i) normal mode, the standard for UART operations, where the receiver and Rx and Tx signals pass-through the mode switch module, directly to the respective RxD and TxD pins; (ii) automatic echo mode, where the received data from the RxD pin is routed to both the receiver module and the TxD Pin, immediately transmitting what is being received and stored; (iii) local loopback mode, where both the RxD or TxD pins are unconnected, instead the transmitter Tx output signal is directly connected to the receiver Rx signal (normally used for test purposes); (iv) remote loopback mode, where the RxD and TxD pins are connected to each other and the UART cannot transmit or receive any data, consequently received data is directly transmitted back.

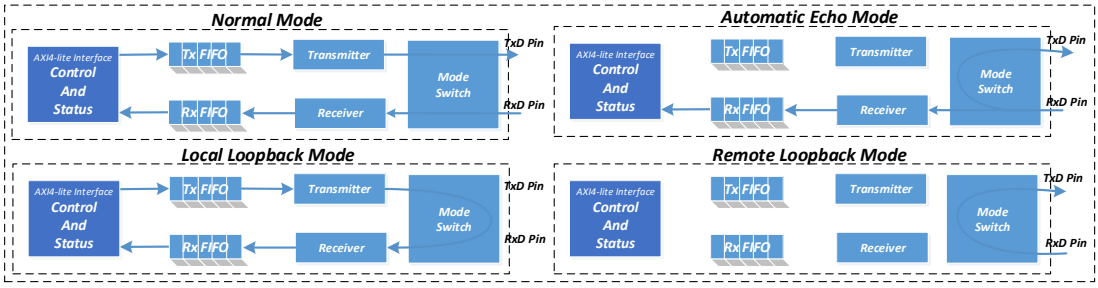


Figure 4.23: UART operation modes.

4.3.6 Modem control module

The modem control module is used to manage communication between the UART and a modem. The module has two associated registers, illustrated in Figure 4.24: The modem status register, contains the current status of Delta Clear to Send, Delta Data Set Ready, Trailing-edge Rind indicator and Delta data carrier detect. Whenever one of these modem signals status changes, an interrupt is issued, by setting the interrupt status bit at the DMSI bit in the interrupt status register. The modem control register sets the data terminal ready (DTR) and request to send signals, and is able to configure the flow control mode as either automatic or Manual. The flow control mode is by default manual, hence the request to send (RTS), and data terminal ready (DTR) are completely controlled by the modem control register. However, if the flow control mode is configured as automatic, these signals are asserted and de-asserted based on the current FIFO level and configured flow delay register, and transmission in only possible when the clear to send signal is asserted.

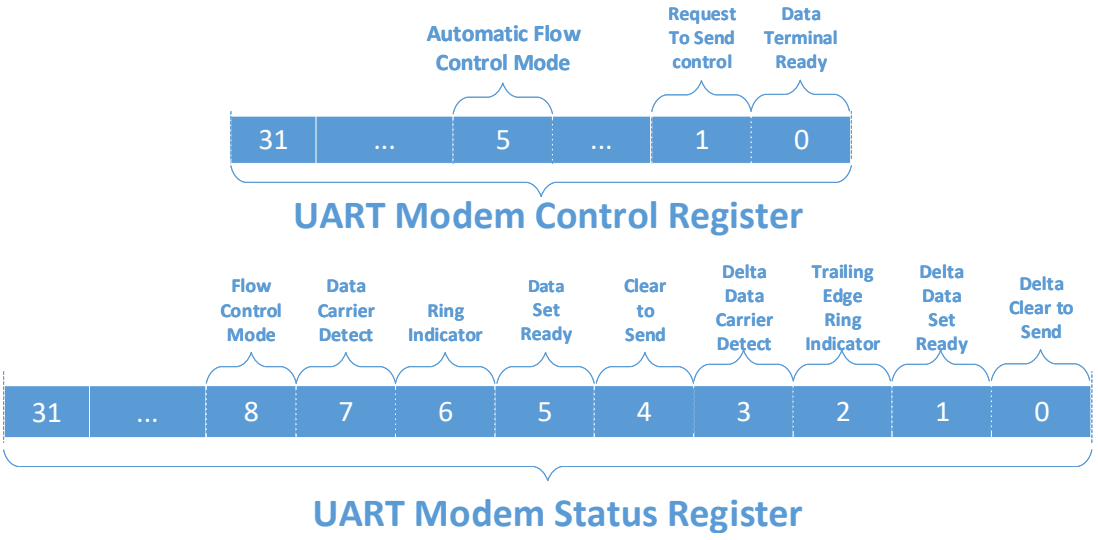


Figure 4.24: UART modem registers layout.

4.3.7 Device driver

To access the device's functionalities, a software interface to the hardware devices was implemented, enabling both the GPOS and the RTOS facilitated access to hardware functions through an abstraction layer. Therefore, to manage the UART by accessing the respective AXI registers, the following functions are provided:

- UART_SetBaudrate: calculates and sets the most adequate baudrate given the input clock and baudrate divider. Also, resets the transmitter and receiver modules and calls the enable function;
- UART_Init: sets up the device data format, operation mode, FIFOs triggers to default values, disable all interrupts and calls the function responsible for configuring interrupts;
- UART_Enable: enables the transmitter and receiver modules on the control register and disables stop breaks;
- UART_Disable: disables the transmitter and receiver modules on the control register and enables stop breaks;
- UART_Reset: disables all interrupts and the receiver and transmitter modules, resets the transmitter and receiver modules consequently clearing all FIFOs, clears status flags and restores FIFO trigger levels and transmitter and receiver to default reset value;
- UART_SelfTest: disables all interrupts, sets the device in local loopback operation mode, sends a full string with "puts" function and compares it with the received string by the receiver, if the comparison returns valid means the device is working correctly, thus, the self test returns successful;
- UART_getc: attempts to read the character from the receiver FIFO by reading the RxFIFO AXI register;
- UART_putc: attempts to write a character to the secure transmitter FIFO by writing the character into the TxFIFO AXI register;
- UART_puts: attempts to write a string to the transmitter FIFO, by sequentially writing the string's constituent characters into the TxFIFO AXI register;

4.3.8 Self-Securing the UART

Following the self-secured concept, the device logic is divided into two separate interfaces composed of different banked registers: the secure interface, exclusively accessible by the secure world and the non-secure interface, accessible by both the secure and non-secure worlds. Most of the required modifications to the original hardware logic are confined to the AXI peripheral and registers within the control and status module. The vulnerable logic part of the device, the secure interface, must be isolated from the non-secure world., This includes registers that may:

- Compromise data validity, as the mode register, where data format is set;
- Trigger unexpected and unintended interrupts by changing interrupt configurations registers, data formats, timeout configurations, operation modes and so on;
- Change devices configuration, which are normally performed at boot time (baud rate and data formats configurations);
- Tamper with the device's normal flow by changing the operation mode of the device through mode and modem control registers (normal, automatic echo, local/remote loopback and flow control modes) or also by transmitting breaks to the receiver through the modem register;
- Contain sensible and secure data that should not be accessible by the non-secure world as data being sent over by the UART while in the secure world or even received secure data. This type of data is stored in the secure transmit and receive FIFOs respectively, present in the secure bank.

Therefore, every AXI register was individually evaluated taking these proprieties into consideration and classified as secure or non-secure registers. Secure banked registers exclusively belong to the secure interface, unlike non-secure registers which are be replicated to the non-secure registers bank, allowing access from both interfaces (Figure 4.25).

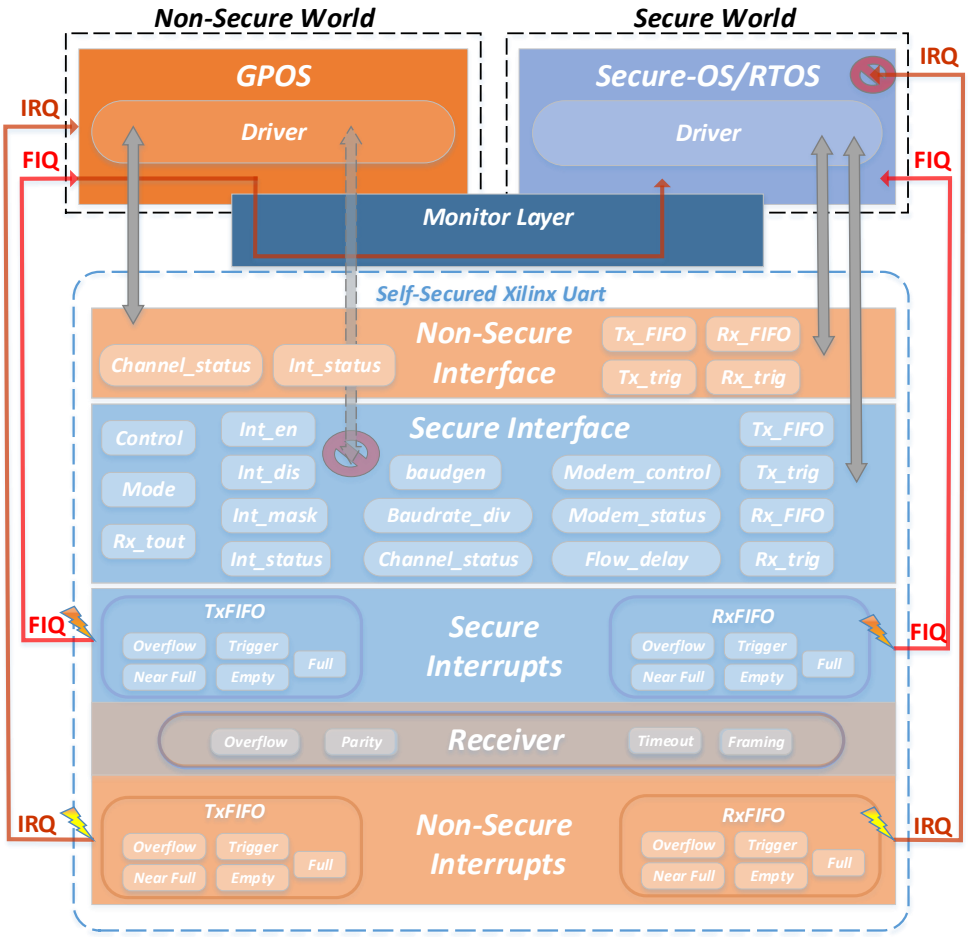


Figure 4.25: Self-Secured UART architecture.

According to Figure 4.25, the exclusive registers from the secure interface encompass the:

- Control register, due to the possibility of compromising UART's main modules by changing configurations such as enable, disable, reset, and break transmissions of the transmitter and receiver modules, shared by both the secure and non-secure interfaces;
- Mode register, sets the transmitted and received data format. If the data format is modified during a transmission/reception, data validity cannot be guaranteed. Data format should be set at configuration time, accordingly to the terminal configuration. If this register is changed by the non-secure world and mismatches terminal configurations, the data validity would be compromised and unintended interrupts could be triggered;
- Interrupt enable/disable/mask registers, used to enable/disable UART's interrupts. Such configurations are normally performed at boot time, and if

modified at any other moment could trigger unintended interrupts. So, the non-secure world should not be able to access it;

- Baud rate generator register, contains the value by which the reference clock is divided to generate the desired baud rate and baud sample. It is also typically set at configuration/boot time, accordingly to the terminal configuration. Hence, if this register is modified and mismatches the terminal configurations, the data validity is compromised;

- Baud rate divider register, contains the value by which the baud sample is divided to generate the desired transmitter and receiver baud rates. Thus, it should also be secured, as the previous register.

- Receiver timeout register, enables the UART to detect an idle condition on the receiver data line. The timeout value indicates the maximum delay for which the UART should wait for a new character to arrive, before issuing a timeout interrupt. Therefore, should only be changed by the secure world, otherwise an unintended timeout interrupt on the receiver module might be triggered;

- Modem control register, controls the interface with the modem. The secure world should be exclusively capable of altering UART's operation mode and setting automatic or manual flow control, due to the possibility of disrupting the device;

- Flow Control Delay register, only used if enabled in the modem control register, and specifies the receiver FIFO level at which the terminal request to send signal (RTS) is asserted/de-asserted. Such as the modem control register it should only be accessible through the secure interface, since it modifies the UART operation mode;

Also according to Figure 4.25, the registers from the non-secure interface encompass the:

- Modem status register, indicates the current state of the control lines of the modem. Since this register does not break any of the above properties, it can be accessed by both worlds without being replicated.

- Interrupt status register, indicates any interrupt event that has occurred since this register was last cleared. This register must be replicated to the non-secure register bank, so that non-secure interrupts status can also be set

and cleared by the non-secure world, while protecting the secure interrupts status;

- Status register, enables the continuous monitoring of the raw unmasked status information of the UART. This register must be replicated to both register banks so that both the secure/non-secure FIFOs status are accessible by the respective interfaces;

- Transmit FIFO, data written to this register is stored into the respective FIFO in order to be sent over by the transmitter. A copy of this register is required in both the secure and non-secure interfaces, preventing secure data from being accessed or non-secure data from being stored in the secure FIFO.

- Receiver FIFO, contains the last data read from the receiver FIFO. A copy of this register is required in both the secure and non-secure banks, preventing secure data from being read through the non-secure interface or non-secure data from being stored into the secure FIFO.

- Transmitter/Receiver FIFO Trigger Level registers, used to set the value at which the receiver and transmitter FIFOs trigger an interrupt event. These registers must be replicated, such as both FIFOs, so that both the secure and non-secure receiver FIFOs level trigger can be respectively configured;

Following the same interrupt model as the timer use case, secure and non-secure interrupts must be differentiated. In this sense, whenever the receiver incoming data source is the secure terminal port, the receiver interrupts are routed as fast interrupt requests (FIQs) to the secure world. Otherwise, if the source is the non-secure terminal, interrupts are routed as interrupt requests (IRQs) to the non-secure world. Differently, interrupts upcoming from the non-secure FIFOs are exclusively routed as IRQs and interrupts upcoming from the secure FIFOs are exclusively routed as FIQs.

Summing up, the non-secure bank accessible by both interfaces contains the non-secure FIFOs and their associated triggers level registers, as well as other status registers. Non-secure data is only stored into the non-secure FIFOs and unable to be stored into the secure FIFOs. Both register banks are mapped into the AXI-lite peripheral address space, shown in Table 4.4.

Table 4.4: Self-Secured UART register map.

Base address (0x43C00000)	+	Offset	Type	Name
Secure Register Bank		0	RW	Control register
		4	RW	Mode register
		8	RW	Interrupt enable register
		12	RW	Interrupt disable register
		16	RO	Interrupt mask register
		20	RW	Interrupt status register
		24	RW	Baudrate generator register
		28	RW	Receiver timeout register
		32	RW	RxFIFO Trigger value register
		36	RW	Modem Control register
		40	RW	Modem status register
		44	RO	Channel status register
		48	RW	TxFIFO register
		52	RW	Baud rate divider register
		56	RW	Flow delay register
		60	RW	TxFIFO Trigger value register
		64	RW	RxFIFO register
Non-Secure Register Bank		68	RW	NS RxFIFO Trigger value register
		72	RW	NS TxFIFO Trigger value register
		86	RW	NS RxFIFO register
		80	RW	NS TxFIFO register
		84	RW	NS Interrupt status register
		88	RW	NS Channel status register

The behavior of the AXI registers access upon read/write operations from both the secure and non-secure interfaces is illustrated in Figure 4.26. When accessing the UART from the secure world, all banked registers can be accessed.

Although, not all are illustrated in Figure 4.26 (extensive list). Differently, from the non-secure world perspective, only the non-secure register bank is accessible. As previously explained and hereby illustrated, write accesses are restricted based on the AWPROT TrustZone extended protection signal and read accesses restricted through the ARPROT signal.

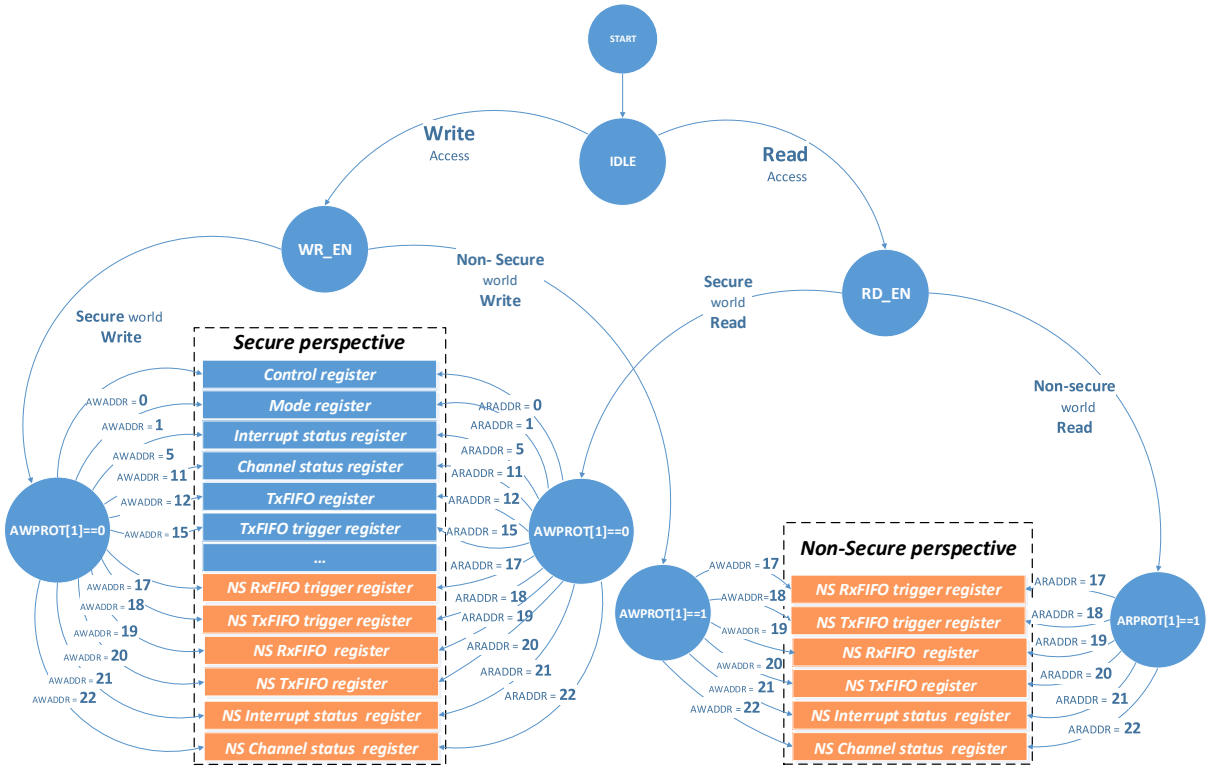


Figure 4.26: Self-secured UART register access flow.

In addition to the register and control signals modifications to self-secure the device, some minor modifications are still required at the UART's main modules used by both interfaces. These modifications, shown in Figure 4.27, encompass the previously mentioned duplication of receiver and transmitter FIFOs (data isolation), and minor modifications to the receiver and transmitter modules to store the data in the respective FIFOs, accordingly to the terminal data source security. Given the time-criticality of secure applications using the UART through the secure interface, secure data transmission and reception must be prioritized.

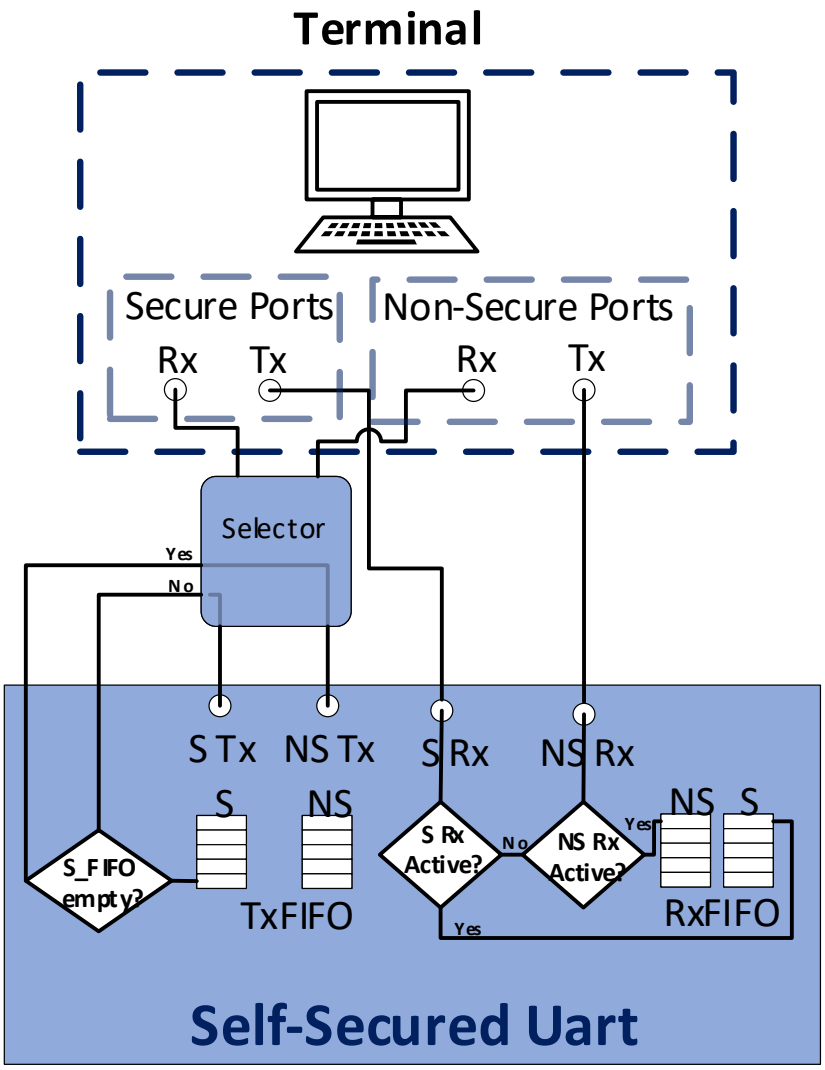


Figure 4.27: Self-Secured UART application example.

At the transmitter level, secure data transmission prioritization is ensured by only transmitting non-secure data when the secure FIFO is completely empty. Whenever the transmitter is in idle state, it verifies if a read request has already been issued to the Transmitter FIFO and validated, through the valid TxFIFO signals. If so, the transmitter changes to the next state to begin transmitting the data and the output Tx signal is selected according to the data source security (secure or non-secure FIFO). Otherwise, if no read request has been issued to both TxFIFOs yet, the transmitter verifies which FIFO has available data to be transmitted, through the FIFOs empty status signals. Since the secure FIFO data is prioritized, the transmitter starts by checking the secure FIFO for available data. If there is available data to read in this FIFO the read enable signal is issued, for one active clock, to the secure FIFO. However, if the secure FIFO is

empty, the transmitter further checks for available data on the non-secure FIFO. If data is available for transmission on the non-secure FIFO, the enable signal is issued to the non-secure FIFO instead. Once the transmission is completed, the transmitter waits in idle state for a new FIFO read operation to be validated, in order to proceed to the next transmission state, with the Tx signal selected according to the transmission security state. Listing 4.5 shows the aforementioned behavior Verilog implementation.

```
1  if(rd_flag) begin TxFIFO_rd_en_S <=0; TxFIFO_rd_en_NS <=0; end
2  case (state) IDLE: begin
3      if(!(TxFIFO_valid_S | TxFIFO_valid_NS) & !valid_flag) begin
4        if(!TxFIFO_empty_S)begin
5           rd_flag <= 1; TxFIFO_rd_en_S = 1; tx_state <= 1; end
6        else begin if (!TxFIFO_empty_NS)begin
7           rd_flag <= 1; TxFIFO_rd_en_NS =1; tx_state <= 0; end
8      end end
9    if(TxFIFO_valid_S) begin
10       data_reg <= i_data_S;
11       valid_flag <= 1;end
12   if(TxFIFO_valid_NS) begin
13       data_reg <= i_data_NS;
14       valid_flag <= 1;end
15
16   if(valid_flag & baudrate_Tx_neg)begin
17       state <= START_BIT;
18       rd_flag <= 0;
19       valid_flag <= 0;end
20 end ....
21 assign Tx_S =  (tx_state) ? Tx : 1'b1;
22 assign Tx_NS = (tx_state) ? 1'b1 : Tx;
```

Listing 4.5: Secure data transmission prioritization. Verilog code extract.

At the receiver module, the security of the receiving data is ensured by continuously monitoring of both the secure and non-secure RxD signals, at the specific baudsample frequency. Upon a start bit detection from the secure RxD, the receiver prepares for a secure data reception. Whether if the receiver is in idle mode or already receiving non-secure data, the new data is prioritized. In the latter case, the receiver dumps the non-secure data and forces the receiver to restart and receive the secure data immediately, unless the data being received is already secure. Otherwise, if the secure reception has already been performed, while in idle mode the receiver is able to start receiving the non-secure data.

Receiver's parity, framing and timeout interrupts are duplicated, enabling receiver interrupts to be routed as either FIQs or IRQs according to security source of the data being received whenever an interrupt is triggered. Listing 4.6 shows the aforementioned behavior Verilog implementation.

```
assign RxD = (S_STATE) ? RxD_S : RxD_NS ;
assign i_RxD_S_negedge = (i_RxD_S_previous && RxD_S == 0) ? 1 : 0;
assign i_RxD_NS_negedge = (i_RxD_NS_previous && RxD_NS == 0)? 1:0;
if(baudsample)begin// iRxD NEGEDGE DETECTION
  ...
  if(!S_STATE)begin
       if(i_RxD_S_negedge)begin
       //dump NS data and restart S reception
       ...//clean global counters
       state <= IDLE;
       first_negedge <= 1;
       S_STATE<= 1;end
     else if (i_RxD_NS_negedge)  first_negedge <= 1;
    end
    if(state!=IDLE & state!=WAIT_FIFO) first_negedge<= 0;
end
case (state) ...
//After detecting the negedge, the start bit must be validated
  IDLE: if(first_negedge) ... if(start_bit) state <= next_state;
  DONE: ... S_STATE = <0; state <= IDLE; endcase
assign NS_RxFIFO_wr_en = (state == DONE & !S_STATE) ? 1 : 0;
assign  S_RxFIFO_wr_en = (state == DONE &  S_STATE) ? 1 : 0;
```

Listing 4.6: Secure data reception prioritization. Verilog code extract.

4.3.8.1 Device Driver Modifications

To provide access from the secure world to both the secure and non-secure interfaces, and as a consequence of the implemented changes in the AXI registers addresses, some minor modifications to the device driver were performed. These include the following functions:

- UART_getc, attempts to read the character from the secure receiver FIFO by reading the RxFIFO AXI register of the secure register bank, accessible exclusively through the secure interface;
- UART_get_NS, attempts to read the character from the non-secure receiver FIFO by reading the NS_RxFIFO AXI register of the non-secure register bank, accessible through both interfaces;

- UART_putc, attempts to write a character to the secure transmitter FIFO by writing the character into the TxFIFO AXI register of the secure register bank, accessible exclusively through the secure interface;
- UART_putc_NS, attempts to write a character to the non-secure transmitter FIFO by writing the character into the NS_TxFIFO AXI register of the non-secure register bank, accessible through both interfaces;
- UART_puts, attempts to write a string to the transmitter FIFO, by sequentially writing the string's constituent characters sequentially into the secure TxFIFO AXI register, accessible exclusively through the secure interface;
- UART_puts_NS, attempts to write a string to the transmitter FIFO, by sequentially writing the string's constituent characters sequentially into the non-secure TxFIFO AXI register, accessible through both interfaces;

The device driver used in both the secure and non-secure OSs is exactly the same. Modifications at the device driver level are not required, because the security mechanisms for isolating both secure and non-secure accesses are implemented at the hardware level.

4.4 LTZVisor Integration

The Self-secured devices were integrated in an LTZVisor-based system to test all the provided features, while being simultaneously shared by both the secure and non-secure worlds. Targeting a dual-guest OS configuration, the LTZVisor configures as non-secure the required resources for the non-secure guest execution. As previously mentioned, the LTZVisor memory configuration is performed through the TrustZone system level control register (SLCR), which enables memory segments to be configured as either secure or non-secure. Based on this configuration the non-secure guest must be previously compiled to run within the respective assigned memory region. To assign the non-secure region, it is mandatory accessing the SLCR, which needs to be unlocked before changing any of its registers. Thereupon, in Zynq-based devices, through *TZ_DDR_RAM* the seven first memory segments are configured as non-secure.

In order to allow non-secure accesses to propagate to the AXI-lite slaves used by the devices, both *TZ_FPGA_M* and *security_fssw_s0* must be set, enabling non-secure accesses to be propagated through the PL AXI master ports and the general purpose interface (*AXI_GP*), respectively. This allows self-secured devices

to identity the security state of the access, and restrain non-secure accesses through their internal hardware logic.

Moreover, resources required for the Linux OS execution, such as the global timer must be set has non-secure. Other optional resources might also be configured as non-secure, such as the SDIO and QSPI. For instance, it can be used to store the Linux image in these interfaces. Lastly, after configuring the resources security, the SLCR must be locked again with the respective key. If the SLCR registers are left unlocked it would expose them to being accidentally overwritten. These configurations are performed at the board initialization and shown in Listing 4.7.

Listing 4.7: Resources security configuration at board inialitzion.

```
1 uint32_t board_init(void){...
2 /** Unlocking SLCR register */
3 write32( (void *)SLCR_UNLOCK, SLCR_UNLOCK_KEY);
4 /* Handling DDR memory security (first 7segments NS)1 */
5 write32( (void *)TZ_DDR_RAM, 0x0000007f);
6 /* M_AXI_GP0 master security (NS) */
7 write32( (void *)TZ_FPGA_M, 03);
8 /* M_AXI_GP0 slave security (NS) */
9 write32( (void *)SECURITY_FSSW_S0, 0x1);
10 //SCU access control register, contains global timer
11 write((void *) SECURITY_SCU, 0xf);
12 //SCU Non-secure Access Control Register, contains global timer
13 write((void *) SECURITY_NS_SCU, 0xfff);
14 /* SDIO0 slave security (NS) */
15 write32( (void *)SECURITY2_SDIO0, 0x1);
16 /* SDIO1 slave security (NS) */
17 write32( (void *)SECURITY3_SDIO1, 0x1);
18 /* QSPI slave security (NS) */
19 write32( (void *)SECURITY4_QSPI, 0x1);
20 /** Locking SLCR register */
21 write32( (void *)SLCR_LOCK, SLCR_LOCK_KEY);
22 }
```

Nonetheless, for devices configured as non-secures, their respective interrupts should also be configured as non-secure, through the interrupt security configuration (*ICDISRX*) registers of the GIC distributor. Instead, devices configured as secure should have their associated interrupts configured as secure. Likewise, interrupts from the implemented hardware devices should also be set accordingly. Furthermore, the CPU Interface Control Register (*ICCCICR*) is configured, so

secure interrupts are routed as FIQs. These configurations are performed through GIC API at the hardware initialization phase, as shown in Listing 4.8.

Listing 4.8: LTZVisor GIC hardware initial security configuration.

```
1 uint32_t ltzvisor_hw_init(void){
2 /* Config Interrupts Security */
3         interrupt_security_configall();
4 //Self-Secured UART
5         interrupt_security_config(RX_PAR_Intr_NS, Int_NS);
6         interrupt_security_config(RX_PAR_Intr_S, Int_S);
7 // Global Timer
8         interrupt_security_config(27, Int_NS);
9 //Triple Timer Counters
10        interrupt_security_config(TTC1_TTCx_2_INTERRUPT,Int_S);
11        interrupt_security_config(TTC0_TTCx_2_INTERRUPT,Int_S);
12 //ENABLE NON-SECURE INTERRUPTS
13        interrupt_enable(RX_PAR_Intr_NS,TRUE);
14 }
```

4.4.1 FreeRTOS

The chosen OS to run on the secure world was the FreeRTOS, more specifically version 7.0.2. To deploy this OS on the LTZVisor its source code needs to be copied into the LTZVisor secure guest directory. Due to the implementation-defined secure guest and hypervisor compounded compilation into a single image, modifications in the LTZVisor makefile are mandatory, enabling the compiler and linker to add the FreeRTOS files upon the image generation. The directory path of each FreeRTOS added source file and included library must be added to the LTZVisor global makefile, along with the FreeRTOS objects makefile where all the respective output objects that should be generated upon compilation are included.

4.4.1.1 Interrupt management

Modifications on the FreeRTOS source code mainly consisted on: replacing IRQs, so that it can execute using FIQs instead; and adding support for the required FIQ handling (Listing 4.9), which is the secure OS responsibility, according to the LTZVisor interrupt model where all the secure interrupts must be handled by the secure guest.

LTZVisor asymmetrical scheduler depends on FIQ handling to perform partitions scheduling and resume the RTOS tasks. Moreover, the FreeRTOS must

be able to handle the system tick FIQ, triggered by its Triple Timer counter interrupt. To achieve such behavior, at FreeRTOS interrupt setup, the system tick interrupt (previously configured as secure) should: (i) be associated with the respective handler that will increment the RTOS system tick count, run the highest priority task ready, and clear the interrupt; (ii) set the interrupt target; (iii) set the priority level; and lastly, (iv) enable the interrupt. The same process is performed to set up other secure interrupts, such as interrupts from the implemented devices. Another important consideration during secure interrupt configuration is to assign their priority with a higher level than IRQs, in the lower half of the spectrum (ARM interrupt priority scale is inverted). The FreeRTOS interrupts setup is shown in Listing 4.9.

Listing 4.9: FreeRTOS interrupt setup.

```
1 uint32_t prvSetupInterrupt(void){
2         interrupt_enable(TTC1_TTCx_2_INTERRUPT,TRUE);//System tick intterupt
3         interrupt_enable(RX_PAR_Intr_S,TRUE);//UART Secure parity interrupt
4         interrupt_target_set(RX_PAR_Intr_S,0,1);
5         interrupt_target_set(RX_FRAM_Intr_S,0,1);
6         vFreeRTOS_handler_set(TTC1_TTCx_2_INTERRUPT,vTickISR);
7         vFreeRTOS_handler_set(RX_PAR_Intr_S, handler_RX_PAR_Intr_S);
8         interrupt_priority_set(TTC1_TTCx_2_INTERRUPT,6);
9         interrupt_priority_set(RX_PAR_Intr_S,7);
10 }
```

4.4.2 Linux

Differently from FreeRTOS, the non-secure Linux OS is compiled separately and then loaded to the non-secure guest memory during boot phase. Even though the non-secure OS does not need as many modifications as FreeRTOS to run on top of LTZVisor, some minor changes in the device file tree, disabling FIQs, and compiling Linux to the pre-established address in LTZVisor shall be performed. Nonetheless, to deploy Linux in the LTZVisor four main components are required: the device tree, Linux file system, Linux built image, and Linux boot loader.

4.4.2.1 Linux device tree

Firstly, modifications were performed to the Linux device tree (DTS), which maps every device used by Linux and allows associating the devices with the respective device drivers and device interrupts. The device tree should be kept

minimal, therefore modifications consisted on only adding the mandatory devices for Linux, such as the CPU, memory, interrupt controller, UART, global timer, SLCR, SD card, and user implemented devices. For instance, in order for the implemented timer device interrupts to be detected and handled by Linux, an entry containing device information, physical address, associated device driver and interrupts must be added, as demonstrated in Listing 4.10.

Listing 4.10: Private Timer entry on Linux device tree.

```
amba_pl: amba_pl {
#address-cells = <1>;
#size-cells = <1>;
compatible = "simple-bus";
ranges ;
SelfSecured_PTIMER_approach1_0: SelfSecured_PTIMER_approach1@43c00000 {
        compatible = "mycompany,ptimer_driver";
        interrupt-names = "interrupt";
        interrupt-parent = <&intc>;
        interrupts = <0 29 4>;
        reg = <0x43c00000 0x10000>;
        xlnx,s00-axi-addr-width = <0x4>;
        xlnx,s00-axi-data-width = <0x20>;
}...};
```

Other required modifications are necessary in the device tree boot arguments, enabling Linux single core execution and defining the memory location of the initial Linux file system specified in the linker script.

4.4.2.2 Linux file system

Linux file system has the initial composing files of Linux system and can be built with two different formats: *ramdisk* and *initramfs*. To modify the default file system (e.g. adding custom precompiled user space applications) the following operations must performed in the respective order: (i) extract the initial ramdisk image from the zipped archive; (ii) mount the extracted disk image; (iii) access the mounted file system and perform the pretended modifications, (iv) unmount the extracted disk image and recompress the image which will be used in the boot loader phase that will be further addressed. A relevant example of a precompiled user application added to the Linux file system to access the implemented device driver is illustrated in Listing 4.11.

Listing 4.11: Example of an user application to access a device driver.

```
1 int main(){...
2    //Open device with read/write access...
3    fd = open("/dev/UART_driver", O_RDWR);
4    // Send the string to the device driver
5    ret = write(fd, stringToSend, strlen(stringToSend));
6    // Read the response from the device deriver
7    ret = read(fd, receive, rcv_length);
8 ...}
```

4.4.2.3 Linux modifications and build

We have selected Xilinx Linux version 2015.4, available at Xilinx *Git* repository. Vivado toolchain already provides a set of default kernel configurations (kconfig) for their different SoCs, including for the Zynq-7000 SoC, which facilitates the kernel custom configuration. Therefore, before performing any modification, the Zynq default configurations should be applied beforehand. The kernel configuration can be further modified through the kconfig menu interface to add necessary features, device drivers, and other custom options.

Due to LTZVisor's interrupt model, the FIQ stack initialization had to be removed, since FIQs are exclusively routed to the secure world, and IRQs routed to the non-secure world. Furthermore, devices configured as secured are occasionally required by the non-secure world, imposing accessibility problems. Hence, to access some required secure devices the non-secure world needs to perform this accesses mediated by the hypervisor. To perform this accesses, as well as access some secure CP15 and SLCR registers, three SMC instruction were added (i.e. *secure_read*, *secure_write*, *secure_cp15_write*), allowing the non-secure guest OS to perform this operations, under the LTZVisor's supervision.

4.4.2.4 Second stage bootloader

The employed Linux bootloader *zcomposite* is a minimal boot loader which assembles the four required components into a single binary file. Apart from the Linux components, the boot loader requires three other files: the linker script, a makefile, and an assembly file. In the Linux linker script Zynq file the entry point for the *zcomposite* image, device tree offset, file system offset, and Linux image offset addressed are all specified. The *cleareg* assembly file, added to the first section of the linker script, provides a simple register setup code for stand-alone Linux booting setting parameters, such as Xilinx machine number, Linux starting point address, and device tree blob address. The makefile generates the binary

file image, which must be included in the secure world memory, from where the hypervisor will copy the file to the non-secure world and jump to its address. This binary file is generated in a specified address entry point, based on the agglomeration of the previous cross-compiled Linux image, compressed file system, and device tree blob.

In the LTZVisor non-secure guest configuration file, the binary load address should be changed to the same address specified in the Linux boot loader linker script. Also, the Linux binary file should be included by adding the binary path to the include binary parameter in the LTZVisor non-secure guest file.

4.4.2.5 Device Drivers

Device drivers must be integrated into Linux OS kernel, providing access to the hardware device's functionalities and perform associated non-secure interrupts (IRQs) handling. The device driver should be added before the Linux image compilation in the device drivers directory, where the makefile and kernel configurations file should be appropriately updated with the added device drivers. Inside the driver's directory, a makefile and kernel configuration file should also be created for calling upon the driver's files when the driver is enabled. Lastly, in order to add the created kernel configuration file and install the respective driver, the architecture specific configuration (Xilinx Zynq default config.) should also be updated.

Occasionally in some devices, such as the implemented hardware devices, the kernel is unable to retrieve some of its information, such as its associated interrupt lines, even if mapped in the kernel device tree. Platform drivers [Cor] are able to bound these undiscoverable devices and respective information with their drivers by matching names. By registering the platform driver with resources information specified on its structure, the kernel is able to get device's information, as the associated device tree entry, IRQ number, memory locations, and so on. The platform driver should include at least the *probe* and *remove* functions. The driver's *init* function calls the device registration function, providing the kernel with a list of the devices able to service, along with a pointer to the structure which contains device's information (e.g. device name), *probe* and *remove* functions. Then, the kernel is responsible for calling the *probe* function of each device where hardware is initialized, the device's resources allocated, and the device registered within the kernel. Inside this function, the device's physical address space is mapped into the virtual address that will be used to access the device from the kernel space, and the interrupt line read from the DTS entry and assigned to the

respective IRQ handler. An example of a platform driver is shown in Listing , 4.13.

Listing 4.12: Platform driver code extract.

```
1 static irqreturn_t mydriver_interrupt(int irq, void * dev_id)
2 {/* service iterrupt here */ return IRQ_HANDLED;}
3 static int mydriver_of_probe(struct platform_device *ofdev)
4 {...
5 //Map Physical address to Virtual address
6    dev_virtaddr = ioremap(PTIMER_BASEADDR, PTIMER_HIGHADDR-PTIMER_BASEADDR+1);
7    res = platform_get_resource(ofdev, IORESOURCE_IRQ, 0); ...
8    // save the returned IRQ
9    dm.irq = res->start;
10   printk(KERN_INFO "IRQ read form DTS entry as %d\n", dm.irq);
11   rval = request_irq(dm.irq, mydriver_interrupt, 0 , P_TIMER, &dm);...
12   //Device initialization actions...
13 }
14 static int mydriver_of_remove(struct platform_device *of_dev)
15 {       free_irq(dm.irq, &dm);return 1; iounmap(dev_virtaddr);}
16 static const struct of_device_id mydriver_of_match[] = {
17 { .compatible = "mycompany, ptimer_driver", },
18 };
19 static struct platform_driver mydrive_of_driver = {
20         .probe      = mydriver_of_probe,
21         .remove     = mydriver_of_remove,
22         .driver = {     .name = P_TIMER},
23 }; .....
24 module_init(mydrive_of_driver_init);
25 module_exit(mydrive_of_driver_cleanup);
```

4.4.2.6 Para-TrustZone modifications

The state-of-the-art Para-TrustZone method was implemented on the LTZVisor in order to perform comparative evaluations with other methods. Some slight modifications were performed to enable the GPOS driver to send requests for the secure device. These requests are performed through TrustZone privileged instruction SMC (*Secure Monitor Call*), which enables the non-secure world entering monitor mode.

Within Linux device drivers, instead of the device's operations being carried out through the usual accesses to the physical address of the device, which in this scenario is configured as secure and would trigger an external abort, the device's operations are now replaced with calls to the following shown assembler

functions. These calls will execute a SMC with the required kernel privilege, for the hypervisor to carry out the pretended device operation in the secure side.

Listing 4.13: Example of Para-TrustZone driver assembly functions.

```
#include <asm/assembler.h>
 ...
.global set_counter
.global set_timer
.global get_counter
        set_counter:
                mov     r1, r0
                ldr     r0, = PTIMER_SETCOUNTER
                smc     #0
                bx      lr
        set_timer:
                mov     r1, r0
                ldr     r0, = PTIMER_SET
                smc     #0
                bx      lr
        get_counter:
                ldr     r0, = PTIMEMR_GETCOUNTER
                smc     #0
                bx      lr
....
```

The LTZVisor must handle the requests from the GPOS driver, thus the pretended supported device operations are added to the the system call handling funcion (*board_handler*) and carried out depending on the SMC passed arguments upon the driver call.

Listing 4.14: Board handler function in *board.c*.

```
uint32_t board_handler(uint32_t arg0, uint32_t arg1, uint32_t arg2){
switch(arg0) {
        case (PTIMER_SETCOUNTER):
                write32( (void *)(0x43C00004) , (uint32_t)(arg1));
                break;
        case (PTIMER_SET):
                backup = read32((volatile void*)0x43C00000 + (uint32_t)(8));
                if (arg1) backup = backup | 0x00000001; else backup = backup &
                     ~(0x00000001);
                write32( (void *)(0x43C00008) , (uint32_t)(backup));
                break;
        case (PTIMER_GETCOUNTER):
```

```
12                  arg0 = read32((volatile void*)0x43C00000 + (uint32_t)(4));
13                  break;
14 ...       default: break;
15 }
16 return arg0;
17 }
```

5. Evaluation

This section evaluates the developed self-secured devices, along with the most relevant existing shared device methods. The evaluation was conducted on a Zybo Board running at 50 MHz. More details regarding the hardware platform are available in Section 3.3. The system was configured to run FreeRTOS (version 7.0.2) and Linux (2015.4 Xilinx version) as secure and non-secure VMs, respectively, running on top of LTZVisor with a single-core configuration. Hereby are described the performed experiments in order to test the developed work. Then, the obtained results are discussed, displayed, and compared quantitatively among most important state-of-the-art solutions in order to obtain tangible results in terms of: (i) security; (ii) engineering effort; (iii) memory footprint; (iv) hardware costs; and (v) performance.

5.1 Engineering effort

In order to assess the engineering effort associated with the implementation of the self-secured devices in a concrete system, we used the Understand software tool to measure the number of lines-of-code (LoC) of both the HDL and software files.

5.1.1 Hardware Modifications

Figure 5.1 shows the LoC of the Verilog files implemented for each self-secured device. In the timer device, the additional effort to implement both self-secured approaches is less than 20% relative to the native solution, while device duplication would have duplicated the number of required LoC. In the self-secured timer default approach, the few additional LoC are related to the provided support for the secure world to access the non-secure interface. While on the minimal approach, the self-secured timer is only able to access the non-secure interface exclusively from the non-secure world, hence support is not required for the additional AXI interfaces.

In the UART device, the required modifications are reduced to approximately 15%, due to the higher complexity of the device hardware logic, which enables for the approach application efforts to be dispersed among the overall logic.

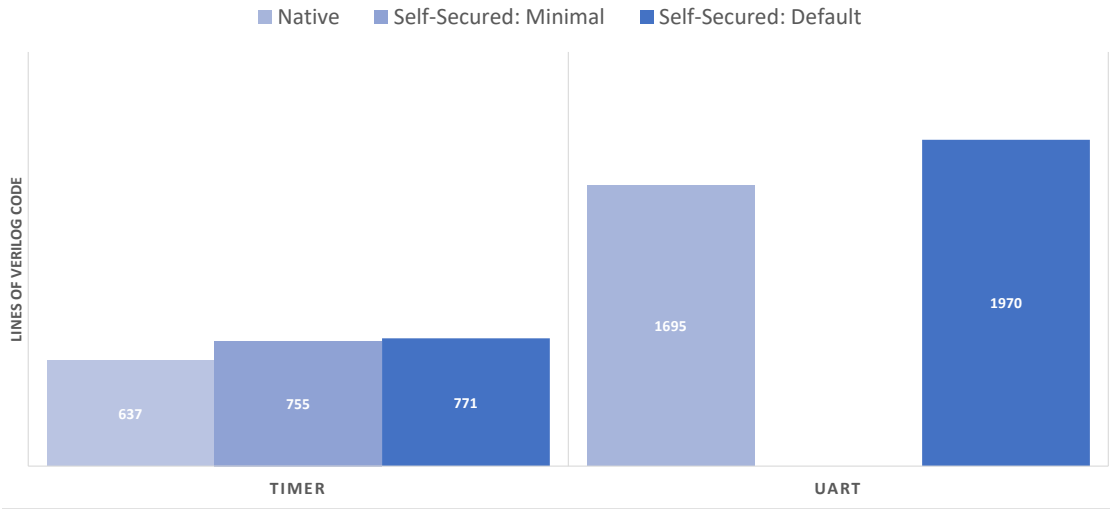


Figure 5.1: LoC of the Verilog files, with and without the self-secured implementation.

5.1.2 LTZVisor Modifications

Figure 5.2 illustrates the LTZVisor LoC for each method. For the self-secured approach, in both devices, the few additional lines are related to the security configuration and routing of the differentiated interrupts upcoming from the secure and non-secure device interfaces. The re-partitioning approach does not require any changes to the hypervisor, since the pure method mechanisms are entirely implemented in both OSes.

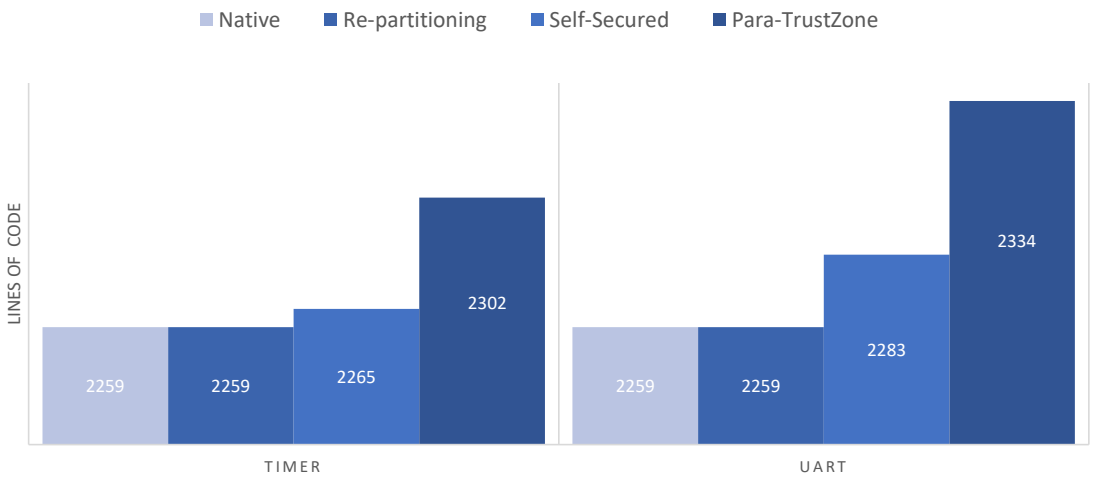


Figure 5.2: Number of LTZVisor lines of source code for each approach.

The Para-TrustZone approach has a significantly associated engineering effort when compared to other methods. As a consequence of the hypervisor itself having to handle every SMC upcoming from the GPOS driver and carrying the respective device operation, based on the passed arguments of every request. However, if only minimal support to the device's operations is provided to the GPOS, this cost could be further reduced, a trade-off between the number of supported functionalities and the necessary engineering effort.

5.1.3 FreeRTOS Modifications

As depicted in Figure 5.3, the Para-TrustZone method has no associated changes with the RTOS, since it only depends on the hypervisor to perform the non-secure requests. Besides, the Self-Secure Private Timer minimal approach also does not impose changes to the RTOS device drivers, due to only supporting accesses to the secure interface. However, the default Self-Secure approach introduces some minor modifications, that merely consist in providing support for the secure world to perform accesses to both the secure and non-secure device interfaces.

In contrast, on the Repartitioning approach, the added effort is related to the required repartitioning mechanisms, which are responsible for sending the "UNPLUG" event whenever the RTOS needs the device, and the "PLUG" event when it is no longer needed. Furthermore, upon these events the RTOS should save and restore every device register at each device repartition, as well as reconfigure the device security according to the world context switch event.

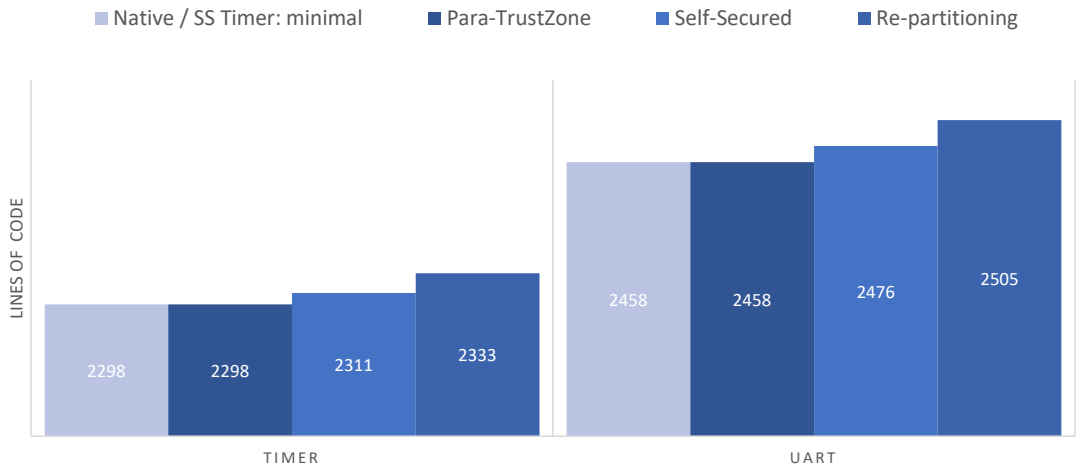


Figure 5.3: Number of FreeRTOS and SW device driver LoC for each approach.

5.1.4 GPOS Modifications

Figure 5.4 shows the LoC of the GPOS device driver source code implemented for each approach. On the self-secured approach, modifications only consisted of re-mapping the registers accesses to the non-secure interface registers and providing support for the non-secure interface interrupt handling.

Diversely, in the Re-Partitioning method, considerable modifications to the GPOS are necessary, but at the user-level instead. Although, for a fair comparison, these modifications are also included in this evaluation. These modifications are related to the task that runs continuously, checking for upcoming *UNPLUG* and *PLUG* events, which are responsible for unloading and re-loading the device driver, respectively.

The Para-TrustZone requires extensive modification to the GPOS driver, as a consequence of being mandatory replacing the devices operations for SMCs with the respective arguments. This SMCs enable the GPOS to access the secure device through operation requests, sent out directly to the hypervisor.

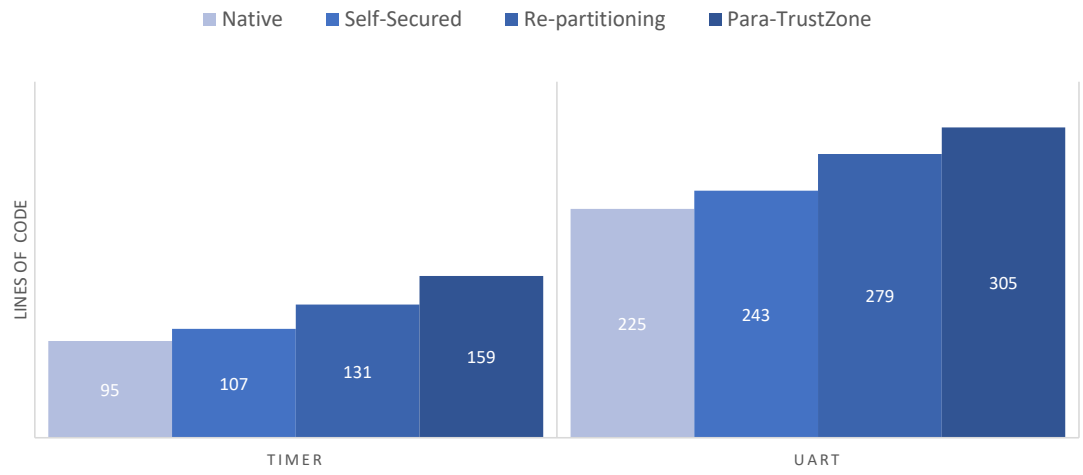


Figure 5.4: GPOS Device drivers number of lines of code on each approach.

5.2 Memory Footprint

The memory footprint of each implemented approach was measured using the size tool of the ARM GNU toolchain. This tool is able to calculate the memory footprint of an output or image file, and report it into three different categories: (i) *.text*, contains executable code, constant variables, and vector tables; (ii) *.data*, contains initialized system variables; (iii) *.bss*, contains non-initialized system variables, stack and heap dynamic variables.

Table 5.1 depicts the measured overhead of the entire LTZVisor image for the different approaches. The presented values include the hypervisor code itself, boot code, libraries, FreeRTOS code, and its device drivers. As a consequence of the previously shown incurred modifications to the LTZVisor, FreeRTOS, and device drivers, these modifications are also reflected in terms of memory overhead. Meaning, the Para-TrustZone, and Repartitioning approach, respectively introduce the highest overall.

Table 5.1: LTZVisor and FreeRTOS memory footprint (bytes).

	Memory Footprint in bytes			
LTZVisor image	.Text	.data	.bss	**Total**
Self-Secured Timer: Minimal	51530	468	460432	512430
Self-Secured Timer: Default	51828	468	460440	512736
Timer Repartitioning	52375	476	460448	513299
Timer Para-TrustZone	52898	468	460456	513822
Self-Secured UART	52341	500	460472	513313
UART Repartitioning	52818	508	460488	513814
UART Para-TrustZone	53582	500	460504	514586

Table 5.2 the size (bytes) of the GPOS device driver for the different implemented approaches. The observed differences among the methods are related to the previously mentioned modifications, required by each approach.

Table 5.2: Device drivers memory footprint (bytes).

	Memory Footprint in bytes			
Device drivers	.Text	.data	.bss	**Total**
Native Timer	1764	164	292	2220
Timer Repartitioning	1764	164	292	2220
Self-Secured Timer: Mininal	1764	164	292	2220
Self-Secured Timer: Default	1780	164	292	2236
Timer Para-TrustZone	1856	164	300	2324
Native UART	1884	164	548	2496
UART Repartitioning	1884	164	548	2496
Self-Secured UART	1904	164	548	2616
UART Para-TrustZone	1986	164	564	2714

5.3 Performance

Performance was evaluated using the PMU component to accurately determine the number of clock cycles consumed by read/write operations, and device re-partitioning associated latency of each re-partitioning event. Results shown in Figure 5.5 represents the average of one hundred collected samples and demonstrate a considerable performance overhead introduced by the para-TrustZone method upon read/write accesses to the device. In contrast, both the self-secure and re-partitioning approaches has a performance similar to the native execution, due to device accesses being performed directly to the hardware. Even though the re-partitioning method does not incur any overhead on read/write accesses, this method introduces considerable device latency (7055 clock cycles) on every device repartitioning, which is the amount of time that the FreeOS has to wait until the shared device can be again used reliably.

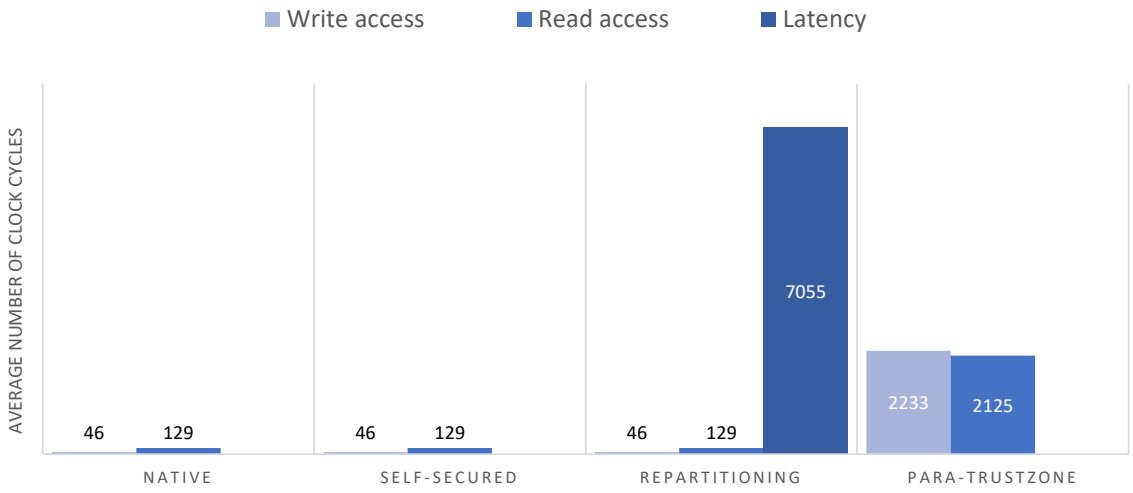


Figure 5.5: Number of clock cycles for write/read device operations and incurred device latency.

5.4 Hardware Costs

In order to measure the impact at the hardware level, the post-implementation hardware results of the Private Timer and UART, with and without the self-secured extension, and upon device duplication were assessed and compared.

FPGAs are programmable semiconductor devices, based around a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects [BGM11]. Therefore, a CLB can be considered the basic logic unit of an FPGA, which consists of a configurable Look-Up Table (LUT), Flip-Flops (FF), and some selection circuitry (e.g. multiplexer). LUTs are highly flexible and can be configured to handle combinatorial logic, shift registers or RAM. LUTRAMs are faster, used for smaller memory needs, the read is asynchronous, and they place less burden on the place and router. However, when a large amount of memory is required BRAM should be used instead if this memory does not need to be accessed during the same cycle in which the address is provided. Global buffer (BUFG) resource is one of the most expensive resources in FPGA, used for distribution of internal clocks throughout the FPGA. Furthermore, whenever logic operations require a large number of LUTs, consequently require a large area of the FPGA which makes these operations slower. However, the FPGA contains dedicated DSP blocks that can be used instead, to perform these operations faster, use less power and within a smaller FPGA area.

Vivado utilization report parameters indicates the number of registers, LUTs,

I/Os, BUFGs, DSPs and FFs of the current design required for the implementation. The more logic is added to the current design, consequently increases the utilization rate of these parameters.

5.4.1 Self-Secured: Private Timer

The assessed post-implementation hardware results of the Private Timer are depicted in Figure 5.6. In both self-secured devices the utilization rate of global buffers (BUFG) and (LUTRAM) remained the same, while the Look-Up table (LUT) utilization increased from 3,5% to 4,3% in the minimal approach, or to 4,5% in the default approach. Also, Flip-flop (FF) utilization had just a slight increase from 2% to 2,16% or 2,6% for the minimal and default. However, in case of device duplication in the same scenario, the following hardware costs are increased: Flip-flop (FF) utilization from 2% to 3.2%; Look-Up table (LUT) utilization from 3,5% to 5,7%; and (LUTRAM) utilization has a slight increase of 0,05%.

Therefore, on the minimal self-secured approach the Look-Up table (LUT) and Flip-flop (FF) utilization had a relative change of 23% and 8%, respectively. While on the default approach the relative change was of 28% and 30%, respectively. In contrast, with device duplication, the Look-Up table (LUT) and Flip-flop (FF) utilization had a considerable relative change of 60% and 63%, and also a 5% increase of LUTRAM utilization.

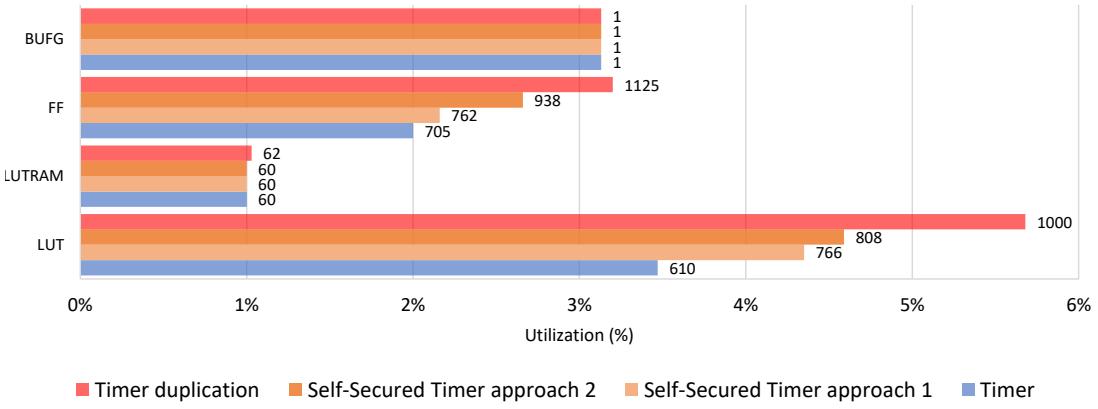


Figure 5.6: Self-Secured private timer post-implementation hardware costs.

5.4.2 Self-Secured: UART

Figure 5.7 shows the assessed post-implementation hardware results of the UART device. With the application of the Self-Secured approach to the device, the utilization rate of global buffers (BUFG), digital signal processing (DSP) blocks

and distributed RAM (LUTRAM) remained the same, while the I/O utilization increased from 8% to 10%, block RAM (BRAM) from 1,7% to 3,33%, Flip-flop (FF) utilization raised from 3,9 % to 4,7%, and Look-Up table increased from 39% to 40,5%. However, if the device is completely replicated the hardware costs are the following: I/O utilization increased from 8% to 16%; DSP blocks utilization raised from 1,25% to 2,5%; Block RAM (BRAM) from 1,7% to 3,33%; Flip-flop(FF) utilization increased from 3,9 % to 6,9%; LUTRAM had a slight increase from 1% to 1,03%; and Look-Up table had the highest increase, from 39% to 74%.

Therefore, with the Self-Secured method I/O, BRAM, FF and LUT utilization had a relative change of 25%, 95,8%, 20,5% and 3,8%, respectively. However, when duplicating the entire device I/O, DSP blocks, BRAM, FF, LUTRAM, LUT and LUTRAM utilization rate has a considerable relative increase of 100%, 100%, 95,8%, 77%, 89% and 3%, respectively.

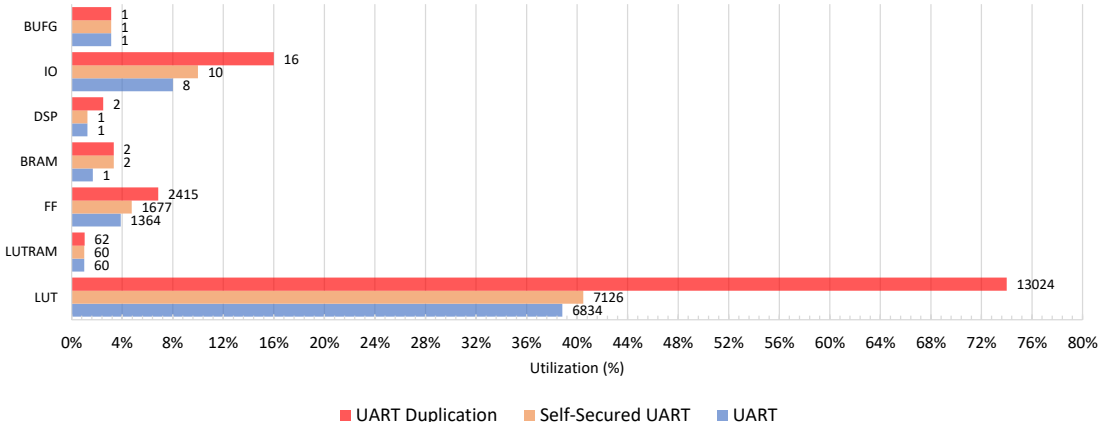


Figure 5.7: Self-Secured UART post implementation hardware costs.

5.5 Security

In this section, the security properties of the developed self-secured devices are analyzed, outlining the security guarantees provided by the implemented solution regarding the four fundamental elements of CIA (control, integrity, and availability, and confidentially). Furthermore, to evaluate security on self-secured devices some performed experiments are demonstrated in order to test the implemented protection mechanisms against device misuse.

5.5.1 Security Guarantees

Self-Secured devices have fully or partially achieved the three fundamental elements of CIA:

- **Confidentiality** is the ability to restrict data to those authorized to access it. Self-Secured devices provide confidentiality by means of TrustZone' strong spatial isolation mechanisms. The GPOS cannot access any data allocated on the secure interface nor registers stored in the secure register bank, because the implemented protection mechanism denies any unauthorized access.

- **Integrity** enforces the consistency and trustworthiness of the device over its entire life cycle. Self-Secured devices ensure the successful completion of its secure operations. Meaning, that an operation performed by the secure world must be completed, regardless of operation requests coming from the non-secure side. Device's integrity is also ensured by denying the non-secure access to configuration registers that may tamper with the device's normal flow;

- **Availability** refers to the ability that authorized parties to have access to the device whenever needed. Self-Secured devices ensure that the secure world has full access over the secure interface of the device at any time. Preventing any action from the non-secure world that may inhibit the secure-world usage of the shared device for an unbounded amount of time. Additionally, due to the co-existence of privileged (FIQs) and unprivileged (IRQs) interrupt sources, FIQs belonging to the secure interface of the device are able to preempt the execution of the GPOS, even when executing an IRQ request;

5.5.2 Security Experiments

The first experiment was performed with the self-secured private timer and is illustrated in Figure 5.8. This experiment consisted on setting both self-secured timer interfaces counter values, from a FreeRTOS secure task. Both secure and non-secure interrupts were enabled, as well as the timer itself. Then, the respective counters from each interface were decremented until reaching zero and triggering an overflow. The non-secure counter, was configured with a lower value, thus reached the overflow first and triggered a non-secure interrupt (IRQ). Soon after, the secure counter , with the highest configured value, reached the secure overflow and triggered a FIQ instead, proving that both interfaces can be accessed through the secure world and its respective interrupts are routed according to their security.

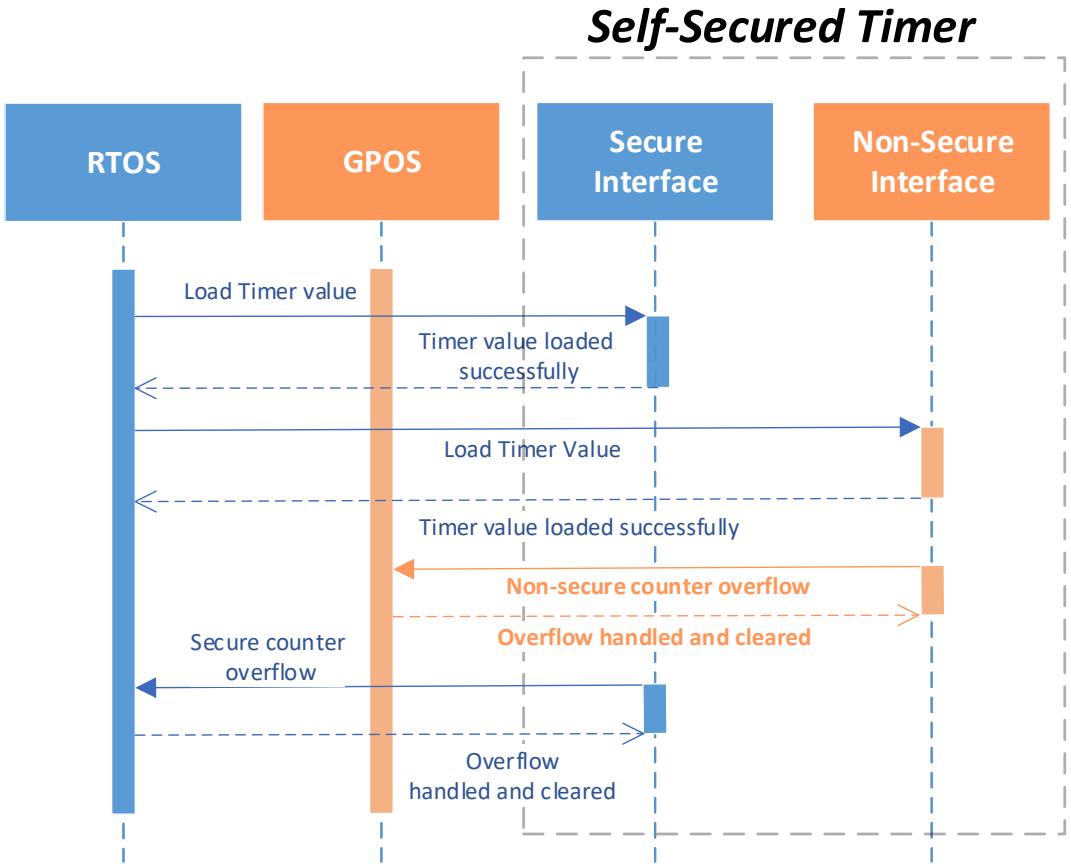


Figure 5.8: Protection mechanisms of the Self-Secured Timer upon FreeRTOS accesses.

On a second experiment (Figure 5.9) the same exact accesses were performed, this time from the GPOS side instead. As expected, the secure timer interface remained unaltered, while the non-secure interface is successfully accessed, and its respective interrupt triggered whenever the non-secure counter overflows. On both experiments, the values of both counter interfaces were printed continuously, and TrustZone signals debugged through the Integrated Logic Analyzer to confirm the device hardware logic protection mechanisms against the illegal accesses.

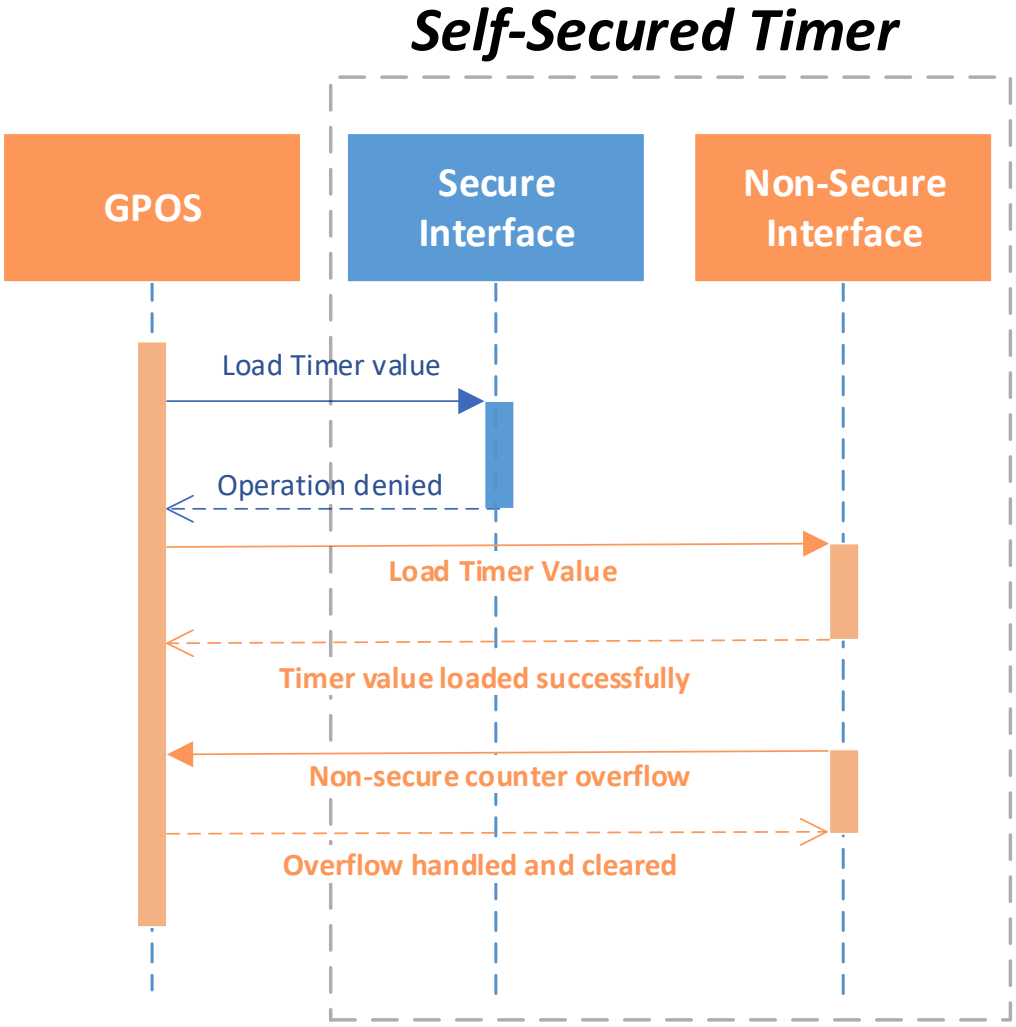


Figure 5.9: Protection mechanisms of the Self-Secured Timer upon GPOS accesses.

Another relevant experiment was performed with the Self-Secure UART and is illustrated in Figure 5.10. Following the same principles as previously performed tests on the Self-Secured Timer, a specific device operation is performed from both worlds. In this illustrated case, the chosen UART operation is a string transmission. The FreeRTOS was able to send the string to both separate terminals, while the GPOS is only capable of transmitting to the non-secure terminal. The same experiment was also performed with wrong data formats in order to trigger the respective interrupts and verify that the triggered interrupts security matches the data security type. All experiments were performed with the support of two separate auxiliary terminals connected to the corresponding ports, capable of sending data and displaying the received data.

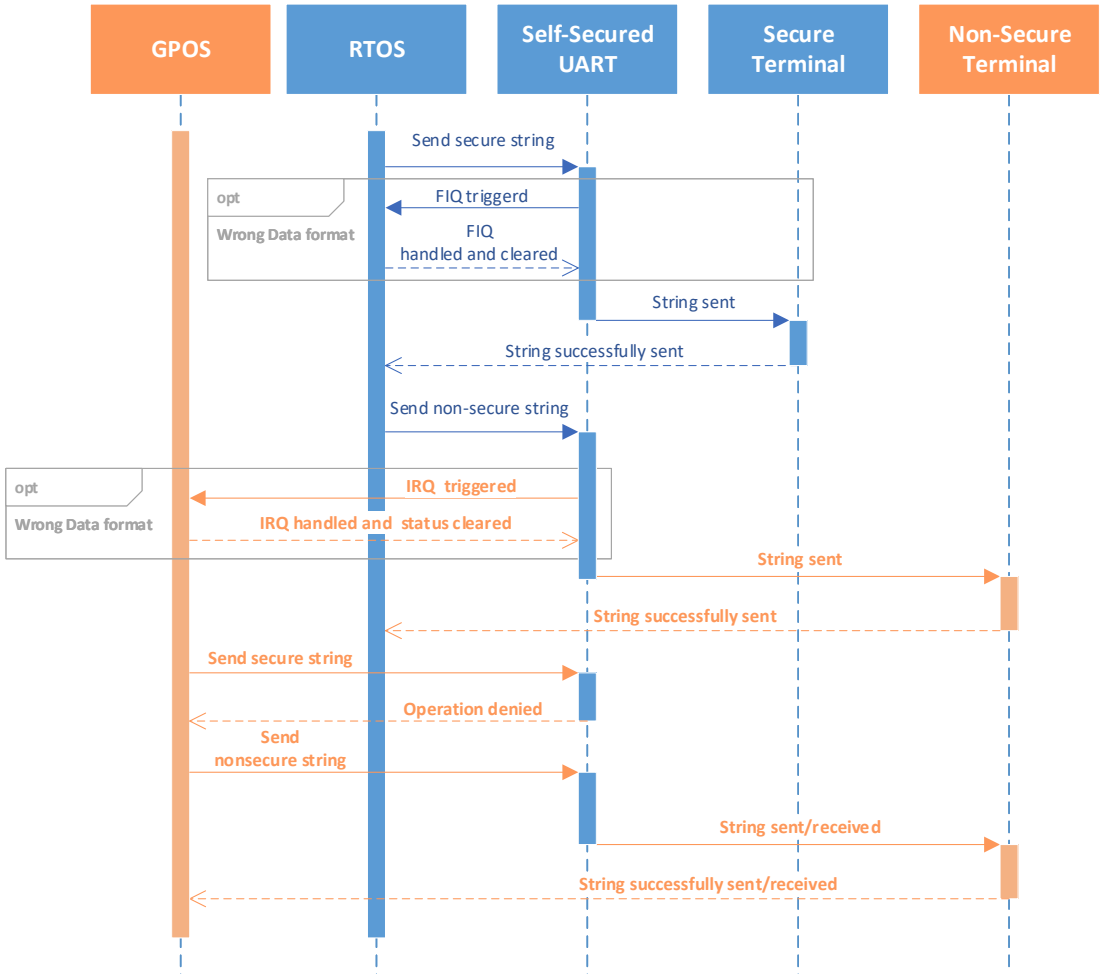


Figure 5.10: Protection mechanisms of the Self-Secured UART.

5.6 Discussion

Given all the previously evaluated metrics of the new self-secured approach along with other implemented existing shared device access methods, Table 5.3 shows a side by side comparison of all the collected evaluations.

Starting the table analysis from the top, with device duplication each guest OS owns a dedicated copy of the device by simply duplicating the entire hardware logic, without any required modifications or additional engineering effort, and leaving no margin for security issues nor performance degradation. Although, it is not a reasonable solutions since it implicates huge hardware costs, unsuitable for embedded system solutions with a small form factor.

With the novel self-secured approach at the acceptable costs of additional hardware protection mechanisms and minimal device driver modifications, native performance is maintained and security utterly safeguarded by TrustZone extensions.

Besides, is clearly noticeable that when the concept is applied to higher complexity level devices, the incurred hardware costs are dispersed by the overall logic, causing additional costs to become acceptable under the performance-security-hardware spectrum.

Additionally, two software state-of-art methods were also evaluated. Device Para-TrustZone grants the GPOS access to the secure device through SMC, consequently introducing overhead to read/write device operation, beyond the required engineering effort to handle and carry out each SMC. Device Re-partitioning dynamically reassigns the device security on run-time, and once the device is assigned it allows both OSs to perform read/write accesses without any overhead. However, the reassignment mechanism implementation requires considerable engineering efforts and introduces performance overhead. Furthermore, during the reassignment process period, the device is unusable, incurring a considerable device latency that may break time-critical needs.

Most importantly, in both methods, the frequency of GPOS accesses is not limited, which in case of GPOS misbehavior massive request may be performed, potentially causing the secure device failure. From a general point of view, both existing methods lack in terms of security and performance, especially when compared with the new approach proposed in this work.

Table 5.3: Evaluation results comparison.

Device access method	*Hardware Costs*	*Engineering effort*	*Memory Footprint*	*Performance*	*Security*
Device duplication	○	●	●	●	●
Self-Secured on low complexity devices	◑	◕	●	●	●
Self-Secured on higher complexity devices	◕	◕	●	●	●
Device Re-partition	●	◑	◑	◑	◑
Device Para-TrustZone	●	◔	◔	◔	◔

● Excellent ◕ Good ◑ Satisfactory ◔ Poor ○ Unsuitable

6. Conclusion

With the advent of the IoT, security concerns have been escalating exponentially. In an endeavor to enhance security, embedded systems development has been unsuccessfully focusing on providing additional security features to the system in a later stage. Instead, security must start to be guaranteed from the outset.

ARM TrustZone is an example of a security technology which promotes hardware as the initial root of trust and has been proven that it can be efficiently exploited as a secure virtualization solution. As in any virtualized system, hardware resources need to be shared between multiple virtual environments in the same platform. However, in TrustZone-enabled SoCs, devices can only be configured as secure or non-secure. Not being able to share devices between virtual environments is an enormous bottleneck to the characteristic scalability of virtualization.

This thesis presented a novel approach for shared device access in TrustZone-based architectures, extending the dual-world concept of TrustZone to the inner logic of the device by splitting the device's logic into a secure and non-secure interface. To accomplish this, it was imperative to identify the vulnerable part of the device's logic, that can potentially be exploited, and restrain accesses through the TrustZone extended protection signals, present in the main system bus.

This concept was experimented through the implementation of low and medium complexity devices, in order to assess the hardware costs behind such implementations and link them to their complexity level. The obtained results are encouraging, managing to keep the additional hardware costs acceptable for the achieved security enhancements. As demonstrated, the hardware costs diminish with higher complexity level, as the implementation costs are dispersed throughout the overall logic.

Additionally, a comparative study between current existing shared devices access approaches and the new self-secured approach was performed, assessing results in terms of security, engineering effort, performance and memory footprint. The

evaluation demonstrated that the self-secured approach is able to achieve a noticeable speedup, without the considerable TCB expense and memory footprint increase when compared to existing state-of-art methods, while safeguarding the device through TrustZone extended control signals.

6.1 Future Work

The concept was implemented and can be used on reconfigurable platforms, however hardware costs could be estimated for application-specific integrated circuit (ASIC) deployments. Although the obtained results clearly demonstrate considerable enhancements, this methods' application is use-case dependent, i.e. to apply this approach to a device it requires an in-depth analysis of its internal logic.

The application of the self-secured approach to a device with an even higher complexity would enable establishing a more accurate relation of the hardware cost to the complexity level. Also, it might aid to comprehend from which device complexity level is worth applying the concept and how to reduce the engineering effort even further. After linking each device results and hardware costs with their respective complexity level, further research should focus on achieving a generic methodology of the concept to any device, regardless of their complexity level.

Furthermore, future research could focus on developing an automated design process that interprets the device hardware logic and generates the hardware device with the self-secured extension embedded in it.

Lastly, microcontrollers have been evolving towards recent demands and are now capable of consolidating multiple OSes in the same platform. Moreover, some modern microcontollers already feature TrustZone technology. Therefore, the self-secured concept might be hereafter exploited in low-end heterogeneous architectures.

References

[ARM09] ARM. ARM Security Technology. Building a Secure System using TrustZone Technology ARM. Technical report, 2009.

[ARM12] ARM. Cortex -A9 MPCore: Technical Reference Manual. 2012.

[ARM15] ARM. New amba specification extends security to embedded design: Amba 5 ahb5, 2015.

[ARM16] ARM. ARM ® Cortex ® -A Portfolio ARMv8-A. Technical report, 2016.

[BGM11] Alexander Biedermann and H Gregor Molter. *Design Methodologies for Secure Embedded Systems*, volume 78. January 2011.

[Chi07] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. First edition, 2007.

[Cor] Jonathan Corbet. Platform devices and drivers. `https://lwn.net/Articles/448499/`.

[DD03] James P Davis and James P Davis. Universal Asynchronous Receiver Transmitter (UART - 8251). In *Spring*, page 12, March 2003.

[FLWH10] Torsten Frenzel, Adam Lackorzynski, Alexander Warg, and Hermann Härtig. ARM trustzone as a virtualization technique in embedded systems. In *Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya*, October 2010.

[Fre] FreeRTOS website. `https://www.freertos.org/`.

[Hei08] Gernot Heiser. The Role of Virtualization in Embedded Systems. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 11–16, April 2008.

[HGX+17] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM Trustzone. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 541–556, August 2017.

[Kai09] Robert Kaiser. Complex embedded systems - A case for virtualization. In *2009 Seventh Workshop on Intelligent solutions in Embedded Systems*, pages 135–140, June 2009.

[Kal14] Stefan Kalkowski. Virtualization Dungeon on ARM - Hands on experience talk about virtualization experiments. February 2014.

[KLJ+13] Se Won Kim, Chiyoung Lee, Moowoong Jeon, Hae Young Kwon, Hyun Woo Lee, and Chuck Yoo. Secure device access for automotive software. In *2013 International Conference on Connected Vehicles and Expo, ICCVE 2013 - Proceedings*, pages 177–181, January 2013.

[Lan11] Ralph Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. In *IEEE Security Privacy*, volume 9, pages 49–51, May 2011.

[LI04] ARM Limited and ARM Ihi. AMBA AXI Protocol. Technical report, 2004.

[LMH+14] Wenhao Li, Mingyang Ma, Jinchen Han, Yubin Xia, Binyu Zang, Cheng-Kang Chu, and Tieyan Li. Building Trusted Path on Untrusted Device Drivers for Mobile Devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, APSys '14, pages 8:1–8:7, June 2014.

[MAC+17] José Martins, João Alves, Jorge Cabral, Adriano Tavares, and Sandro Pinto. μRTZVisor: A Secure and Safe Real-Time Hypervisor. In *Electronics*, volume 6, page 93, October 2017.

[MJNH16] Carlos Moratelli, Sergio Johann, Marcelos Neves, and Fabiano Hessel. Embedded virtualization for the design of secure IoT applications. In *2016 International Symposium on Rapid System Prototyping (RSP)*, pages 1–5, October 2016.

[OGP18] Daniel Oliveira, Tiago Gomes, and Sandro Pinto. Towards a Green and Secure Architecture for Reconfigurable IoT End-devices. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, August 2018.

[OMC+18] André Oliveira, José Martins, Jorge Cabral, Adriano Tavares, and Sandro Pinto. TZ- VirtIO: Enabling Standardized Inter-Partition Communication in a Trustzone-Assisted Hypervisor. In *2018 IEEE 27th International Symposium on Industrial Electronics (ISIE)*, pages 708–713, June 2018.

[Pal14] Prushothaman Palanichamy. TrustZone Technology Support in Zynq-7000 All Programmable SoCs. Technical report, May 2014.

[PBB13] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. In *ACM Comput. Surv.*, volume 45, pages 40:1–40:34, June 2013.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. In *Commun. ACM*, volume 17, pages 412–421, July 1974.

[PGP+17] Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. IIoTEED: An Enhanced, Trusted Execution Environment for Industrial IoT Edge Devices. In *IEEE Internet Computing*, volume 21, pages 40–47, January 2017.

[Pin17] Sandro Pinto. *Thesis: Safe Virtualization-based Framework for Embedded Systems Development*. PhD thesis, Universidade do Minho, 2017.

[PMB15] Dorottya Papp, Zhendong Ma, and Levente Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, volume 00, pages 145–152, July 2015.

[POP+15] Sandro Pinto, Daniel Oliveira, Jorge Pereira, Jorge Cabral, and Adriano Tavares. FreeTEE: When real-time and security meet. In *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–4, September 2015.

[POP+17] Sandro Pinto, André Oliveira, Jorge Pereira, Jorge Cabral, João Monteiro, and Adriano Tavares. Lightweight multicore virtualization architecture exploiting ARM TrustZone. In *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society*, pages 3562–3567, October 2017.

[PPG+17a] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. Towards a TrustZone-Assisted Hypervisor for Real-Time Embedded Systems. In *IEEE Computer Architecture Letters*, volume 16, pages 158–161, July 2017.

[PPG+17b] Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the Key. In *29th Euromicro*

Conference on Real-Time Systems (ECRTS 2017), volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 4:1–4:22, June 2017.

[PPO+14] Sandro Pinto, Jorge Pereira, Daniel Oliveira, F. S. Alves, Esam Qaralleh, Mongkol Ekpanyapong, Jorge Cabral, and Adriano Tavares. Porting SLOTH system to FreeRTOS running on ARM Cortex-M3. In *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pages 1888–1893, June 2014.

[PS18] Sandro Pinto and Nuno Santos. Demystifying Arm TrustZone: A Comprehensive Survey. In *ACM Computing Surveys*, volume preprint, 2018.

[PTM16] Sandro Pinto, Adriano Tavares, and Sergio Montenegro. Space and time partitioning with hardware support for space applications. In *Data Systems In Aerospace (DASIA), European Space Agency,(Special Publication) ESA SP*, August 2016.

[Pul16] Henley Court Pullman. ZYBO™ FPGA Board Reference Manual. Technical report, 2016.

[Rea16] Real Time Engineers Ltd. The Free RTOS™ Reference Manual. Technical report, 2016.

[RHFN+12] Fernando Rodríguez-Haro, F Freitag, Leandro Navarro, Efraín Hernánchez-sánchez, Nicandro Farías-Mendoza, Juan Guerrero-Ibañez, and Apolinar González. A summary of virtualization techniques. In *Procedia Technology*, volume 3, pages 267 – 272, December 2012.

[RS07] Himanshu Raj and Karsten Schwan. High Performance and Scalable I/O Virtualization via Self-virtualized Devices. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 179–188, June 2007.

[SGB+16] Junaid Shuja, Abdullah Gani, Kashif Bilal, Atta Ur Rehman Khan, Sajjad A. Madani, Samee U. Khan, and Albert Y. Zomaya. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. In *ACM Comput. Surv.*, volume 49, pages 1:1–1:36, April 2016.

[SHT12a] Daniel Sangorrín, Shinya Honda, and Hiroaki Takada. Reliable and efficient dual-OS communications for real-time embedded virtualization. volume 29, pages 182–198, November 2012.

[SHT12b] Daniel Sangorrín, Shinya Honda, and Hiroaki Takada. Reliable Device Sharing Mechanisms for Dual-OS Embedded Trusted Computing. volume 7344, pages 74–91, June 2012.

[SN05] James E. Smith and Ravi Nair. The Architecture of Virtual Machines. In *Computer*, volume 38, pages 32–38, May 2005.

[SRSW14] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Playstation. In *SIGARCH Comput. Archit. News*, volume 42, pages 67–80, February 2014.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, pages 1–14, June 2001.

[SYKS14] Takumi Shimada, Takeshi Yashiro, Noboru Koshizuka, and Ken Sakamura. A real-time hypervisor for embedded systems with hardware virtualization support. In *2015 TRON Symposium (TRONSHOW)*, pages 1–7, December 2014.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. In *Computer*, May 2005.

[USK11] Arijit Ukil, Jaydip Sen, and Sripad Koilakonda. Embedded security for Internet of Things. In *2011 2nd National Conference on Emerging Trends and Applications in Computer Science*, pages 1–6, March 2011.

[VH11] Prashant Varanasi and Gernot Heiser. Hardware-supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, July 2011.

[VMw06] VMware. Virtualization overview. Technical report, 2006.

[WSC+07] Paul Willmann, Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, and Willy Zwaenepoel. Concurrent Direct Network Access for Virtual Machine Monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 306–317, February 2007.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven Gribble. Denali:

Lightweight Virtual Machines for Distributed and Networked Applications. March 2002.

[WW09] Yi-Hsien Wang and I-Chen Wu. Achieving high and consistent rendering performance of Java AWT/Swing on multiple platforms. In *Softw., Pract. Exper.*, volume 39, pages 701–736, March 2009.

[Xil] Xilinx. Zynq linux support, Xilinx wiki. `http://www.wiki.xilinx.com/Zynq+Linux`.

[Xil11] Xilinx. Xilinx AXI Reference Guide. Technical report, 2011.

[Xil12] Xilinx. LogiCORE - AXI4-Lite IP Interface. Technical report, 2012.

[Xil16a] Xilinx. Bringing Ultra High Productivity to Mainstream Systems & Platform Designers Vivado Design Suite HLx Editions. Technical report, 2016.

[Xil16b] Xilinx. Xilinx Software Development Kit (SDK). Technical report, 2016.

[Xil18] Xilinx. Zynq-7000 SoC Manual. Technical report, 2018.

[YBW10] Weider Yu, Dipti Baheti, and Jeremy Wai. *Real-Time Operating System Security.* PhD thesis, Computer Engineering Department, San Jose State University, 2010.

[ZMH15] Samir Zampiva, Carlos Moratelli, and Fabiano Hessel. A hypervisor approach with real-time support to the MIPS M5150 processor. In *Sixteenth International Symposium on Quality Electronic Design*, March 2015.