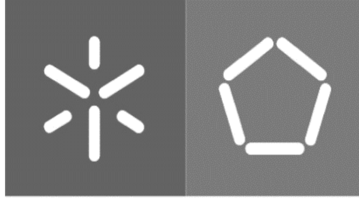
Universidade do Minho
Escola de Engenharia

Sérgio Augusto Gomes Pereira

A TrustZone-assisted Secure Silicon on a Co-Design Framework

Outubro de 2018



Universidade do Minho
Escola de Engenharia

Sérgio Augusto Gomes Pereira

A TrustZone-assisted Secure Silicon on a Co-Design Framework

Dissertação de Mestrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Doutor Sandro Pinto

Outubro de 2018

Acknowledgements

Firstly, I would like to thank my thesis advisor Dr. Sandro Pinto for the excellent guidance and all support giving during my studies, he consistently allowed this project to be of my own work, and steered me in the right the direction whenever he thought I needed it. I must pronounce my recognition to Dr. Adriano Tavares for all the advisement and transmitted knowledge during my master's degree.

I would also like to thank my fellow classmates (Ângelo Ribeiro, Hugo Araújo, Ailton Lopes, Pedro Machado, Nuno Silva, Pedro Ribeiro, José Silva, Francisco Petrucci and Ricardo Roriz) for the collaborative environment, the sleepless nights of working together before deadlines, and for all the fun we have had. To José Martins and all members of ESRG for the stimulating discussions, for all the help and support given to all of us. Without their passionate participation, the validation survey could not have been successfully conducted.

Some special words of gratitude to my friends who have always been a major source of support, cheered me on, and celebrated each accomplishment: "Duque", Manso, Joana, Pedro, Tiago, Alex, "BTT" and with my all sincere esteem José Silva a.k.a. "Astro". To my dear Carolina Clasen for listening to me and, at times, having to tolerate me over the past months.

Last but not the least, I would like to thank my family: my parents and to my two sisters, for supporting me spiritually throughout writing this thesis and my my life in general. Thank you for giving me this opportunity to follow my dreams and to never impose restrictions when it comes about studies.

Abstract

Embedded systems were for a long time, single-purpose and closed systems, characterized by hardware resource constraints and real-time requirements. Nowadays, their functionality is ever-growing, coupled with an increasing complexity and heterogeneity. Embedded applications increasingly demand employment of general-purpose operating systems (GPOSs) to handle operator interfaces and general-purpose computing tasks, while simultaneously ensuring the strict timing requirements. Virtualization, which enables multiple operating systems (OSs) to run on top of the same hardware platform, is gaining momentum in the embedded systems arena, driven by the growing interest in consolidating and isolating multiple and heterogeneous environments. The penalties incurred by classic virtualization approaches is pushing research towards hardware-assisted solutions. Among the existing commercial off-the-shelf (COTS) technologies for virtualization, ARM TrustZone technology is gaining momentum due to the supremacy and lower cost of TrustZone-enabled processors.

Programmable system-on-chips (SoCs) are becoming leading players in the embedded systems space, because the combination of a plethora of hard resources with programmable logic enables the efficient implementation of systems that perfectly fit the heterogeneous nature of embedded applications. Moreover, novel disruptive approaches make use of field-programmable gate array (FPGA) technology to enhance virtualization mechanisms.

This master's thesis proposes a hardware-software co-design framework for easing the economy of addressing the new generation of embedded systems requirements. ARM TrustZone is exploited to implement the root-of-trust of a virtualization-based architecture that allows the execution of a GPOS side-by-side with a real-time OS (RTOS). RTOS services were offloaded to hardware, so that it could present simultaneous improvements on performance and determinism. Instead of focusing in a concrete application, the goal is to provide a complete framework, specifically tailored for Zynq-base devices, that developers can use to accelerate a bunch of distinct applications across different embedded industries.

Resumo

Os sistemas embebidos foram, durante muitos anos, sistemas com um simples e único propósito, caracterizados por recursos de hardware limitados e com cariz de tempo real. Hoje em dia, o número de funcionalidades começa a escalar, assim como o grau de complexidade e heterogeneidade. As aplicações embebidas exigem cada vez mais o uso de sistemas operativos (OSs) de uso geral (GPOS) para lidar com interfaces gráficas e tarefas de computação de propósito geral. Porém, os seus requisitos primordiais de tempo real mantém-se. A virtualização permite que vários sistemas operativos sejam executados na mesma plataforma de hardware. Impulsionada pelo crescente interesse em consolidar e isolar ambientes múltiplos e heterogéneos, a virtualização tem ganho uma crescente relevância no domínio dos sistemas embebidos. As adversidades que advém das abordagens de virtualização clássicas estão a direcionar estudos no âmbito de soluções assistidas por hardware. Entre as tecnologias comerciais existentes, a tecnologia ARM TrustZone está a ganhar muita relevância devido à supremacia e ao menor custo dos processadores que suportam esta tecnologia.

Plataformas hibridas, que combinam processadores com lógica programável, estão em crescente penetração no domínio dos sistemas embebidos pois, disponibilizam um enorme conjunto de recursos que se adequam perfeitamente à natureza heterogénea dos sistemas atuais. Além disso, existem soluções recentes que fazem uso da tecnologia de FPGA para melhorar os mecanismos de virtualização.

Esta dissertação propõe uma framework baseada em hardware-software de modo a cumprir os requisitos da nova geração de sistemas embebidos. A tecnologia TrustZone é explorada para implementar uma arquitetura que permite a execução de um GPOS lado-a-lado com um sistemas operativo de tempo real (RTOS). Os serviços disponibilizados pelo RTOS são migrados para hardware, para melhorar o desempenho e determinismo do OS. Em vez de focar numa aplicação concreta, o objetivo é fornecer uma framework especificamente adaptada para dispositivos baseados em System-on-chips Zynq, de forma a que developers possam usar para acelerar um vasto número de aplicações distintas em diferentes setores.

Contents

Abstract vii

Resumo ix

List of Figures xvi

List of Tables xvii

List of Listings xix

Glossary xxi

1 Introduction 1

1.1 Goals . . . 3

1.2 Document Structure . . . 4

2 Background, Context and State of the Art 7

2.1 Operating Systems . . . 7

2.1.1 General-purpose Operating Systems . . . 7

2.1.2 Real-time Operating Systems . . . 8

2.1.3 Hardware-Software Real-Time Operating Systems . . . 10

2.2 Virtualization . . . 12

2.2.1 Hardware-assisted Virtualization . . . 14

2.3 ARM TrustZone . . . 15

2.4 FPGA Technology . . . 16

2.4.1 Hardware Description Languages . . . 17

2.5 Multicore Processor Technology . . . 17

2.5.1 Symetric Multiprocessing . . . 18

2.5.2 Asymetric Multiprocessing . . . 19

2.6 Mixed Criticality Systems . . . 19

2.6.1 SafeG . . . 19

2.6.2 NOVA/MINI-NOVA 20
2.6.3 TZVisor Project 21

3 Platform and Tools 23
3.1 Zynq Platform 23
3.1.1 Security 24
3.2 Linux 25
3.2.1 HDMI Transmitter driver 26
3.3 FreeRTOS 26
3.3.1 Structure 27
3.3.2 Task Management 27
3.3.3 Scheduler 30
3.3.4 Synchronization Mechanisms 30
3.3.5 Software Timers 31
3.4 LTZVisor 31
3.4.1 General Architecture 31
3.4.2 LTZVisor-AMP: Multicore Extension 33
3.5 Benchmarks 33
3.5.1 Thread-Metric 34
3.5.2 LMBench 34

4 Trust SecSi Code: A TrustZone-assisted Secure Silicon Co-design Framework 37
4.1 Overview 37
4.2 Secure VM (Hardware-Software RTOS) 39
4.2.1 System Structure 39
4.2.2 SysTick Timer 41
4.2.3 Task Management 42
4.2.4 Scheduler 49
4.2.5 Software Timers Service 51
4.2.6 Synchronization Service 52
4.2.7 Secure Guest OS modifications 53
4.3 Non-Secure VM (Linux with HDMI video output) 54
4.3.1 Non-Secure Guest OS modifications 56
4.4 LTZVisor Modifications 57
4.4.1 Secure Co-Design Guest Decoupling 58
4.4.2 Multi-Processing 59

5 Evaluation **61**

5.1 Engineering effort . . . 61

5.2 Memory Footprint . . . 62

5.3 Hardware Costs . . . 62

5.4 Performance . . . 63

5.4.1 Secure VM (Hardware-Software RTOS) . . . 63

5.4.2 Non-Secure VM (Linux) . . . 66

6 Conclusion **69**

6.1 Future Work . . . 69

References **75**

List of Figures

2.1 RTOS Kernel . . . 10
2.2 Hypervisor Types . . . 14
2.3 Mini-NOVA Architecture Overview . . . 21
2.4 μRTZVisor Architecture Overview . . . 22

3.1 Zynq-7000 SoC . . . 24
3.2 Task state machine . . . 28
3.3 Task Control Block Structure . . . 29
3.4 LTZVisor system architecture. . . . 32
3.5 LTZVisor AMP architecture. . . . 33

4.1 Trust SecSi CoDe system architecture. . . . 38
4.2 Hardware-Software Co-design RTOS architecture . . . 40
4.3 AXI Timer and the Counter IP . . . 41
4.4 Task Manager IP. . . . 42
4.5 Create and Delete Task Interface. . . . 44
4.6 Task List and Priority List. . . . 44
4.7 Resume and Suspend Task Interface. . . . 45
4.8 Delay and Abort Delay Task Interface. . . . 46
4.9 Temporal Blocked List. . . . 46
4.10 Bit Array of Priorities . . . 47
4.11 Scheduler Interface. . . . 48
4.12 Scheduler IP. . . . 49
4.13 Scheduler State Machine. . . . 50
4.14 Comparison between different scheduler behaviours during system ticks . . . 51
4.15 Software Timers IP. . . . 52
4.16 Software Timers List. . . . 52
4.17 Semaphore IP. . . . 53
4.18 Semaphore Lists. . . . 54
4.19 ADV7511 reference design . . . 55

5.1 Engineering effort results 61
5.2 TrustSecSiCoDe framework hardware costs 63
5.3 Thread-Metric benchmarks results 64
5.4 Context switch (vTaskSwitchTask()) latency results. 65
5.5 xIncrementTick() latency results 65
5.6 LMBench arithmetic operations latency (lat_ops) benchmark results. 66
5.7 LMBench memory bandwith (bw_mem) benchmark results 67

6.1 Trust SecSi Code for Industrial Control Systems. 70

List of Tables

5.1 TrustSecSiCode memory footprint (bytes) 62

List of Listings

4.1 FreeRTOS_Tick_Handler() modification 40
4.2 Offtsets Declaration 56

Glossary

AMBA	Advanced Microcontroller Bus Architecture
AMP	Asymmetric Multi-Processing
AP	All Programmable
API	Application Programming Interface
APU	Application Processing Unit
ASIC	Application Specific Integrated Circuits
AXI	Advanced eXtensible Interface
BOM	Bill Of Materials
BRAM	Bus RAM
BSP	Board Support Package
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
DBT	Dynamic Binary Translation
DDR	Double Data Rate
DMA	Direct Memory Access
DOS	Denial-Of-Service
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
ESRG	Embedded System Research Group
FF	Flip-flops
FIFO	First In, First Out
FIQ	Fast Interrupt Request
FPGA	Field-Programmable Gate Array
FPSoC	Programmable SoCs, or Field program-mable SoCs
FPU	Floating-Point Unit
GIC	Generic Interrupt Controller
GPOS	General-Porpose Operating System
GUI	Graphical User Interface
HDL	Hardware Description Language

HDMI	High-Definition Multimedia Interface
hS IP	hardware Scheduler IP
hSmphr IP	hardware Semaphore IP
hST IP	hardware Software Timers IP
hTM IP	hardware Task Manager IP
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Intellectual Property
IPC	Inter-Partition Communication
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISR	Interrupt Service Routine
LoC	Lines of Code
LTZVisor	Lightweight TrustZone-assisted Hypervisor
LUT	LookUp Tables
LUTRAM	LookUp Tables RAM
MCS	Mixed-Criticality System
MMCM	Mixed-Mode Clock Manager
MMU	Memory Management Unit
MPMU	Message Passing Management Unit
MPU	Memory Protection Unit
OS	Operating System
PAL	Programmable Array Logic
PL	Programmable Logic
PS	Processing System
ROM	Read-Only Memory
RTOS	Real-time Operating System
SMC	Secure Monitor Call
SMP	Symmetric Multi-Processing
SoC	System-on-a-chip
SRAMs	Static Random-Access Memory
SWAP-C	Size, Weight, Power and Cost
TCB	Trusted Computing Base
Trust SecSi CoDe	TrustZone-assisted Secure Silicon Co-design
TZASC	TrustZone Address Space Controller
TZMA	TrustZone Memory Adapter
TZPC	TrustZone Protection Controller

VGA	Video Graphics Array
VM	Virtual Machine
VMM	Virtual Machine Monitor
XSDK	Xilinx Software Development Kit

1. Introduction

Embedded systems were, for a long time single-purpose systems, showing simple functionality and interfaces and little to no need for communication channels or compatibility with other devices, characterized by constraints hardware resources and real-time requirements. Therefore, they used to have reduced software complexity. Nowadays, their functionality is ever-growing, coupled with an increasing complexity that is associated with a higher number of bugs and vulnerabilities. Moreover, the pervasive connectivity of these devices in the modern Internet of Things (IoT) era significantly increases their attack surface [POP+17]. Due to their myriad of applications and domains, ranging from consumer electronics to aerospace control systems, there is an increasing reliance on embedded devices that often have access to sensitive data and perform safety-critical operations [Hei11, BBO+16].

Platform virtualization, which enables multiple operating systems (OSes) to run on top of the same hardware platform, is gaining momentum in the embedded systems arena, driven by the growing interest in consolidating and isolating heterogeneous environments [Hei11]. This technology is well established in the server and desktop domains, providing benefits such as service consolidation, load balancing and power management. While in industrial control or automotive, systems virtualization has been used to integrate real-time control functionalities with high-level or infotainment environments [BBO+16], in aeronautics and aerospace, virtualization provides isolation for safety-critical components [PTM16b]. Without virtualization these systems would commonly be distributed across multiple physically interconnected hardware platforms. Applying virtualization to the embedded domain brings a smaller form factor and a reduced bill of materials (BOM). All of these advantages minimize overall costs and consequently improve profit margins [PTM16a]. Despite the differences among several embedded industries, all share an upward trend for integration, due to the common interest in building systems with reduced size, weight, power and cost (SWAP-C) budget [BBO+16, PPG+17].

According to the virtualization taxonomy [SGB+16], virtualization techniques can be classified mainly into two types: full-virtualization and paravirtualization. Full-virtualization is a virtualization technique in which the hypervisor presents to guest OSes an exact replica of the underlying hardware. Guest OSes do not require any modification to their kernel code, and are completely

unaware of the system's virtualization. Privileged instructions trap to the hypervisor and are emulated on behalf of the guest OS. This trap-and-emulate procedure has been, for a long time, supported by dynamic binary translation (DBT) techniques, which result in large overheads, compromising either the real-time capability of embedded devices or their security [CSA+16]. This drawbacks led the processor industry's Big Players to extend their ISAs to support hardware-assisted full-virtualization. The advent of hardware virtualization extensions (e.g., Arm Virtualization Extensions, Intel Virtualization Technology) has resulted in new virtualization solutions capable of providing efficient hypervisors and real-time guarantees [DN14, ZMH15]. Para-virtualization, in contrast, requires modification of the guest OSes. The privileged instructions are replaced by specific hypervisor calls (hypercalls), to request services directly from the hypervisor. The guestOSes communicate directly with the hypervisor, instead of implicitly and unknowingly invoking the hypervisor through virtual resource access, resulting in significant performance advantages at the cost of a considerably higher engineering effort.

Arm TrustZone technology, although implemented for security purposes, has enabled a specialized, hardware-assisted form of system virtualization [FLWH10]. With virtual hardware support for dual world execution, an extra processor mode (i.e., the monitor mode), and other TrustZone features like memory segmentation, it is possible to provide time and spatial isolation between execution environments. Basically, the non-secure software runs inside a VM whose resources are completely managed and controlled by a hypervisor running in the secure world. TrustZone-assisted virtualization is not particularly considered full-virtualization neither paravirtualization, because, although guestOSes can run without modifications on the non-secure world side, they need to co-operate regarding the memory map and address space they are using. There are several open-source TrustZone-based solutions for virtualization, such as SafeG [SHT10] and LTZVisor [PPG+17].

Reconfigurable platforms, hybrid platforms, programmable systems-on-a-chips SoCs, or Field programmable SoCs (FPSoC) are different designations describing the same concept: technologies for the implementation of digital systems combining software components with hardware accelerators [PRAM17]. The Xilinx Zynq-7000 SoC family integrates the software programmability of an ARM-based processor with the hardware programmability of an field programmable gate array (FPGA), enabling hardware acceleration while integrating central processing unit (CPU), digital signal processor (DSP), and mixed signal functionality on a single device. Programmable SoCs are becoming leading players in the embedded systems space, because the combination of a plethora of hard-wired resources with configurable logic enables the efficient implementation of systems that perfectly fit the heterogeneous nature of embedded applications [PRAM17].

Reconfigurable hardware promises to help mitigating the rigid constraints (e.g., real-time, performance, power consumption, safety, and security) typical of the embedded domain. The applicability of such technology ranges from offloading specific applications to hardware [PRAM17,

CI18], to offloading real-time operating system (RTOS) services [GGP+16] or even hypervisor features [XPN15] to enable guest to access the reconfigurable logic. RT-SHADOWS [GGP+16] and ReconOS [LP09] are just a few examples of operating systems which take advantage of reconfigurable hardware technology to improve system performance, determinism and real-time. Mini-Nova [XPN15] exploits the potential of CPU-FPGA systems by proposing a virtualization framework taking advantage of both virtualization and dynamic partial reconfiguration (DPR) techniques. The innovative open-source Xilinx Python-based framework [CI18, G. 18] provides an infrastructure that binds Programmable SoC hardware to the Python environment simplifying the addition of ad-hoc hardware modules in the programmable logic.

In addition to the bound that most embedded systems have with real-time constraints, embedded applications also require general-purpose systems to handle operator interfaces, databases, and general-purpose computing tasks. Due to the heavy and complex nature of these systems, a good solution to improve the performance of these systems is to use multicore technology.

This project presents a hardware-software co-design framework which explores several capabilities of Zynq-based platforms for easing the economy of building the new generation of embedded systems devices. To the best of our knowledge, no existing framework targeting the Zynq SoC is able to fully and simultaneously explore TrustZone, multiprocessing, as well as an hardware-accelerated RTOS via reconfigurable logic. Arm TrustZone technology is exploited to implement the root of trust of a virtualization-based architecture that allows the execution of a General Purpose Operating System (GPOS) side-by-side with a real-Time OS. RTOS services were offloaded to the FPGA fabric, so that we can present simultaneous improvements on performance and determinism. Instead of focusing in a concrete application, we provide a complete framework, specifically tailored for Zynq-base devices, that developers can use to accelerate a bunch of distinct applications across different embedded industries.

1.1 Goals

The main goal of this thesis goes towards the implementation of a hardware-software codesigned virtualization framework. The cornerstone of this virtualization framework development is the in-house hypervisor LTZVisor, given that a from-scratch design and implementation would be overkill. LTZVisor [PPG+17] is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticallity systems. LTZVisor provides a virtualization solution based on the two virtual execution environments provided by the TrustZone hardware, in which implements a dual-guest OS configuration: the secure world hosts the RTOS and the hypervisor while the non-secure world is assigned to the GPOS. Notwithstanding, it is believed that the exploration of reconfigurable computing can bring several benefits and present a promising approach.

In this sense, this thesis aims at accomplish the following goals:

1. Enhance the perfomance and multimedia capability of the non-secure OS, by setting up the multicore configuration and configuring audio/video interface. These goals can be splitted in two parts:

   (a) Enrich the multimedia capabilities of the general-purpose OS guest by incorporating a high-definition multimedia interface (HDMI) transmitter driver. This will allow the system to handle operator interfaces and other multimedia services.

   (b) Enable the multi-processing configuration so that the GPOS runs on a dual-core platform. This will impose changes on the hypervisor, since the LTZVisor as it stands, only allows the execution of single-core VMs.

2. In behalf of ensuring the hard real-time requirement of critical software some RTOS services of the secure guest will be deployed to the reconfigurable hardware. This overall goal can be into more fine-grained objectives:

   (a) Perform a comprehensive survey on FreeRTOS, an open source RTOS kernel, identifying which process are a major source of overhead for the Trusted computing base (TCB) and/or impose latency and indeterminism;

   (b) Deployment of the processes that were classified has major sources of jitter and overhead to the programmable logic. Consequently, give these offloaded services the same level of security of the associated guest.

3. Integrate the operating systems designed in the base hypervisor, LTZVisor. Assigning the respective privilege level to them and all their components. Taking into account the necessary changes to the current scheme of the LTZVisor in order to integrate the multicore general-purpose guest and the hardware-software co-design real-time system;

4. Evaluate the hypervisor's implemented artifacts, namely in terms of performance and security;

5. Evaluate the conceived work regarding the hypervisor and its guests, namely identifying the gains of the used approach as well as the counter-parts.

1.2 Document Structure

The present thesis is divided into 6 different chapters. This chapter, Chapter 1, described the contextualization for the problem addressed, the motivations for its implementation. Chapter 2 overviews the basic concepts and background knowledge to understand the developed work. It covers the technologies used during the development of the thesis, with emphasis on virtualization, multicore and reconfigurable hardware technologies. Chapter 3 describes the tools and

platform. It starts by overviewing LTZVisor, and then focus on describing FreeRTOS. In addition, the architecture of , since the hardware-software real-time VM implementation will be based on this operating system. Chapter 4 addresses the design and implementation of Trust SecSi CoDe framework. It starts by describing the goals for the framework, specifying the enhancements to the current LTZVisor architecture. Then it addresses the implementation of the hardware-assisted RTOS as well as the modifications needed to support the multimedia GPOS. Additionally a description of the necessary modifications on the hypervisor in order to host the hardware-assisted secure VM and the multicore non-secure VM. Chapter 5 presents the evaluation, focusing on memory footprint, performance, and real-time. Finally, Chapter 6 provides a summary of this thesis and draws conclusions regarding open issues of Trust SecSi CoDe, outlining a roadmap for future improvements and research.

2. Background, Context and State of the Art

The chapter 2 intent to provide the necessary fundamental concepts and related work to understand and contextualize the developed project. First, it overviews and compares general-purpose operating systems and real-time systems. Second, it focus on explaining and describing virtualization. Then it provides a description of the TrustZone, field-programmable gate array (FPGA) and multiprocessing, focusing on how these technologies are exploited to implement the root-of-trust of a virtualization-based architecture. Lastly, it presents the state of art in mixed criticality and hybrid hardware-software systems.

2.1 Operating Systems

Operating system (OS) is the software component responsible for the management and coordination of activities, as well as the sharing of the system resources. Operating systems tend to alleviate the complexity of embedded systems development by providing several different mechanisms, such as multithreading, semaphores, timers and interrupt handling, in order to abstract and coordinate the overall system behaviour.

Embedded systems often have the need for deterministic response to real-time events. To support the requirement for determinism, embedded applications typically use real-time operating systems (RTOSs). Embedded applications also employ general-purpose OSes to handle operator interfaces, databases, and general-purpose computing tasks. System architectures that can combine both types of processing on the same platform can save costs by eliminating redundant hardware.

2.1.1 General-purpose Operating Systems

A general-purpose operating system (GPOS) is a fully-featured operating system intended to provide a good user experience. These OSs are designed to optimize average performance of application programs at the expense of predictability. GPOSs typically provide a non-deterministic

response, where there are no guarantees as to when each task will complete, but they will try to stay responsive to the user.

In applications that have several distinguish purposes and do not impose time critical, general-purpose operating systems are the most suitable OS, due to GPOSs commonly uses a fairness policy to dispatch threads and processes. Such policy enables the high overall throughput required by user interface and server applications, but offers no guarantees that high-priority, time critical threads will execute in preference to lower-priority threads. Examples of GPOSs include the well-known Windows, Linux and MacOS.

A good example of an operating system used in the embedded systems world is Embedded Linux OS. Its application is usually motivated by the availability of device support, file-systems, network connectivity, and UI support [OK13b] besides being open-source and free to use. These features can be available in a RTOS, but often with less broad support, or at additional cost or integration effort.

2.1.2 Real-time Operating Systems

Most embedded systems are bound to real-time constraints [SR04]. Real-time constraint means that the system behaviour depends not only on the logical results of the computation, but also on the physical instant at which these results are produced. Basically, in strict real-time systems a late result is not just late but wrong. An operating system which is capable of taking care of real-time constraints is called a real-time operating system (RTOS).

The real-time strictness depends on the purpose of the application and it can be hard real-time or soft real-time [OK13a]. In hard real-time systems, tasks have to be performed not only correctly but on time and a failure to meet a deadline is considered to be a fatal fault and lead to disastrous consequences. In a soft real-time system, tasks are performed by the system as fast as possible, and if possible, in its specific time. A deadline miss on a soft real-time system is undesirable but will not cause a serious harm, however could lead an array of deadline misses until failure. Most real time systems have a combination of soft and hard requirements. Real-time software applications are typically more difficult to design than non-real-time applications.

Following, some important real-time operating system concepts:

- Shared resource is any entity used by a task that can be used by other tasks. Each task should gain exclusive access to the shared resource to prevent data corruption. The code that needs to be treated indivisibly is called critical section of code. To ensure this section of code is not interrupted, interrupts are typically disabled before the critical code is executed and enabled when the critical code is finished.

- The process of scheduling and switching the CPU between several tasks is called multi-tasking. Multitasking maximizes the utilization of the CPU and also provides for modular

construction of applications. One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent to real-time applications. Application programs are typically easier to design and maintain if multitasking is used.

- The design process for a real-time application involves splitting the work to be done into tasks which are responsible for a portion of the problem. Each task is assigned a priority, its own set of CPU registers, and its own stack area. Each task typically is an infinite loop that can be in any one of five states: SUSPEND, READY, RUNNING, WAITING, or INTERRUPTED. The SUSPEND state corresponds to a task which resides in memory but has not been made available to the multitasking kernel. A task is READY when it can execute but its priority is less than the currently running task. When the tasks is in RUNNING state it has control of the CPU. A task is WAITING when it requires the occurrence of an event (waiting for a time to expire, an I/O operation to complete, a shared resource to be available, a timing pulse to occur, etc.). Finally, a task is INTERRUPTED when an interrupt has occurred and the CPU is in the process of servicing the interrupt.

- The process of saving the current task's context and restore the new task's context from its storage area and then resume execution of the new task's code, is called context switch or a task switch. Context switching adds overhead to the application. The more registers a CPU has, the higher the overhead. The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU.

- The kernel is the part of a multitasking system responsible for the management of tasks and the communication between tasks. The fundamental service provided by the kernel is context switching. The use of a real-time kernel will generally simplify the design of systems by allowing the application to be divided into multiple tasks managed by the kernel. However, the kernel add overhead to the system because it requires extra memory for the kernel data structures but most importantly, each task requires its own stack space which has a tendency to require large amount of RAM RAM quite quickly.

The basic services required in a RTOS kernel are illustrated in the Figure 2.1. Task Management is shown at the center of the RTOS kernel. The RTOS's task management is responsible for task scheduling and provide a set of services that handle the tasks behaviour. The scheduler is the part of the kernel responsible for determining which task will run next. Most real-time kernels are priority based. Each task is assigned a priority based on its importance. The priority for each task is application specific. In a priority-based kernel, control of the CPU will always be given to the highest priority task ready-to-run.

The second main section of an RTOS kernel is inter-task communication and synchronization, know as inter-process communication (IPC). Without IPC and synchronization mechanisms

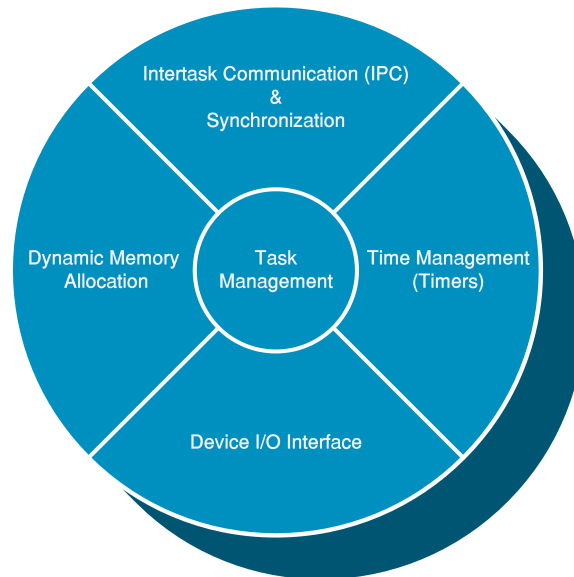


Figure 2.1: RTOS Kernel

tasks could communicate corrupted information or otherwise interfere with one another. Most RTOS kernels offer a variety of inter-task communication and synchronization mechanisms that may include message queues, pipes, semaphores, mailboxes, event groups and asynchronous signals. The dynamic memory allocation section in RTOS typically uses pools memory allocation mechanism.

RTOS emphasize predictability, efficiency, and include features to support timing constraints. In contrast to GPOS, this type of OS attempt to minimize latency rather than maximize throughput. To reduce the run-time overheads, the kernel has to have quick response to external interrupts and context switches between tasks, as well as small size. In order to meet the timing requirements, the kernel must support multi-tasking, provide priority-based preemptive scheduling and synchronization mechanisms. Such rigid constraints and requirements can be more easily achieved by using hardware-software co-design architectures.

2.1.3 Hardware-Software Real-Time Operating Systems

Hardware-software co-design is the concurrent design of both hardware and software of the system by taking into consideration the cost, energy, performance, speed and other parameters of the system. Hardware-software OSes are systems that has some of the services implemented in hardware and such services can be specific application tasks or even kernel functionalities.

The presence of an operating system introduces new sources of latency and lack of determinism [SWP04]. Lack of determinism is caused by response time variation (jitter), another of the least desired characteristics of a RTOS. Most of jitter sources comes from RTOS' dynamic data structures and their management and traversal. Migration to hardware of software tasks and

services, addresses the jitter issues leading to solutions able to cope with these increasingly strict requirements. Migration of RTOS services, such as scheduling, task management and synchronization services, to dedicated hardware modules, allows the RTOS to meet metrics requirements and consequently, provides increased system performance.

There is a large number of projects/publications focusing on attempt to make an embedded system capable of providing hardware support to a RTOS, typically following a component based model [vOvdLKM00, LMP+05]. Back in 1991, efforts were being made to integrate operating systems' functionalities in hardware, providing microprocessors with the ability to offer support to the RTOSs' primitives [Lin91]. The emergence of high capacity reconfigurable FPGAs at a lower cost, renewed the interest in this field in recent years [SOLMTS04, PS12]. Nevertheless, to the best of our knowledge, there is no work that has linking hardware-software co-design OS with virtualization technologies.

The Hybrid Real-Time Operating System [GPG+15], for example, is a hardware-software co-designed RTOS that takes advantage of hardware accelerators to improve the RTOS performance and determinism. HcM-FreeRTOS [QLG+15] makes use of the ARM Generic Interrupt Controller (GIC) to offload FreeRTOS kernel components to a commercial-off-the-shelf (COTS) multicore hardware. RT-SHADOWS [GGP+16] and HartOS [LASS12] are more examples of operating systems which take advantage of reconfigurable hardware technology to improve system performance, determinism and real-time guarantees.

2.1.3.1 Hthreads

The Hthreads operating system consists of a multitasking RTOS kernel that integrates a hybrid task programming model developed for hybrid systems. This hybrid task programming model is intended to provide a high level programming environment where programmers are able to access the field-programmable gate array (FPGA) resources through the use of a software programming model [APA+05].

This operating system supports up to 256 software tasks, 256 hardware tasks, 64 blocking semaphores, 64 binary semaphores, preemptive priorities, round robin and First In First Out (FIFO) scheduling algorithms, 256 tasks are allowed to run simultaneously with 128 priority levels and use up to 64 traffic lights. Functionalities of the operating system, such as task manager, scheduler, semaphores, and interrupt controller are mapped to hardware, resulting in shorter interrupt latencies, fewer CPU context shifts, and reduced jitter.

2.1.3.2 ReconOS

In the ReconOS operating system, the programming model and system architecture provides unified operating system services for software and hardware execution functions and a standardized interface for integrating custom hardware accelerators. This OS was based on the eCos OS

[SOSA08].

ECos is an open-source RTOS intended for embedded applications, it is highly configurable which enables this operating system to be adapted to the requirements of the applications, offering the best possible run-time performance and an optimized footprint for the hardware resources [SOSA08].

In ReconOS, all hardware tasks have access to all relevant operating system services offered by eCos in a transparent way. For example, the tasks they communicate do not need to know if their communication partners are running as hardware tasks or software [SOSA08], and this system can simultaneously perform software and hardware tasks. The existing operating system layer provides a symmetry between software and hardware tasks that offers benefits for reconfigurable hardware systems, that can initially develop the entire system using software implementation and later the parts of the application with critical performance can be migrated for one-to-one hardware tasks. The software tasks have a sequential execution and to use operating system services the task simply calls the corresponding function of the operating system library but the hardware tasks present another paradigm because they have a parallel execution, which means that there is no controlled flow and no apparent notion of calling a function from the operating system. In order to present itself as a possible unified programming model for the user, the approach of structuring a hardware segment was followed so that all interactions with the operating system were managed by a sequential state machine, in two processes: synchronization state machine and user logic.

2.2 Virtualization

Virtualization refers to the creation of an integral or partial abstraction in software of the physical hardware by providing an environment that abstract the underlying hardware platform and enables the safe sharing of available resources. Thereby, the software executed on these virtual machines is separated from the underlying hardware resources. Consequently, the system is transformed so that it appears as an emulated, different machine, or even a set of multiple machines, as much as possible a replicate of the original one.

Virtualization is a widely used because encompasses a wide range of technologies and can mean very different things in different contexts. In this context, virtualization refers to the technology that allows the co-existence of multiple operating systems (OSes) environments on a single physical platform. This method introduce an additional software layer to provide this abstraction, often called Virtual Machine Monitor (VMM) or hypervisor [AH10, SGB+16, Kai09, GZ12]. To guaranteed isolation, the hypervisor usually runs with full privileges while the virtual machines (VMs), also called guests, are pushed to lower privilege layers. Therefore, the hypervisor has full control of the underlying hardware platform. Usually the full privilege mode of the system is associated to

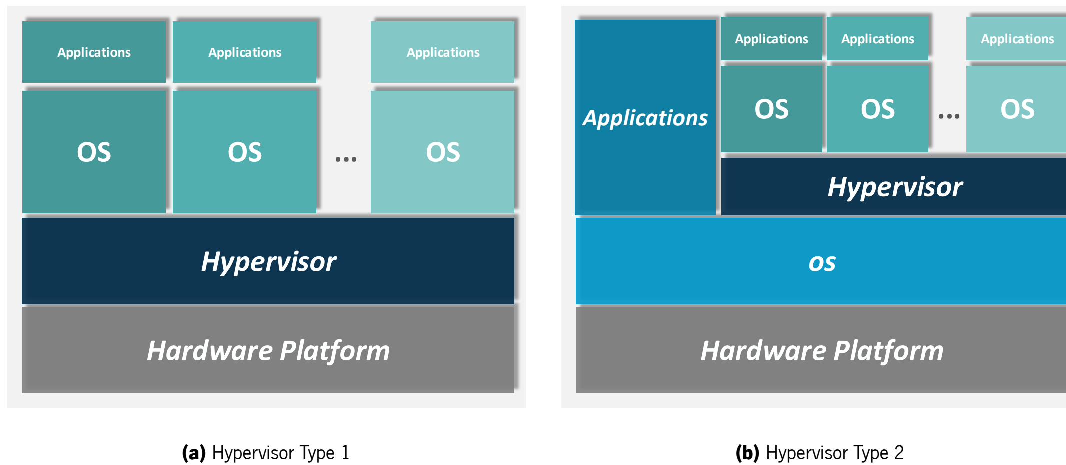
the exclusive control of certain peripherals that are used to apply temporal and spatial isolation, such as timers and memory controllers. The exclusive assignment of the hypervisor privilege mode is considered to be one of the major challenges in implementing virtualization techniques on embedded systems.

Besides the benefit of providing multiple concurrent VM environments while multiplexing the underlying resources, virtualization brings benefit of isolation and encapsulation. Consequently, improve system's safety and security, two key requirements of current embedded applications. Virtualization technology also deliver application consolidation benefits, which helps in reducing production costs by lowering the amount of required hardware to support all the desired functionality, and to reduce energy consumption by load-balancing across clusters [AH10, Hei08, AG09].

The hypervisor should be responsible for the scheduling management of the operating systems guests,, likewise an OS should also perform the context switch and the changes necessary to scheduling possibility. According to the methodology of how they are implemented, hypervisors can be classified by two types, as shown in the Figure 2.2.

- **Type 1** hypervisors execute directly above the hardware; Hypervisors, or bare-metal Hypervisors, have direct access to the hardware layer and manage the execution permissions of every system component, which means that all the hardware accesses are to be mediated and controlled by the VMM. As a consequence of this being the most privileged software on the running platform, the performance degradation of guests OSes will only be influenced by the performance of the Hypervisor itself, making this type of VM more suited to systems that must meet time constraints.
- **Type 2** hypervisors run on top of an OS [AH10, SGB+16, GZ12], and are also called hosted hypervisors. This type of VMM usually does not have permissions to access and perform any operation on the hardware directly since those responsibilities usually rest in the system software that runs below the VMM, which usually results in lower performance ratings compared to the type-1 hypervisors.

Regardless of their type, there are mainly two different approaches towards a virtualization solution: Full Virtualization and Para-Virtualization. Full virtualization schemes simulate the hardware environment. Therefore, VMs do not need modifications to execute. In contrast, paravirtualization demand certain changes at the OS level in order to execute in the virtual machine, replacing critical instructions with system calls to the hypervisor (hypercalls). There are benefits and disadvantages in each approach. A full-virtualization configuration require little engineering effort for VM's deployment, this is more suitable for the deployment of single binaries, when using proprietary OSes for which the source code is not available. Notwithstanding, classic full-virtualization shows poor performance and high complexity due to the high-frequency mode courses, need for



(a) Hypervisor Type 1

(b) Hypervisor Type 2

Figure 2.2: Hypervisor Types

software to impersonate hardware in trap-and-emulate and DBT. Para-virtualized solutions usually incur better performance, as the performed changes may also aim to remove unnecessary operations; however, it typically has associated a higher. involve a higher engineering effort of manually modify each guest source-code. Besides, the lack of a standardized VMM interface for VMs, the need to keep up with OS versions and, of course, the obligation for the availability of OS source code [PKR+13, SK10].

2.2.1 Hardware-assisted Virtualization

Due to the gradual growth of its popularity even with the known ones disadvantages adjacent to software-assisted virtualization solutions, the development and use of hardware-assisted virtualization techniques gained a new focus. These techniques make use of integrated hardware extensions in the processor architectures itself to implement virtualization solutions without all or part of the software-assisted disadvantages.

Extensions are usually characterized by adding to the processor a new level of privilege,which has higher privilege of execution than the kernel and user modes. Depending on the extension itself, these primitives may include additions or changes to the components of the primitive processor to produce a authentic virtual CPUs in hardware. In between platform peripherals, general purpose registers or coprocessors, MMUs, caches, and the memory or memory controller itself.

Within the hardware extensions for using virtualization in the domain embedded systems include the ARM Virtualization Extensions (ARM VE) and Intel Virtualization Technology (Intel VT) [DN14, ZMH15].

2.3 ARM TrustZone

Arm TrustZone consists of hardware security extensions introduced into Arm application processors (Cortex-A) in 2004 [ARM09]. More recently, TrustZone has been adapted to cover the new generation of Arm microcontrollers (Cortex-M) [PS]. TrustZone follows a system-wide approach to security. In the remainder of this section, when describing TrustZone, the focus will be on the specificities of this technology for Cortex-A processors.

The Arm TrustZone technology is centered around the concept of protection domains named secure world and non-secure world. The software executed by the processor runs either in the secure world or in the non-secure world. A new 33rd processor bit, the Non-Secure (NS) bit, indicates in which world the processor is currently executing. To preserve the processor state during the world switch, TrustZone adds an extra processor mode: monitor mode. This mode is completely different from other modes because, when the processor runs in this privileged mode, the state is always considered secure, independently of the NS bit's state. Software stacks in the two worlds can be bridged via a new privileged instruction-Secure Monitor Call (SMC). The monitor mode can also be entered by configuring it to handle interrupts and exceptions in the secure side. To ensure a strong isolation between secure and non-secure states, some special registers are banked, while others are either totally unavailable for the non-secure side.

The TrustZone Address Space Controller (TZASC) and the TrustZone Memory Adapter (TZMA) extend TrustZone security to the memory infrastructure. TZASC can partition the dynamic random-access memory (DRAM) into different secure and non-secure memory regions, by using a programming interface which is only accessible from the secure side. By design, secure world applications can access normal world memory, but the reverse is not possible. TZMA provides similar functionality but for off-chip read-only memory (ROM) or static random-access memory (SRAMs). Both the TZASC and TZMA are optional and implementation-specific components on the TrustZone specification. In addition, the granularity of memory regions depends on the system on chip (SoC). In Zynq-7000 devices, the memory subsystem includes the TZASC, which allows the (dynamic) configuration of the security state of memory segments with a granularity of 64 MB [Xil14]. The TrustZone-aware memory management unit (MMU) provides two distinct MMU interfaces, enabling each world to have a local set of virtual-to-physical memory address translation tables. Furthermore, three different exception vectors exist: the secure-world vector, the non-secure world vector and the monitor vector. The isolation is still available at the cache-level because processor's caches have been extended with an additional tag that signals in which state the processor accessed the memory.

System devices can be dynamically configured as secure or non-secure through the TrustZone Protection Controller (TZPC). The TZPC is also an optional and implementation-specific component on the TrustZone specification. In Zynq-7000 devices, it is possible to (dynamically) configure

the security state of several SoC (e.g., Triple Timer Counter, Ethernet controller) devices, as well as access to FPGA AXI ports as secure or non-secure [Xil14]. 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. The interrupt controller supports interrupt prioritization, allowing the configuration of secure interrupts with a higher priority than the non-secure interrupts. Such configurability prevents non-secure software from performing a denial-of-service (DOS) attack against the secure side.

2.4 FPGA Technology

Application specific integrated circuits (ASICs) are non-programmable but highly integrated, small, fast, and energy efficient. Also, ASICs make the process of copy/reverse engineering more difficult for both hardware and software perspective. In contrast, due the non-programmability nature, these chips need to be redesigned and refactoring all over again if modifications have to be made.

Field-programmable gate array (FPGA) technology consists of logic and interconnect resources that permit to configure an uncommitted chip into the desired functions for different applications. These chips can be configured to implement custom hardware functionalities without going through the long fabrication process of custom ASIC design. FPGA technology continues to gain momentum since their invention by Xilinx in 1984. FPGA chip adoption across all industries is driven by the fact that FPGAs combine the best parts of ASICs and processor-based systems.

FPGAs provide hardware-timed speed and reliability, but they do not require high volumes to justify the large upfront expense of custom ASIC design. Reprogrammable platforms also has the same flexibility of software running on a processor-based system, but it is not limited by the number of processing cores available. Unlike processors, FPGAs are truly parallel in nature, so different processing operations do not have to compete for the same resources. Each independent processing task is assigned to a dedicated section of the chip, and can execute autonomously without any influence from other logic blocks. As a result, the performance of one part of the application is not affected when you add more processing.

Taking advantage of hardware parallelism, FPGAs exceed the computing power of digital signal processors (DSPs) by breaking the paradigm of sequential execution and accomplishing more per clock cycle. Controlling inputs and outputs (I/O) at the hardware level provides faster response times and specialized functionality to closely match application requirements. FPGA technology offers flexibility and rapid prototyping capabilities in the face of increased time-to-market concerns. The growing availability of high-level software tools decreases the learning curve with layers of abstraction and often offers valuable IP cores (pre-built functions) for advanced control and signal processing. Because system requirements often change over time, the cost of

making incremental changes to FPGA designs is negligible when compared to the large expense of refactoring an ASIC.

2.4.1 Hardware Description Languages

For a long time, programming languages such as FORTRAN, Pascal, and C were being used to describe computer programs that were sequential in nature. Similarly, in the digital design field, designers felt the need for a standard language to describe digital circuits. Thus, Hardware Description Languages (HDLs) came into existence [Pal03]. HDLs allowed the designers to model the concurrency of processes found in hardware elements. Both Verilog and VHDL simulators to simulate large digital circuits quickly gained acceptance from designers.

Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs. Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections. HDLs are used for simulation of system boards, interconnect buses, FPGAs, and Programmable Array Logic (PALs). A common approach is to design each IC chip, using an HDL, and then verify system functionality via simulation

2.5 Multicore Processor Technology

With the increasing demand for computing power, greater levels of security and higher performances, more control applications will be used which can require more complex design and implementation techniques. The need of higher computing power is being covered by the adoption of multi-core architectures. The multicore processor comprises of two or more cores or computational/processing units that operate in parallel to read and execute instructions. The key factor about multicore processor is that it gives the same performance of a single faster processor at lower power dissipation and at a lower clock frequency by handling more tasks or instructions in parallel. The performance of a processor is a function of three major factors:

- Instructions per cycle, can be improved by increasing instructions level parallelism and thread level parallelism.
- Clock cycles per instruction, can be increased by the techniques of pipelining.
- Clock frequency, if increased, the power dissipation increases, in turn cause overheating.

Therefore, at a lower clock frequency, the multicore processor will process more data than the single core processor. In addition to this, multicore processors deliver high performance and handle complex tasks at a comparatively lower power as compared with a single core.

The cores normally are independent of one another, except for resources such as main RAM and some peripherals I/O. There is also a cache level (typically L2) that is shared between the cores, however the most important caches (L1) are particular to each core. In these cache level a coherence problem can occur: caches on different cores may contain different values from the same address in the main memory. To resolve this kind of situation, it could use software-assisted solutions responsible for maintaining consistency across the cores of each core, or hardware solutions for example the Snoop Control Unit (SCU) in ARM processors.

A multicore architecture where every core is just an image of the other is called homogeneous multicore. A heterogeneous multicore is a set of cores which may differ in area, perfomance, power dissipated etc. There are a number of challenges involved in designing a high-perfomance multicore system: parallel processing and shared resources are some of the difficulties on migration of single-core applications. The problems are mitigated by using syncronisation and communications services.

There are different types of multicore configuration. The most used approaches encompass the Symmetric Multiprocessing (SMP) configuration and Asymetric Multiprocessing (AMP) configuration.

2.5.1 Symetric Multiprocessing

The symmetric multiprocessing (SMP) configuration is only seen on homogeneous platforms and there is only one operating system that commands the platform cores belonging to the SMP configuration.

Any application, process or task, which does not have affinity to one of the cores, can be executed in any core, and is the scheduler's job migrate those same tasks to different cores. Moreover, the idea is to achieve an favourable workload across all cores, through the migration of different tasks. However, this migration should not be too frequent since the change of tasks between cores may affect cache performance.

This multicore configuration presents the smallest memory footprint because the different cores run the same image of the operating system. All cores have the same view of memory and shared hardware. Usually one of the cores will be attributed the responsibility to boot the operating system as well as to ensure the startup of the remaining cores and will also eventually 'command' access to shared I/O peripherals.

In a single-core operating system it is necessary to use synchronism mechanisms for the correct operation of the tasks that share the same resources, since they execute in parallel, i.e. they can be schedule at intervals. In a multicore SMP operating system the situation is aggravated by the fact that the execution is truly parallel, and it is necessary to implement multicore synchronism mechanisms (eg spinlock) in order to avoid problems of competition of tasks. In this configuration where the cores have access to the same memory (i.e., the main memory will be

a shared between the colors) it is necessary to use its own hardware to maintain the coherence between the data in cache and memory main.

2.5.2 Asymetric Multiprocessing

The asymetric multiprocessing (AMP) configuration is defined by treating each core individually, i.e., each core executes independently of the other cores, running in each its individual version of an OS. The AMP configuration can be considered homogeneous if each one runs an individual copy of the same operating system or heterogeneous if each core has an OS different from the one executed in the other cores. This configuration is characterized by the increase memory footprint, because each core will have a copy of its own version of the operating system that it runs. However, they do not require data coherence mechanisms, since each core would have its own memory, cache and MMU. Each core can have a different view of available memory and shared hardware.

From the point of view of each core, it runs as if it were in a single-core configuration. In that case, due to the nature of the configuration (each core has its OS), no core knows the existence of others (except the synchronization and communication mechanisms). This particularity allows to have a execution environment identical to that of a single-core configuration, thus facilitating the migration of legacy applications.

2.6 Mixed Criticality Systems

An increasingly important trend in the design of real-time and embedded systems is the integration of components with different levels of criticality onto a common hardware platform. For example, embedded systems have a need for a deterministic response to real-time events and there is also an interest in using general-purpose OSs to handle operator interfaces, databases, and general-purpose computing tasks. Systems that can combine both criticality leves of processing on the same platform can save costs by eliminating redundant computing hardware. Mixed-criticality systems (MCS) integrate two or more distinct levels (for example safety critical, mission-critical and low-critical) in one system. The following sections give a glance in some hypervisors that deal with guests of different criticality levels.

2.6.1 SafeG

Safety Gate (SafeG) [SHT10], from TOPPERS Project, is a dual-OS open-source solution that takes advantage of ARM TrustZone extensions to concurrently execute an RTOS and a GPOS on the same hardware platform.

The VMM execute in secure monitor mode and handle the switching between the GPOS, executed in non-secure, and the RTOS, executed in secure. Spatial isolation is supported by configuring resources (memory and devices) used by the RTOS to be accessible only from secure. The remaining resources are configured to be accessible both from secure and non-secure. This configuration is performed at initialization time after SafeG is loaded. If the GPOS tries to access some resource configured as secure space, an exception occurs and SafeG is called. Time isolation of the RTOS is supported by carefully using the two types of interrupt. FIQ interrupts are forwarded to the RTOS, while IRQ interrupts are forwarded to the GPOS. In secure state, IRQs are disabled so that the GPOS cannot interrupt the execution of the RTOS. For that reason, the GPOS can only execute once the RTOS makes an explicit request, through a Secure Monitor Call (SMC), to SafeG. On the other hand, during the GPOS execution, FIQs are enabled so that the RTOS can recover the control of the processor. TrustZone is configured to prevent the non-secure side from disabling FIQ interrupts.

2.6.2 NOVA/MINI-NOVA

The NOVA microhypervisor [SK10][KHV14] , originally developed for the x86 desktop environment and relying on Intel's virtualization hardware. NOVA proposes a security-oriented solution that deallocates virtualization to user space enforcing the principle of the least privilege in microkernel style. A user-level environment contains the root partition manager, device drivers, and other special-purpose applications and services that have been written for or ported to the hypercall interface. This interface implements capability-based access to kernel objects.

The Mini-NOVA[XPN15] microkernel, which is designed to provide a virtualization approach for the ARM-FPGA platform. Mini-NOVA is built to host paravirtualized operating systems with lower complexity and has the ability to dispatch hardware tasks to virtual machines by supporting the dynamic partial reconfiguration technology.

The Mini-NOVA kernel is an abstract layer between the physical resources and software users. For each guest OS/application, a virtual machine is initiated, running in an isolated user domain. A virtual machine monitor is used to create the virtualized environment for VMs. Based on the microkernel features[Lie95], the VMM should provide VMs with four basic properties: CPU virtualization, memory management, communication, and scheduling. To minimize the TCB size of the privileged code. The Mini-Nova architeture is illustrated in Fig. 2.3

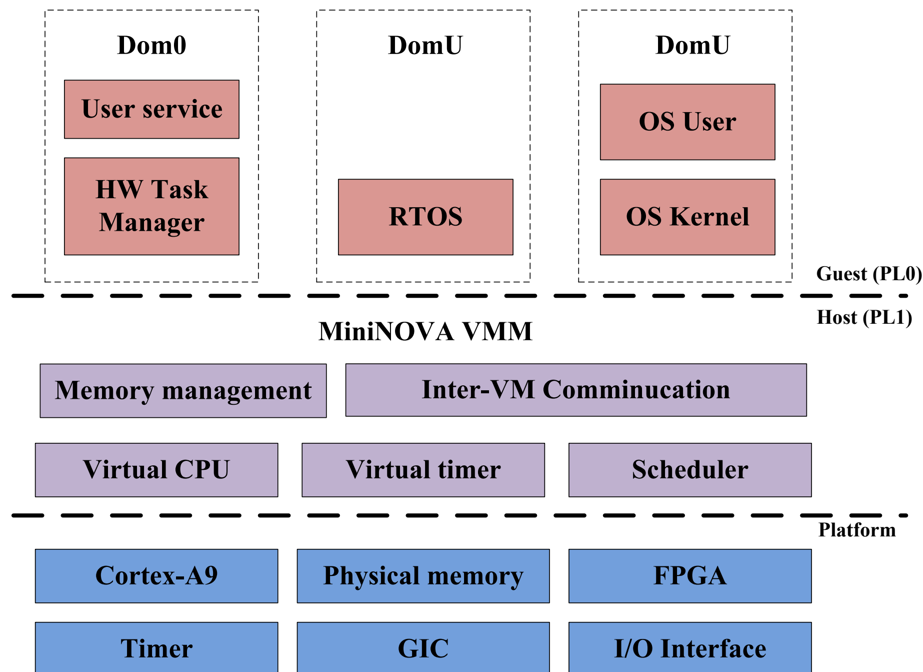


Figure 2.3: Mini-NOVA Architecture Overview

2.6.3 TZVisor Project

TZVisor Project[1,2], are a set of TrustZone-assisted hypervisors, developed and maintained by the Embedded System Research Group (ESRG), part of the ALGORITMI center of University of Minho. The hypervisors RTZVisor, µRTZvisor and LTZVisor belong to the TZVisor Project, all of them rely on on TrustZone hardware.

LTZVisor[PPG+17], is an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticallity systems. LTZVisor implements a dual-guest OS configuration: the secure world hosts the RTOS and the hypervisor, while the non-secure world is assigned to the GPOS. Currently, there are two versions: the single-core version and the multicore version that follows an AMP scheme.

RTZVisor[PTM16a] is a monolithic hypervisor which takes advantage of the ARM's TrustZone security hardware extensions to implement a minimal separation kernel, that allowing the execution of multiple real-time guest OSs. All hypervisor components, drivers and other critical parts of the virtualization infrastructure run in the most privileged processor mode, i.e., the monitor mode. Guest OSs are multiplexed on the non-secure world side. The hypervisor follows a simple and static implementation approach. All data structures and hardware resources are predefined and configured at design time, avoiding the use of language dynamic features. The strong spatial isolation is ensured through the TZASC, by dynamically changing the security state of the memory

[1] http://www.tzvisor.org/
[2] https://github.com/tzvisor/ltzvisor

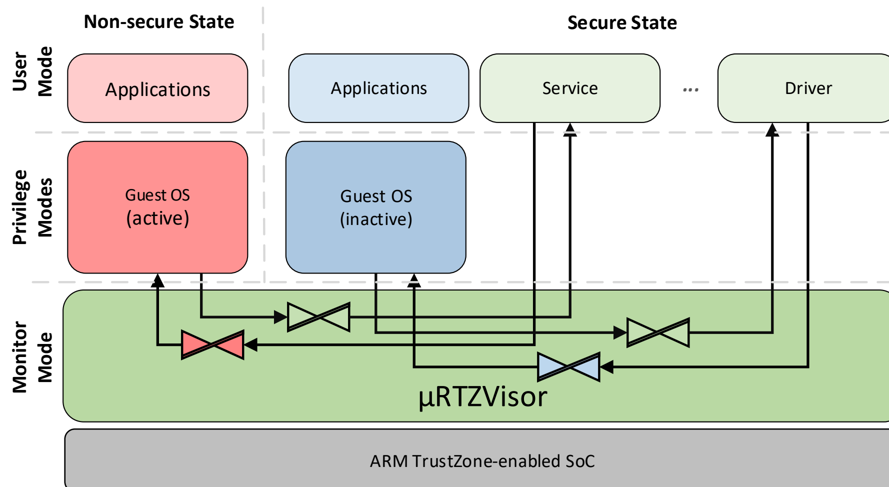


Figure 2.4: µRTZVisor Architecture Overview

segments. Only the guest partition currently running in the non-secure side has its own memory segment configured as non-secure, while the remaining memory is configured as secure.

The µRTZVisor[MAC+17], is a refactoring of RTZVisor, following an object-oriented approach and the MISRA coding guidelines. µRTZVisor follows a microkernel like architecture, maintaining its predecessor's ability of executing coarse-grained partitions in a quasi full-virtualized environment, while providing support for small user-mode tasks, intended to execute system services as extensions of the hypervisor. It provides much more functionality, real-time capabilities and configuration flexibility. This includes the addition of a capability-based access control and hypercall system, and a flexible IPC infrastructure tightly-coupled with the scheduling mechanism ensuring fast and efficient partition interaction. Figure 2.4 illustrate the µRTZVisor hypervisor architeture.

3. Platform and Tools

In this chapter, the base tools and platforms which lay the groundwork for the work carried out in this thesis are described. It starts by pointing out the selected platform and the features that are require for this project. Then a concise survey on the operating systems (OS), FreeRTOS and Linux, that provides the groundwork for this project will be made. In last subsection, we provide an overview of LTZVisor, an open-source lightweight TrustZone-assisted hypervisor mainly targeting the consolidation of mixed-criticality systems.

3.1 Zynq Platform

The Zynq-7000 family is based on the Xilinx All Programmable SoC (AP SoC) architecture, which integrates a feature-rich single or dual-core ARM Cortex-A9 based processing system (PS) and Xilinx programmable logic (PL) in a single device [CEES14]. A block diagram depicting the Zynq-7000 AP SoC architecture is presented in Figure 3.1.

All Zynq devices have the same basic architecture, and all of them contain, as the basis of the processing system, a ARM Cortex-A9 processor. This is a ”hard” processor, which means it exists as a silicon element on the device. The ARM Cortex-A9 CPU(s) is/are the heart of the PS, but the Zynq processing system encompasses also a set of associated computational units forming an application processing unit (APU), as well as further peripheral interfaces, cache, and memory interfaces. The APU is endowed with one or two ARM processing cores, each with associated computational units such as a NEON engine and floating-point unit (FPU), a memory management unit MMU, and a Level 1 data and instruction cache (both of which are 32KB). The APU also contains a Level 2 cache of 512KB for instructions and data, and there is a further 256KB of on-chip memory within the APU.

The second part of the Zynq architecture is the programmable logic. It is based on the Artix-7 and Kintex-7 field-programmable gate array (FPGA) fabric, depending on the specific device of Zynq family. The PL is predominantly composed of general purpose FPGA logic fabric, which is composed of slices (flip-flops, LUTs, and other logic) and configurable logic blocks, input/output blocks for interfacing, and other special resources such as block RAMs.

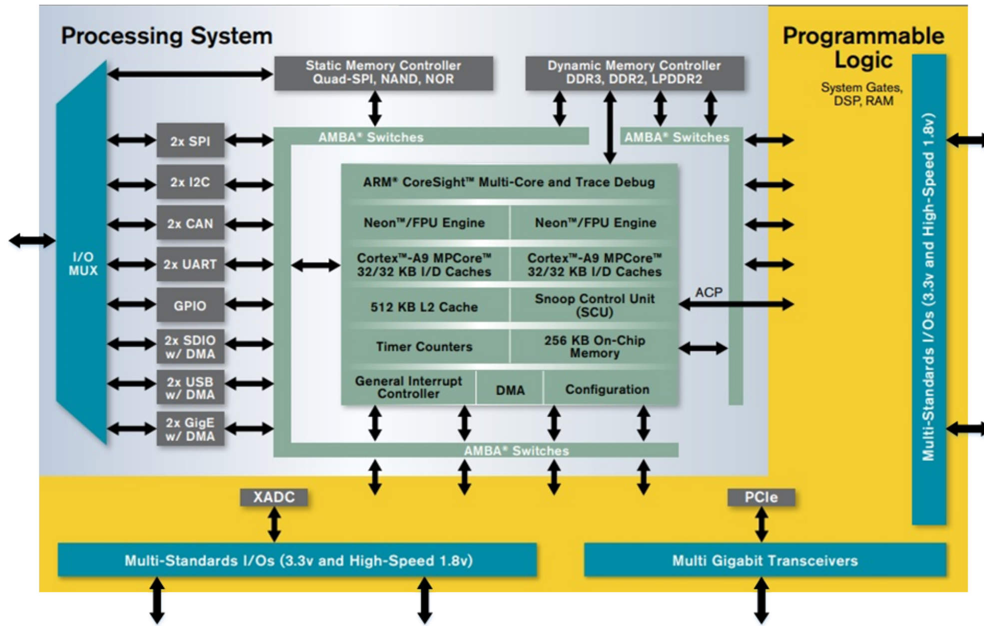


Figure 3.1: Zynq-7000 SoC

Interactions between the PS and the PL are supported through a set of nine Advanced eXtensible Interface (AXI) interfaces, each of which is composed of multiple channels. The current version is AXI4, which is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) 3.0 open standard.

There are a number of Zynq-based development boards. For example, the ZC702 Zynq device, ZedBoard and ZYBO (diminutive of Zynq Board). Zedboard was the selected platform. The ZedBoard is a low-cost, community-based board which features a XC7Z020 Zynq device. It is a joint venture between Xilinx, Avnet (the distributor), and Digilent (the board manufacturer). The Zynq device interfaces a 256 Mbit flash memory and 512MB DDR3 memory. There are diverse peripheral interfaces on the ZedBoard: general purpose I/O, HDMI and VGA video, Ethernet, USB-OTG (peripherals), USB-JTAG (programming), and USB-UART (communication), SD card slot, FMC interface, and Xilinx JTAG header.

3.1.1 Security

Zynq-7000 devices provide a wide range of security features which offer protection of the internal functionality of the system, ranging from dedicated hardware support for multiple encryption standards, secure system boot facilities, and software execution protection. The remainder of this Section briefly introduces the security features provided by Zynq devices.

Zynq-7000 devices have a number of embedded blocks which can support the creation of secure systems. The functionality of these security IPs includes anti-tamper, trust and information assurance, to protect the system from power-on and through runtime [GP09]. These blocks

include authentication, decryption engines, key storage and unique device identification possibilities. Some of the features of Zynq devices which relate to security are listed as follows:

- ARM TrustZone support (PS and PL);
- Secure configuration and boot (PS and PL);
- AES-256 encryption (BBRAM key and eFUSE key);
- HMAC bitstream authentication;
- First stage boot loader (FSBL) RSA-2048 authentication;
- JTAG disable/monitor.

The need for preventing unwanted access to the internal device data or memory does not end after the boot process has completed, and, obviously, there is a need to provide runtime security.

One feature of Zynq devices which can prevent such vulnerabilities is the Zynq specific implementation of ARM TrustZone technology [2.13, 2.1]. As previously explained, the Zynq-7000 SoC is divided into two domains: a processing system and a programmable logic domain. The Zynq-7000 AP SoC supports ARM TrustZone technology in both the PS and PL domains of the device. The PS provides a set of configuration registers related to TrustZone support for all hard custom blocks. These configuration registers can be dynamically programmed during software execution. In the PL, a security-checking feature is provided for each master interface slot in the AXI interconnect IP. A static secure or non-secure status can be assigned to an AXI interconnect master interface slot. All slave IP cores instantiated in the logic can also be individually assigned a secure or non-secure designation. For Xilinx slave 1glsIP cores, secure/non-secure configuration can be designated also at the AXI interconnect level.

3.2 Linux

Linux is an monolithic Unix-like operating system assembled under the model of free and open-source software development and distribution. It is a general-purpose operating system originally developed for personal computers based on the Intel x86 architecture, but which has been ported to a multitude of mainly MMU-enabled platforms, and has been used on ARM-based platforms for decades. Linux implements a monolithic kernel, which means it handles process management, networking, access to the peripherals, and file systems in kernel space. Device drivers are either integrated directly into the kernel, or added as modules that are loaded while the system is running. Its application goes from desktop and server computers to home appliances to smartphones, cars, low-end devices, and plenty other sectors.

Linux has a huge user base and support community, and the possibility of compiling the kernel is a major advantage. When adding new hardware, there are lots of resources necessary for adding drivers, and it is possible that in the open-source community someone has already developed such driver. Different Linux distributions have been ported to several Zynq-based platforms. For example, the Xilinx Linux distribution, the Linaro Linux distribution, and Digilent Linux distribution. All of them have support for ZC702, Zedboard and ZYBO platforms.

Xilinx Zynq Linux[1][2], from Xilinx Inc. [3], is based on open source Linux Kernel. Xilinx provides support for Zynq specific parts of the Linux kernel (drivers and BSP). Also supports Linux through the Embedded Linux forum. As with many open source projects, Xilinx also expects customers to also use the open source mailing lists for Linux in areas that are not specific to Xilinx Zynq. There are plenty third parties that produce many drivers that are compatible to the Xilinx processor or silicon configurations, like the HDMI transmitter IP from Analog Devices Inc.

3.2.1 HDMI Transmitter driver

When adding a display into an embedded platform — whether it be a low resolution, internal LCD panel or an external display output via HDMI — the system must be able to support it. The degree of support that is necessary depends on the specific application but it can generally be split into two levels: driver support and graphical interface support. In some heterogeneous platforms resources, such as the HDMI Transmitter driver, does not have a on-chip directly connected to the hard-wire processor. Instead, these resources need to connect firstly to the programmable logic (PL) and then the PL connects to the processor system in order to give software access to these peripherals.

The ADV7511 is a 225 MHz High-Definition Multimedia Interface (HDMI) transmitter. This reference design made by Analog Devices provides the video and audio interface between the FPGA and ADV7511 Hardware description language (HDL) reference on board. The Analog Devices provides device drivers for Linux, API and tutorials to implement the reference design. This device will be exploited to allow graphical interface to the non-secure guest, i.e. Linux. Assigning the same level of security of the guest who employ the HDMI peripheral.

3.3 FreeRTOS

FreeRTOS[4] is an open-source RTOS designed to be deployed on embedded systems with scarce resources. It is characterized by a very simple and small kernel core, written mostly in

[1] http://www.wiki.xilinx.com/Linux
[2] https://github.com/Xilinx/linux-xlnx
[3] https://xilinx.com
[4] https://www.freertos.org

C, presenting a software architecture divided into two main layers: the ”hardware independent” and the ”portable” layer [Bar10]. The former is responsible for performing processor independent functions and is maintained intact for all architectures, while the second implements some architecture-specific routines (e.g. context-switching).

3.3.1 Structure

The FreeRTOS source structure is very small [Bar09]: the core of the RTOS kernel is contained in only three C files. The tasks.c file provides a set of task management functionalities, including the scheduler component. FreeRTOS implements a preemptive priority-based scheduler policy, which privileges the execution of the highest priority tasks. For tasks with the same priority, the scheduler follows a round-robin model. In addition, the list.c file implements a list data structure for maintaining task queues (ready, waiting and running). These two files, altogether with the port specific code, implement the minimum core kernel high-level functionalities. The optional file queue.c implements a list of queues used for inter-task communication and synchronization. The timers.c file offers a set of functions to implement software timers used by application tasks. The operating system features also a special type of tasks, called ”co-routines”, that present high memory efficiency. These tasks are implemented within croutine.c file. The port.c file contains not only the hardware-specific code, but also the standard API of the OS. At last, the heap.c file provides the memory allocation and deallocation functionality, specific to the target architecture.

The FreeRTOS kernel can be tailored to the application being built through a configuration file called FreeRTOSConfig.h, where it is possible to adjust clock speed, heap size, mutual exclusion objects, API subsets, etc. Moreover, as an open-source RTOS with a small and simple kernel, it is possible to change the internals with a small engineering effort. These set of advantages and features justified the use of FreeRTOS as the target RTOS for this work.

3.3.2 Task Management

Tasks are implemented as C functions. The only thing special about them is their prototype, which must return void and take a void pointer parameter. Each task is a small program in its own right. An application can consist of many tasks. If the processor running the application contains a single core, then only one task can be executing at any given time. This implies that a task can exist, from an high-level perspective, in one of two states: Running and Not Running.

As there are several reasons for a task not to be running, the ”Not running” state can be expanded as shows Figure 3.2. A task can be preempted because of an higher priority task (scheduling is described in section 3.3.3), has been delayed or because it waiting for a event. When a task is ready to run but is waiting for the processor to be available, its state is said ”Ready”. This can happen when a task has is ready to run but there is a more priority task running at this

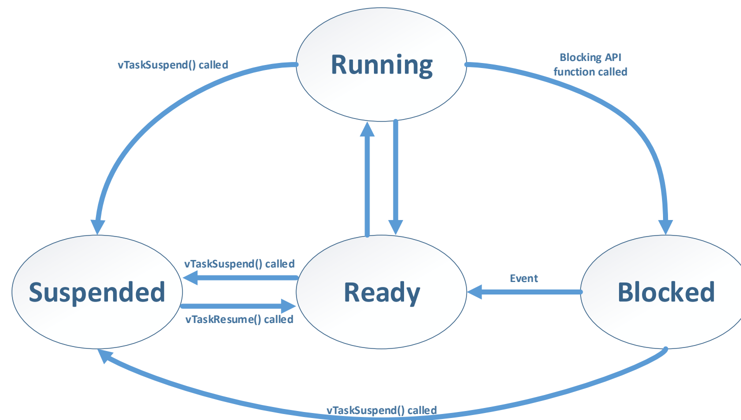


Figure 3.2: Task state machine

time. When a task is delayed or is waiting for another task (synchronisation through semaphores or mutexes) a task is said to be "Blocked". Finally, a call to vTaskSuspend() and vTaskResume() or xTaskResumeFromISR() makes the task going in and out the "Suspend" state.

The scheduler is the only entity that can promote a task for the Running state. Tasks can only be transitioned to the Running state from the Ready state. A task in Running state can exit on its own. However, the scheduler is the only service that can transit a task from Ready to Running state. If the task calls the vTaskSuspend() its state will become suspended and them the scheduler will select the task with higher priority ready for execution - Ready state. A task can also move from Ready to Blocked by waiting for an event. Events can be of two types: timing and synchronization events. When a task needs to be delayed for a certain number of ticks or for a specific amount time, for the vTaskDelay() and vTaskDelayUntil() services can be used, respectively. For synchronization events, FreeRTOS offers several features such as queues, semaphores, countdown lights, recursive semaphores, mutexes. When using synchronization events a timeout can be set; this means the task will exit the Blocked state as soon as its event occurs or when the assigned timeout expires.

The maximum number of priorities that can be assigned to tasks is defined by the constant configMAX_PRIORITIES. FreeRTOS does not impose any limitation on this value but the higher the number of priorities, the more RAM will be consumed by the OS, because for each priority a list of ready tasks is created. In certain platforms that present a "count leading zeros" type instruction in its Instruction Set Architecture (ISA) , the FreeRTOS provides a task selection mechanism (configUSE_PORT_OPTIMISED_TASK_SELECTION) that uses such type of instruction. In this case, the maximum number of priorities (configMAX_PRIORITIES) need to be bellow than thirty two priorities, whereas in other cases, configMAX_PRIORITIES can take any value. However, system designers need to take in mind, the lower the priority the less resources the system will need.

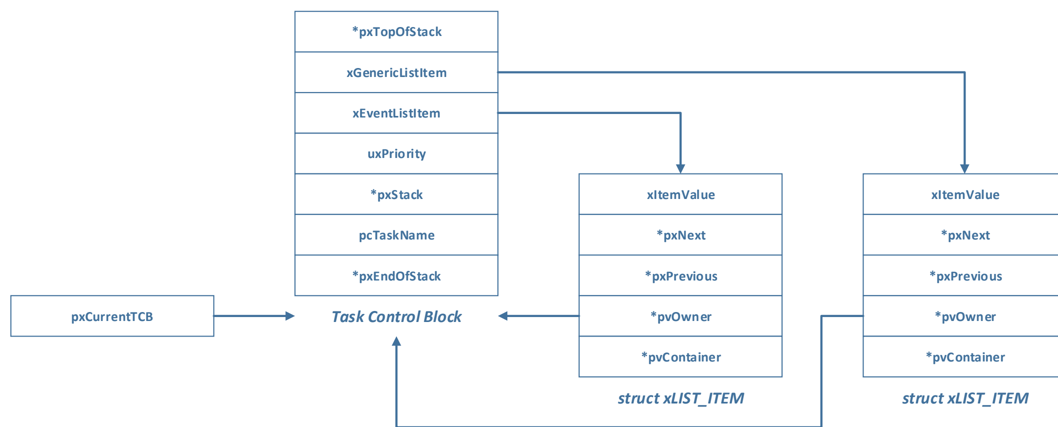


Figure 3.3: Task Control Block Structure

3.3.2.1 Task Lists

In order to manage the tasks, FreeRTOS assigns to each task a data structure, called a Task Control Block (TCB). Figure 3.3 shows this data structure. For each task is assigned a memory space, called stack. The TCB stores important information for the management of this stack, using three pointers for this purpose: pxTopOfStack, pxStack, and pxEndOfStack. The pxTopOfStack is a pointer to the last item placed on the task stack. The pxTopOfStack and pxTopOfStack point to the beginning and end of the stack, respectively. The TCB has also information regarding the task priority (uxPriority) and name (pcTaskName). The xGenericListItem structure is used to point to one of this lists, Ready, Suspended, and the xEventListItem points to the corresponded Block list.

Lists can be used as First In, First Out (FIFO) through the vListInsertEnd() function or as a list sorted by a value. The lists declared by default are: a list of ready tasks (pxReadyTasksLists); a list of tasks that are waiting for a temporary event (xDelayedTaskList); and list of tasks that were acknowledged by the scheduler while it was suspended (xPendingReadyList) and moved to the ready list when the scheduler resumes. There are other lists such as deleted tasks list and suspended task list which are used by the vTaskDelete() and vTaskSuspend() APIs functions

The tasks that are waiting for a temporary event are stored in the xDelayed_TaskList This list sort the tasks by their number of ticks left to unlock. FreeRTOS implements two identical lists for time events to troubleshoot problems associated with the count timer overflow. The pxDelayedTaskList pointer points to the list that is currently in use and the pxOverflowDelayedTaskList pointer points to the list which saves the tasks that will be unlocked after the overflow of the tick count. After an overflow occurs, FreeRTOS swap lists.

FreeRTOS implements some time facilities this by providing for each task a dedicated software-based counter, that is decremented at each system tick. In a complex system, where it is likely to exist several timers active, the management and activation of the task represent a source of

jitter.

3.3.3 Scheduler

A task transitioned from the Not Running state to the Running state is said to have been ”switched in” or ”swapped in”. Conversely, a task transitioned from the Running state to the Not Running state is said to have been ‘switched out’ or ”swapped out”. The FreeRTOS scheduler is the only entity that can switch a task in and out.

The FreeRTOS scheduler has two modes of operation: preemptive scheduling based on priorities and cooperative scheduling. The scheduling mode must be chosen by the system user through the configUSE_PREEMPTION parameter in the FreeRTOSConfig.h file. If this parameter preemptive scheduling will be used, otherwise, it will be used cooperative scheduling. The preemptive scheduling algorithm can be summarized in four points as:

- Each task has an associated priority.
- Each task exists in one of the states.
- At any time only one task is running (Runnning state).
- The scheduler chooses to always execute the task with higher priority with Ready state.

In summary, the preemptive algorithm selects the task ready-to-run with the higest priority choose the task ready-to-run that has the higgest priority. For tasks with the same priority, the scheduler follows a round-robin model, where at each tick the running task swap for another task with the same priority, providing the same time slice for every task with the same priority. The duration of on slice time is inversely proportional to the frequency of the system tick that is defined by the parameter configTICK_RATE_HZ.

3.3.4 Synchronization Mechanisms

The binary semaphores, semaphores, mutexes, queues and task notifications are used to synchronize tasks with other tasks or interrupt services routines (ISRs). These synchronization mechanisms can be used to unblock a task each time a particular interrupt, effectively synchronizing the task with the interrupt. This allows the majority of the event processing to be implemented within the synchronized task, with only a very fast and short portion remaining directly in the ISR.

Binary semaphores and mutexes are very similar but have a subtle differences: mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores a better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes a better choice for implementing simple mutual exclusion. The description of

how a mutex can be used as a mutual exclusion mechanism holds equally for binary semaphores. FreeRTOS declare binary semaphores and mutexes as a queue that can only hold one item. The queue can therefore only be empty or full (hence binary). Tasks and interrupts using the queue don't care what the queue holds - they only want to know if the queue is empty or full. This mechanism can be exploited to synchronise (for example) a task with an interrupt. Counting semaphores are also declare as queues with a length greater than one.

3.3.5 Software Timers

Software timers are used to schedule the execution of a function at a set time in the future, or periodically with a fixed frequency. The function executed by the software timer is called the software timer's callback function. This feature is implemented by, and are under the control of, the FreeRTOS kernel, and it is optional to include in the OS. They do not require hardware support, and are not related to hardware timers or hardware counters. In line with the FreeRTOS philosophy of using innovative design to ensure maximum efficiency, software timers do not use any processing time unless a software timer callback function is actually executing.

Software timer functionality is easy to implement, but difficult to implement efficiently. The FreeRTOS implementation does not execute timer callback functions from an interrupt context, does not consume any processing time unless a timer has actually expired, does not add any processing overhead to the tick interrupt, and does not walk any link list structures while interrupts are disabled. However, the task that does the tick handler add some indeterminism and so the software timers normally aren't so precise.

3.4 LTZVisor

LTZVisor [PPG+17], from TZVisor Project[5][6], is an open-source lightweight TrustZone-assisted hypervisor. This section overviews the LTZVisor architecture as well as the AMP implementation.

3.4.1 General Architecture

LTZVisor targets the consolidation of mixed-criticality systems, and implements a classical dual-guest OS configuration: the secure world hosts the RTOS and the hypervisor, while the non-secure world is assigned to the general-purpose operating system (GPOS). Figure 3.4 depicts the LTZVisor system architecture.

The hypervisor runs in the highest privileged processor mode, i.e. the monitor mode. It is responsible for enforcing the inter-partition isolation, by configuring the security state of memory,

[5] http://www.tzvisor.org/
[6] https://github.com/tzvisor/ltzvisor

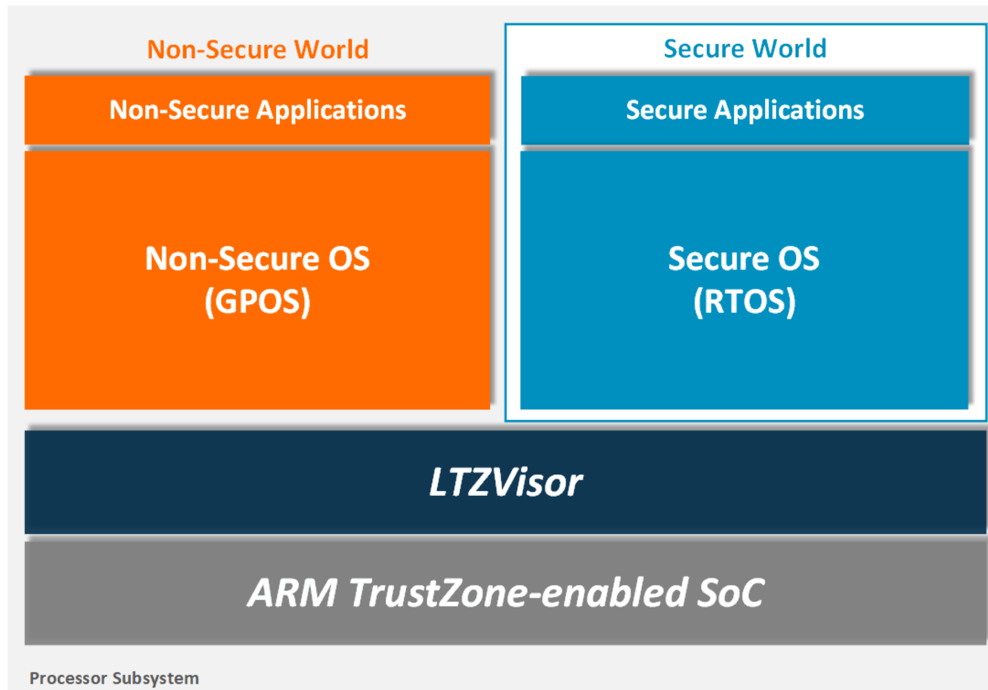


Figure 3.4: LTZVisor system architecture.

devices, and interrupts. The RTOS kernel runs in the secure supervisor mode. Therefore, it has full view over the non-secure privileged software, which means it is part of the trusted computing base (TCB) and necessarily must have a small footprint. The GPOS kernel runs in non-secure supervisor partition. The secure partition is completely isolated from the non-secure partition, and any attempt from the non-secure guest OS to access any of the secure world resources will immediately trigger an exception to be handled by the hypervisor.

LTZVisor implements an asymmetric or idle scheduler. This scheduling policy guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. As a result, LTZVisor overcomes the hierarchical scheduling problem that most real-time environment virtualization solutions enjoy [ZMH15]. Typically, a hypervisor schedules virtual CPUs while a guest RTOS running over the virtual CPU schedules its own tasks.

Spatial isolation between guest OSs is enforced by the TrustZone-aware hardware. The hypervisor uses the TZASC to configure the security state of the memory blocks of the respective partition.

Regarding the interrupt subsystem, LTZVisor follows the suggested ARM model, assigning fast interrupt requests (FIQs) to the secure partition and interrupt requests (IRQs) to the non-secure partition. While executing in the non-secure world, FIQs are set to be handled by the hypervisor. When an FIQ occurs while non-secure guest is executing, the hypervisor will then trigger a context-switch to the secure partition, resulting in minimal interrupt latencies for the real-time OS. This guarantees the secure guest

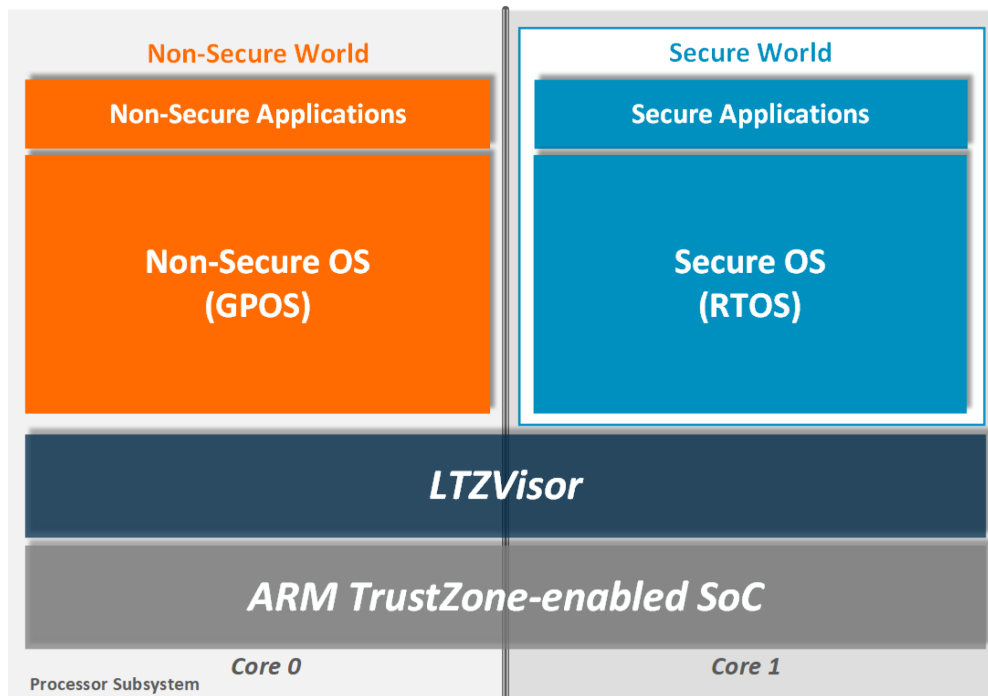


Figure 3.5: LTZVisor AMP architecture.

3.4.2 LTZVisor-AMP: Multicore Extension

LTZVisor-AMP implements support for a supervised asymmetric multi-processing (AMP) configuration [POP+17]. In such schema one core runs in the secure world and hosts the secure software (LTZVisor and RTOS), while the other core runs in the non-secure world and is responsible for hosting the non-secure software (GPOS). Current implementation is limited to the one-to-one mapping between core, guests and worlds. Figure 3.5 depicts the LTZVisor-AMP architecture.

For easing development and to avoid modifications to the Linux kernel, the GPOS runs over the primary core - core 0. This means the real-time OS, running on the secure world side, is assigned to the secondary core - core 1.

The multicore extension solves the problem of starvation which occurs in single-core platforms (when the RTOS does not yield its control of the CPU), while presenting significant performance advantages when the RTOS has a heavy workload [POP+17]. Nevertheless, the fact typical real-time application have frequent idle time, which making the CPU unusable on those time slices is not availed non-secure guest OS performance increase.

3.5 Benchmarks

Benchmarking and performance analysis are a well-established method of comparing the performance of various subsystems across different processors and system architectures. Several

benchmark suites exist targeting different metrics, systems and domains. This section provides a quick look over the Benchmark suites used during the development of this thesis, explaining the main reasons behind their choice.

3.5.1 Thread-Metric

The Thread-Metric benchmark suite[7], from Express Logic, is a freely-available set of benchmarks that measures many aspects of RTOS performance. Criteria such as interrupt response, context-switching, message passing, thread scheduling, memory allocation, and synchronization are particularly important when evaluating an RTOS. Thread-Metric consists of the following benchmarks: cooperative context switching, preemptive context switching, Interrupt processing, interrupt processing with preemption, semaphore processing, message passing, and memory allocation and deallocation. Each benchmark outputs a counter value, representing the RTOS impact on the running application: the higher the value, the smaller the impact.

The number of benchmarks available for evaluating the RTOS overhead/performance is scarce. Thread-Metric has been widely used across academia and industry. It has the advantage of being freely available and made open-source by Express Logic. It is also easily adapted to other RTOSs, just by mapping the generic APIs into the RTOS-specific APIs. No special hardware is required, and the code was tested with various compilers.

3.5.2 LMBench

LMBench [MS96] is a widely known suite of simple and portable micro-benchmarks used for measuring the most important factors that affect system performance, such as bandwidth and latency. The timing harness is the heart of the system, because it manages the benchmarking process: starting the benchmarked activity, repeating the benchmarked activity as long as necessary to ensure accurate results, and finally managing statistics to report representative results. The suite is written in portable ANSI-C using POSIX interfaces and targeting UNIX systems.

The LMBench 3.0 suite includes more than forty micro-benchmarks within three different categories:

- bandwidth - file read, memory read/write/copy, memory map, and others;
- latency - memory latency, inter-process communication using Transmission;
- Miscellaneous - CPU clock speed, translation lookaside buffer (TLB) size, cache line size, arithmetic operations parallelism, memory parallelism, and others.

[7] http://rtos.com/PDFs/MeasuringRTOSPerformance.pdf

There are a signifcant number of available benchmarks for GPOSs, namely targeting different architectural components. LMBench provides a plethora of microbenchmarks, in the same suite, ranging from computing intensive (e.g., arithmetic operations) to memory, communication and I/O intensive tests. Its availability as an open-source tool, as well as its widespread in Unix platforms, make it an attractive option compared to other benchmark suites. The benchmarks are all in C, and so, fairly portable. The source is small and easy to extend.

4. Trust SecSi Code: A TrustZone-assisted Secure Silicon Co-design Framework

This chapter intends to provide details regarding the implementation of the TrustZone-assisted Secure Silicon Co-design framework (Trust SecSi CoDe). The framework development was split into three parts: (i) development of a hardware-software RTOS that is FreeRTOS application compatible; (ii) modified Zynq Linux for supporting multicore and graphical interface; (iii) modified version of the hypervisor LTZVisor to integrate the software-hardware RTOS and the multicore GPOS guest.

4.1 Overview

LTZVisor provides a virtualization solution based on the two virtual execution environments provided by the TrustZone hardware. Notwithstanding, it is believed that the exploration of reconfigurable computing can bring several benefits and present a promising approach.

The main design principle of the co-designed framework continues to be the same as LTZVisor's: rely on TrustZone hardware support as much as possible. In addition, it exploits field-programmable gate array (FPGA) technology to offload secure software services (RTOS) to hardware, and improve the GPOS performance and utility without compromising any real-time deadline. Figure 4.1 depicts the proposed hardware-software co-designed architecture. In this figure, three main software components can be identified: the hypervisor; the real-time guest OS; the general-purpose guest OS. Figure 4.1 also presents two main hardware subsystems: RTOS related services and the HDMI Transmitter Interface.

LTZVisor runs in the highest privileged processor mode, i.e., in monitor mode. The hypervisor, follows an AMP schema, which mean it is split into two parts. The core of the hypervisor is the master of the system and is responsible for the main tasks, configuring memory, interrupts and devices assigned to each guest OS, as well as guaranteeing run-time support for inter-partition communication (IPC). LTZVisor kept the core of the system running in the secondary core - core 1 (to avoid severe modifications on the Linux guest [POP+17]). This limitation of the current

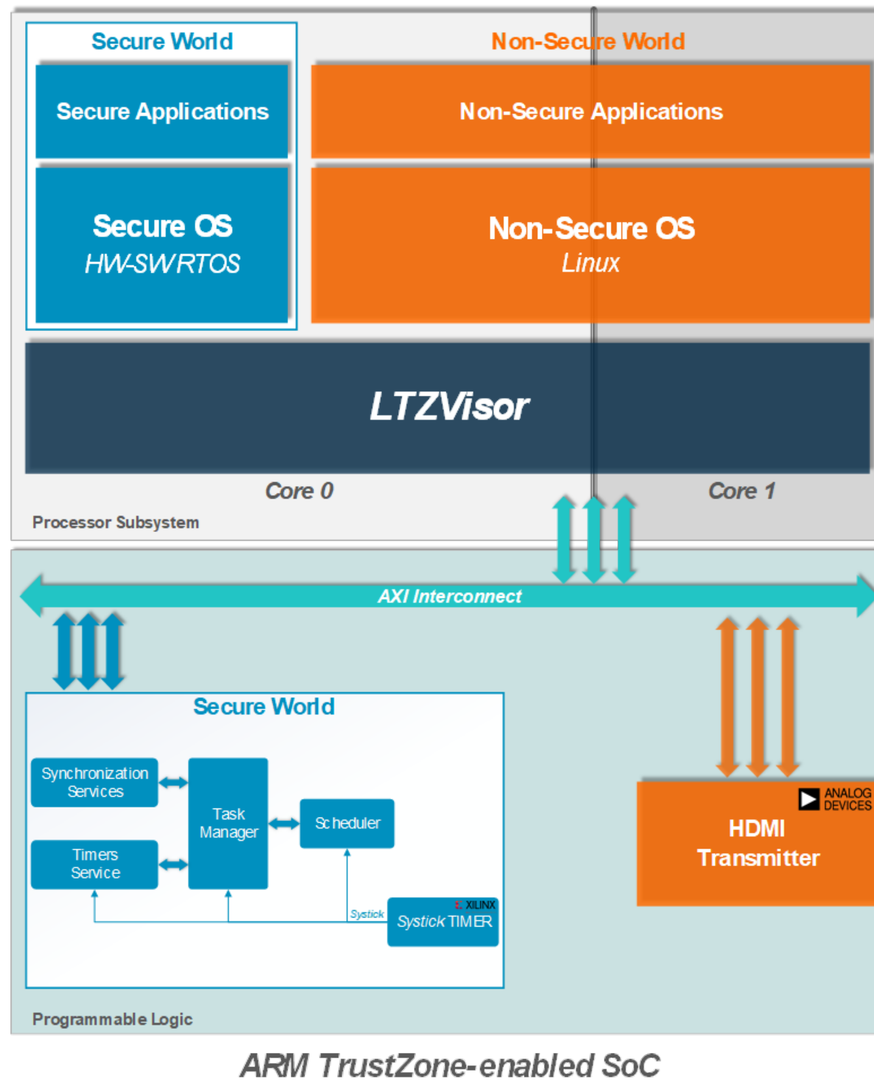


Figure 4.1: Trust SecSi CoDe system architecture.

approach is addressed in this framework by providing the flexibility of executing the hypervisor's main layer in either core.

The real-time guest OS kernel runs in the supervisor mode of the secure world and was assigned to the main core - core 0. This VM must have a small footprint, because, when the processor state is secure, it has full view over the non-secure world side. Nonetheless, offloading RTOS services to hardware also bring benefits in terms of security, due to the contention of the software TCB.

The general-purpose guest OS kernel runs in the supervisor mode of the non-secure world side. LTZVisor-AMP, which obeys a rigid AMP schema where each guest executed is in a dedicated core, this framework brings a higher degree of flexibility and scalability for the implemented AMP configuration. In this sense, the rich or multimedia OS has permission to explore secondary cores during the idle periods of the RTOS. This approach does not induce non-secure VM starvation, as

there will be always one core dedicated for this VM, like in LTZVisor-AMP. Nevertheless, this will not affect the secure VM's execution since TrustZone technology guarantees, by design, that the non-secure world is always less privileged than the secure one, despite the CPU execution mode.

In behalf of ensuring the hard real-time requirement of critical software, some RTOS services were oflloaded to the reconfigurable logic fabric. The scheduler, mutexes and the kernel software timers services represent the major sources of jitter and overhead [KGJ03] in RTOSes and therefore, are major candidates for hardware offloading. The RTOS hardware subsystem is agnostic from the RTOS software implementation that might be running in the secure world.

The HDMI Transmitter IP, which is supported by Analog Devices, provides graphical interface for the GPOS and thereforethe FPGA hardware is configured with the security state as as the associated operating system. The supplier also provides the necessary software driver compatible with Linux OS.

4.2 Secure VM (Hardware-Software RTOS)

Since in LTZVisor's architecture the RTOS is part of the TCB of the system, both hypervisor and RTOS layers act synergically. In order to reduce the software TCB and meet the strict timing requirements and constraints imposed by real-time systems, some RTOS services were deployed in the reconfigurable hardware.

The coexistence of a hybrid, software and hardware model in an operating system environment, raises concerns namely regarding an unified programming model, portability, legacy software support, suitable interface and synchronization mechanisms, communication overhead and resource optimization [ZQCP05, WCW+09, NA07, POP+14], eventually exacerbated in resource constrained embedded contexts.

Determinism and latency are the critical metrics of these systems. Migrating tasks and services from software to hardware help to mitigate/aleviate these issues, leading to solutions able to cope with these increasingly strict requirements. Migration of RTOS services, such as scheduling, time management and task management, to dedicated hardware modules, provides increased system performance and allows the RTOS to improve predictability and determinism [MNK14]. To distinguish between the original FreeRTOS and the implemented co-design, the hybrid system will be named hardware-software RTOS.

4.2.1 System Structure

For the purpose of this project, the task management service, the scheduler, synchronization service, and the software timers feature, were the implemented hardware services. Figure 4.2 illustrates the architecture of the hardware-software RTOS.

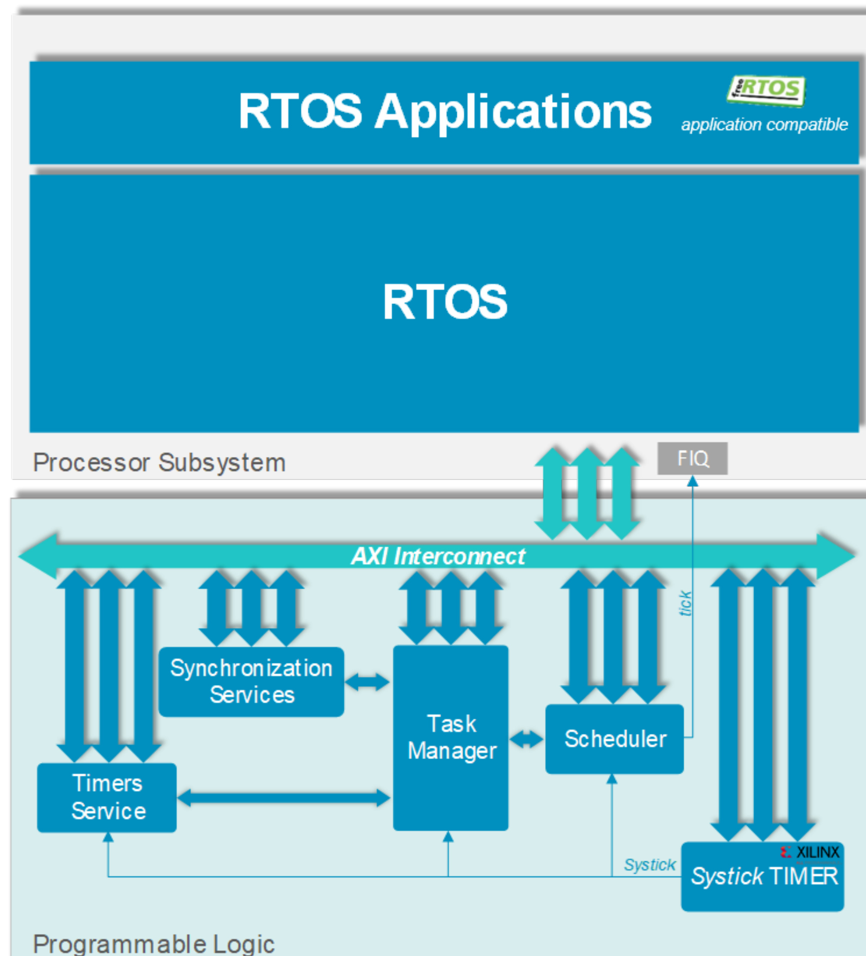


Figure 4.2: Hardware-Software Co-design RTOS architecture

In order to maintain the same FreeRTOS API, the source code was extended with conditional compilation entries. To enable the co-designed system, a new configuration parameter was created, called configRTOS_CODESIGN. When enabled, the classic services available in software are replaced by its respective hardware implementation. Listing 4.1 is an example of modification in the tick ISR, FreeRTOS_Tick_Handler() located in port.c.

Listing 4.1: FreeRTOS_Tick_Handler() modification

```
void FreeRTOS_Tick_Handler( void )
{
    /* If the Hardware-Software Co-Design
    configuration is enable. */
    #if ( configRTOS_CODESIGN == 1 )
        ulPortYieldRequired = pdTRUE;
    #else /* configRTOS_CODESIGN */
        /* Increment the RTOS tick. */
        if( xTaskIncrementTick() != pdFALSE )
```

```
            {
                ulPortYieldRequired = pdTRUE;
            }
        #endif /* configRTOS_CODESIGN */

        /* Ensure all interrupt priorities are active again. */
        portCLEAR_INTERRUPT_MASK();
        configCLEAR_TICK_INTERRUPT();
}
```

The functions that interface with the hardware modules are implemented as INLINE in the header files started with the nomenclature "secsirtos_". They are defined as "inline functions"since they just implement read and write operations on the AXI4-LITE communication. The AXI4-LITE communication is the main peripheral bridge between programmable logic and processing system. In all hardware modules, the inputs and outputs are named with a "_in" and a "_out" at the end, respectively .

4.2.2 SysTick Timer

Due to scheduler and time services hardware deployment, the timer that generates the system tick also have to be implemented in the reconfigurable hardware for the correct use of the migrated services. Since the timers are a generic peripheral, and Xilinx provides its own timer IPs, we have used the AXI Timer IP.

The AXI Timer is organized as two identical timer modules. Each timer module has an associated load register that is used to hold either the initial value for the counter for event generation, or a capture value, depending on the mode of the timer. This timer replaces the timer used by FreeRTOS while presenting the same characteristics, where the configuration of the frequency used by the system tick must be done through the configuration macros that the operating system already makes available. By using such an approach the user does not need to implement a different configuration.

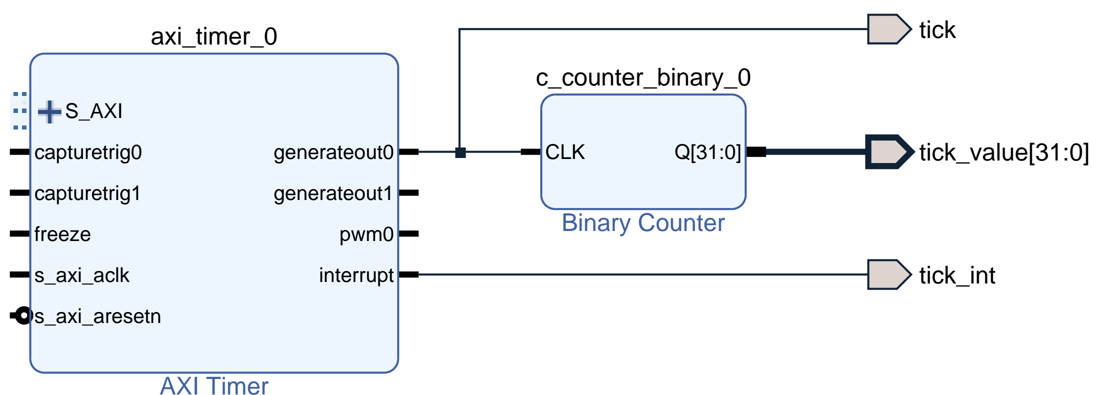


Figure 4.3: AXI Timer and the Counter IP

Figure 4.4: Task Manager IP.

At the interrupt output, a counter has been added so that the kernel hardware can keep track of time intervals. Figure 4.3 depicts the AXI Timer hardware module, the counter, as well as all the connections needed between them.

4.2.3 Task Management

The task management service controls and manages the state of each task, by sorting them in lists. It is worth mentioning that the scheduler is the only entity responsible of changing a task from a "ready" state to a "running" state. Software list manipulation is a good example of a source of unpredictability [GPG+15]. The time taken to select the next task to run in the ready queue list of FreeRTOS' priority-based scheduler, is very dependent on its position into the list.

In order to manage the tasks states, the hardware Task Manager IP (hTM IP) also sorts them in lists. However, due to the parallel nature of reconfigurable platforms, the unpredictability of list manipulation operations will be mitigated.

The hTM IP sorts the tasks ready-to-run according to their priority. There are two lists stored in PL: Task List and Priority list. For each task, Task List saves the TCB address, "*TCB", the priority number, "Priority", and two pointers, "*Previous" and "Next", at the address of the provided task identifier ("ID TASK"). The two pointers are used to create a doubly linked list between all ready-to-run tasks with the same priority. For each priority, Priority List saves the ready-to-run task identifier of the first, "*TASK STRT", and last, "*TASK END" , task saved with the priority in question, as well as the number of tasks associated, "NUMBER ELEMENTS".

For the tasks in temporal block state, the hTM IP sorts them in one linked list (Delay List) instead of having two lists, one for the current tick counter and another for the after overflow of the tick, as the original version of FreeRTOS.

The Delay List sort the delayed tasks by order of expiration time. Starting with the task with the lowest expiration time up to the task with the highest expiration time. To facilitate insertion and deletion this list is doubly linked. So in each address is saved the pointers "*Prev", "*Next", and the expiration time, "DELAY VALUE". The address of each Delay List element correspond to the task identifier in question.

To establish an easy task identification between the hardware module and the software service, the implemented approach uses the trace facilities provided by the FreeRTOS, to store the task hardware identifier. Toward an efficient task management, the hardware module has to be acknowledge when a task creation/deletion (vTaskCreate, vTaskDelete), or a change of task state (vTaskDelay, vTaskDekayUntil, vTaskSuspend, vTaskResume and vTaskAbortDelay) occurs. This module also provides access to the information of a certain task for the hardware scheduler. Figure 4.4 illustrates the Task Manager hardware module and its I/Os.

The following sections explain how each type of task operation is implemented, as well as the necessary inputs and outputs for each operation.

4.2.3.1 Task Create/Delete

In an application designer perspective, when a task creation occurs, several key parameters are specified for a specific task (TCB). In a similar way, the hTM IP that implements the task creation retains the task identifier (uxTCBNumber), the correspond TCB address (pxNewTCB) and the assigned priority (uxPriority), while leaving the software layer to store all the remaining information. These parameters are provided when the prvAddNewTaskToReadyList() is invoked.

For deletion operations, the hardware module has to receive the task identifier (uxTCBNumber) of the desired task. This parameter is provided when the vTaskDelete is invoked.

The creation operation is mastered by the following path:

1. Receive a signal on createTask_in and, at the same time, the values of TCB address (addrTCB_in), priority (priority_in) and task ID (taskID_in);

2. Address the Priority List with the priority received and check if there are ready-to-run tasks with the same priority, i.e., if the parameter "NUMBER ELEMENTS" is not null:

    (a) In case the "NUMBER ELEMENTS" is null, (meaning that the new task is the first ready-to-run task with that certain priority), the hTM IP can save the received parameters of the new task on the Task List;

    (b) Else, address the last ready-to-run task on the doubly linked priority list (such information is stored in "*TASK END") and updates the parameter "*Next" so that the new task is inserted in the linked list.

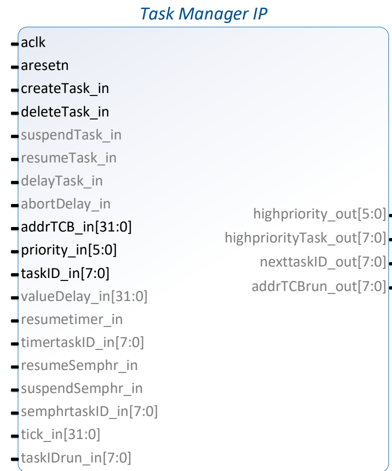


Figure 4.5: Create and Delete Task Interface.

3. Update the "*TASK END" with the inserted task ID and also increment the parameter "NUMBER ELEMENTS". In case of being the first task with a different priority from the previous inserted ones, the "*TASK STRT" is updated with the inserted task ID;

4. The Task List stores the TCB address and priority of the new task at the task ID provided. The parameters "*Prev" and "*Next" save the previous last task and the first inserted task with the same priority, respectively.

The deletion and suspension operations are quite similar. The main difference: the creation operation can successfully occur on a deleted task ID, overwriting the stored task information (see subsection 4.2.3.6). The delete operation proceeds executes as follows:

1. Receive a signal at ("deleteTask_in" and the identifier of the intended task (taskID_in) (Figure 4.5);

2. The hTM IP obtains the priority number, the next pointer and the previous pointer by addressing the Task List with the task ID received;

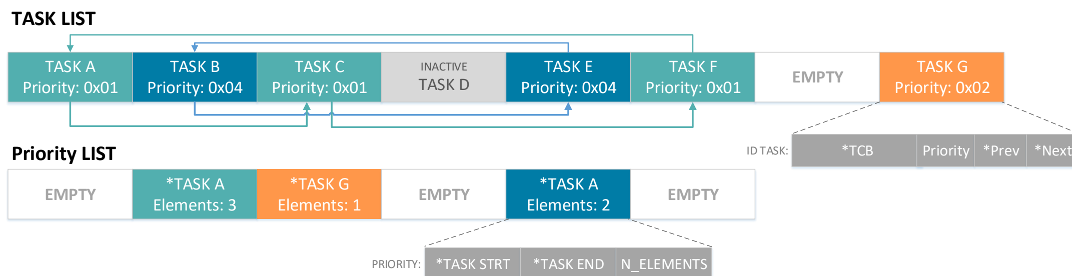


Figure 4.6: Task List and Priority List.

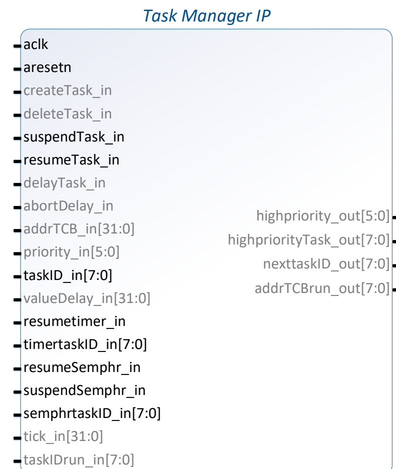


Figure 4.7: Resume and Suspend Task Interface.

3. Address the Priority List with the priority obtained and check if there are ready-to-run tasks with the same priority, i.e., if the parameter "NUMBER ELEMENTS" is not null:
   (a) In case the "NUMBER ELEMENTS" is null, (meaning that the intended task is the first ready-to-run task with that certain priority), the hTM IP can just reset "NUMBER ELEMENTS";
   (b) Else, remove the intended task from the associated doubly linked priority list.
4. Update the "*TASK END" in case the intended task was the last one in the doubly linked priority list. Also the same for the "*TASK STRT" in case the intended task was the first one in the doubly linked priority list.

Figure 4.6 illustrate an example of how the lists will look if there are three tasks, "Task A", "Task C" and "Task F" with same priority, "Task B" and "Task E" with the same priority but different of the other group, "Task G" with a unique priority number and "Task D" currently suspended or blocked.

4.2.3.2 Task Resume/Suspend

The "resume" and the "create" operations have similar behaviour. However, the "resume" operation only needs the task ID because all the remain information is already stored in the Task List (figure 4.7). Consequently, the "resume" has an extra step before checking the Priority list parameter "NUMBER ELEMENTS": get the priority number of the intended task by addressing the Task List.

The "suspend" operation and the deletion operation, as was previously said, are quite similar. However, instead of allowing overwriting operations on this particular task ID, a task in suspend state cannot be overwrited.

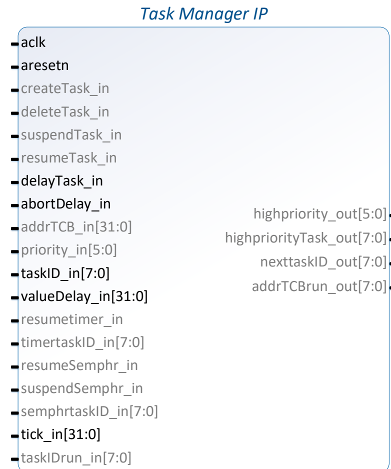


Figure 4.8: Delay and Abort Delay Task Interface.

A ”resume” operation can only execute if the task was previously in suspend state. The same goes for ”suspend” operations, these operations can only be executed if the task was previously in ready-to-run state (see subsection 4.2.3.6).

4.2.3.3 Task Delay/Abort

FreeRTOS API presents a set of functions to delay tasks for a specific amount of number of ticks. The actual time that the task remains blocked depends on the tick rate. vTaskDelay() and vTaskDelayUntil() are two functions from the FreeRTOS API responsible to delay tasks. The function vTaskDelayUntil() differs from vTaskDelay() in one important aspect: vTaskDelay() specifies a time at which the task wishes to unblock in relation to the time at which vTaskDelay() is called, whereas vTaskDelayUntil() specifies an absolute time (rather than a relative time) at which the task wishes to unblock.

In the implemented hardware-software RTOS, the calculations needed to determinate the absolute time at which the calling task should unblock (vTaskDelayUntil() function) remains as part of the software. This way, the hTM IP does not need to distinguish between a ”delay” and a ”delayuntil” operation. Figure 4.8 highlights the necessary inputs.

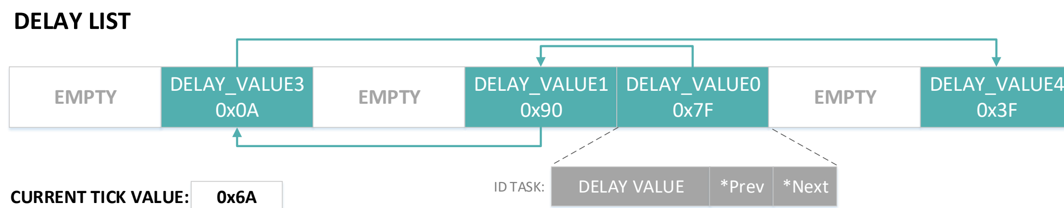


Figure 4.9: Temporal Blocked List.

In order to sort the temporal blocked tasks by their expiration time, the hTM IP presents the following execution path:

1. Receive signal on delayTask_in, the identifier of the intended task (taskID_in), and the expiration tick (valueDelay_in);

2. Remove the intended task from the ready-to-run list;

3. Verify if the Delay list is not empty. In case its not, there is two paths: survey until the it find a task with a higher expiration time than the intended task or survey until the end of the list;

4. Once the survey stops for either of the situations, the hTM IP inserts the intended task information on the interrupted position.

To abort a temporal block of a task, the bit abortTask_in needs to be set as well as the task identifier need to be on taskID_in. If the task was previously in the delay state, the delay will be aborted (removing it from the Delay List) and the task becomes ready again (inserting the intended task back to the ready-to-run linked list).

The hTM IP receives a ”delay” operation assuming that the delay time is not already expired.

The hTM IP only has one list for temporal blocked tasks, instead of two as the original version of FreeRTOS. This approach does not compromise the maximum delay time that the tasks can be submitted nor the number of tasks that can be in delay state at the same time. Figure 4.9 illustrates a representation of the Delay List with four tasks in delay state. The expiration time of the tasks with the delay ”DELAY_VALUE3” and ”DELAY_VALUE4” will happen after an overflow of the system tick.

4.2.3.4 Priority Selector

A classic RTOS implements a preemptive priority-base scheduler policy, where the scheduler selects the task which has the highest priority. This sub-module is responsible for surveying all ready-to-run task priorities and selecting the highest one.

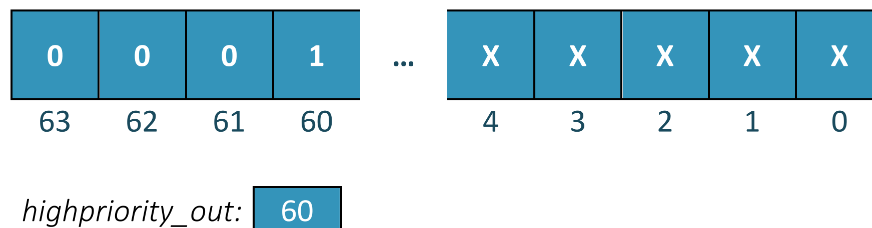


Figure 4.10: Bit Array of Priorities

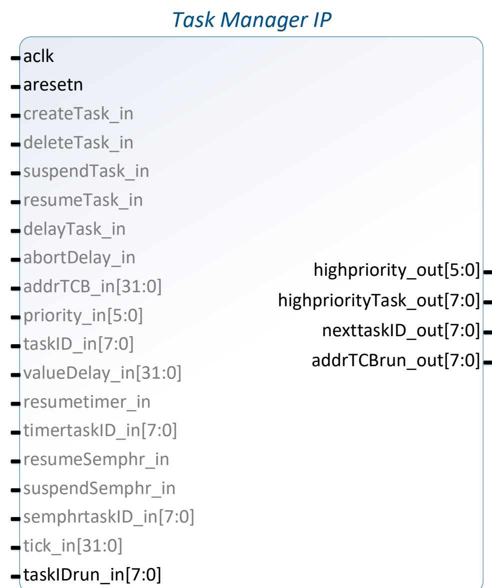


Figure 4.11: Scheduler Interface.

The selector organizes all the ready-to-run task priorities on an array of sixty-four bits, each bit corresponding to a priority number. When a ”create” or ”resume” operation is triggered, the respective priority bit is set; when all the tasks with the same priority are suspended/blocked, the bit is cleared. The output highpriority_out always contain the position of the highest priority bit set, by doing this, so it is always possible to know which is the highest priority level. Figure 4.10 illustrates an example of the bit array were the highest priority ready-to-run task has a priority number of 60.

4.2.3.5 Scheduler Interface

The scheduler is responsible for controlling which task goes to the running state. To allow this, the hTM IP provides the value of the highest priority (highpriority_out), the identifier of the first task with the high priority (highpriorityTask_out) and the identifier of the next task with the same priority (nexttaskID_out). In addition, provides the TCB address (addrTCBrun_out) of the task ID received on the input taskIDrun_in so the scheduler can perform the context switch if necessary. Figure 4.11 highlights these I/Os.

4.2.3.6 State Control

The hTM IP always needs to be prepared to receive an incoming operation operation. In addition, these operation can come from the application designer, from the scheduler, from the syncronization services, from the software timer module or even from internal operation (for example when a ”delay” operation occurs, the hTM IP also needs to remove the intended task

from the ready-to-run list). Before handling any of these commands, this module first verifies if the operation can be executed, them stores each valid operation in a circular buffer. If the necessary operation is possible, executes the saved operation.

In order to be a valid command, it is necessary to check the following rules:

- If the task is in resume state, all commands are valid, except "abort delays" or "create" commands;
- If the task is in suspend state, the "resume" command is the only valid one;
- An "abort delay" can only happen if the task is in delay state.

4.2.4 Scheduler

The scheduler has two modes of operation: preemptive or cooperative. If the preemptive mode is chosen, the scheduler preempts the ready-to-run task who has the highest priority, and when a prioritized tie occurs, the scheduler resolves it using the round robin strategy with time slice. This means that tasks with the same priority share CPU access when running alternately.

The hardware Scheduler IP (hS IP) controls which task starts to execute and which task gets out of the running state. The scheduler algorithm becomes less complex, as a result of the lists structure design. The hS IP was simplified in a way that it has only two assignments:

- Update the task in running state when the highest priority level is updated;
- When a systick is set and the highest priority did not change, the scheduler preempts the next task with the same priority than the previous running task.

Figure 4.13 depicts the behaviour through a state machine diagram. The state machine has three states: IDLE, TICK and PRIORITY. The hS IP starts at IDLE state. If the tick_in is set, the

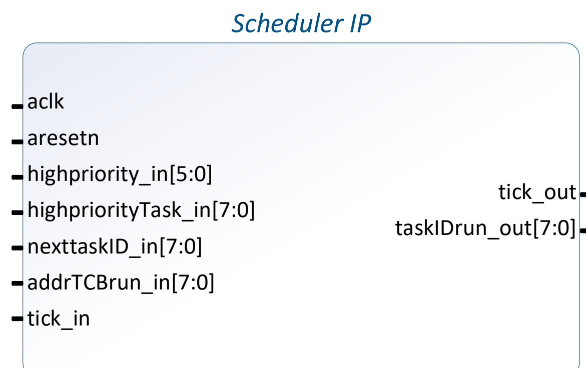


Figure 4.12: Scheduler IP.

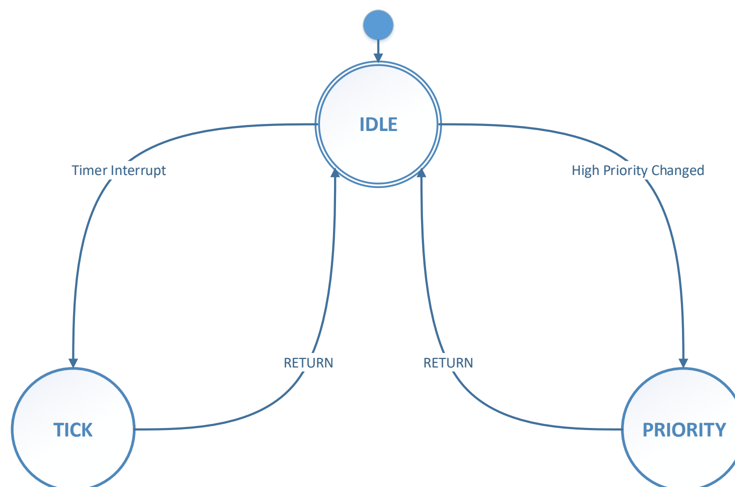


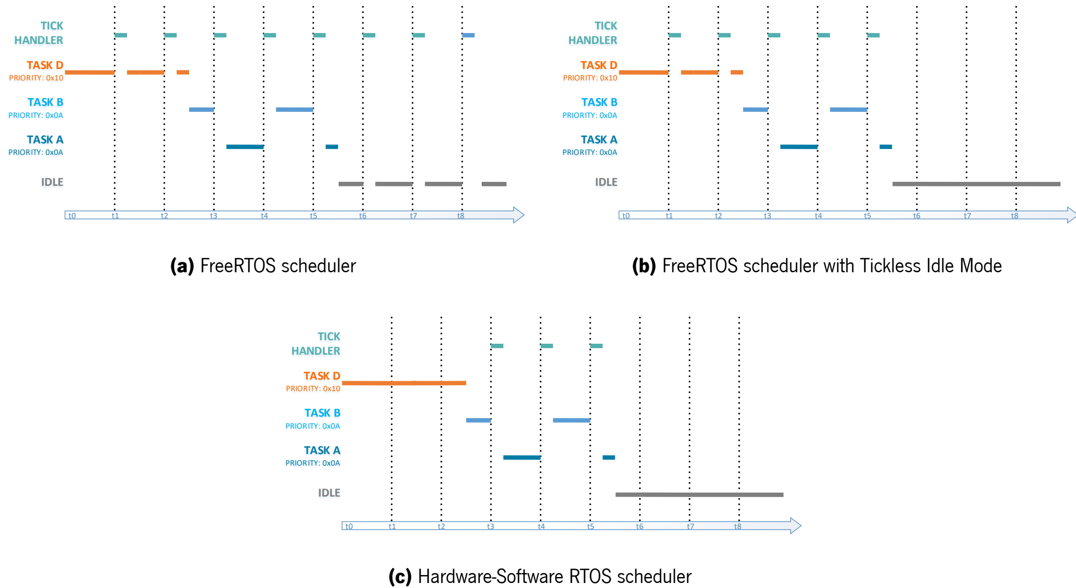
Figure 4.13: Scheduler State Machine.

module transits to the state TICK or if the highpriorityTask_in is updated, the module transits to the state PRIORITY. In PRIORITY state, the hS IP updates the register taskIDrun_out with the value on highpriorityTask_in and then returns to IDLE state. In TICK state, the scheudler updates the taskIDrun_out with the value nexttaskID_in and then returns to IDLE state.

The hS IP not only controls the TCB address of the task to run (addrTCBrun_out) , but also commands the tick interrupt (tick_out). The hS IP only sends an interrupt when a tick is triggered and a context switch is necessary. Therefore, the RTOS ensures that every time a tick handler is called, a context switch need to occur. By mitigating interrupts that would be useless, it improves the performance of each task that runs for more than one time slice.

FreeRTOS has a tickless feature, Tickless Idle Mode [Bar10] that suspends the system timer when the system is in idle time. This feature allows the microcontroller to remain in a deep power saving state (or in this virtualization context, used for non-secure world purposes), until either an interrupt occurs, or it is time for the RTOS kernel to transition a task into the ready-to-run state. The hS IP not only provides this feature, but also does not interrupt the processing of a high priority task, when there is not an higher process to attend.

Figure 4.14 illustrates a case scenario running on three different schedulers: (i) original FreeRTOS scheduler; (ii) FreeRTOS scheduler with Tickless Idle Mode enabled; (iii) implemented hardware-software RTOS scheduler. The case scenario consists of three ready-to-run tasks: "Task D" has the higher priority, and "Task A" and "Task B" have equal low priority. Each scheduler starts executing "Task D" for 2 system ticks. Then, executes "Task B" and "Task A" for for one time slice each. Ending with each scheduler in idle state.



(a) FreeRTOS scheduler

(b) FreeRTOS scheduler with Tickless Idle Mode

(c) Hardware-Software RTOS scheduler

Figure 4.14: Comparison between different scheduler behaviours during system ticks

4.2.5 Software Timers Service

A software timer (or just a "timer") allows a function to be executed at a set time in the future. The function executed by the timer is called the timer's callback function. The time between a timer being started, and its callback function being executed, is called the timer's period. The hardware "Software Timers" IP (hST IP) intends to replace the software functionality that is responsible for delaying the timer task and also migrating the indeterministic processing portion that checks if a timer has expired. Figure 4.15 illustrates the inputs and outputs of this module.

Before any timer creation, the hST IP must receive the task ID of the timers handler task (timerTaskID_in). The hST IP is responsible of resuming this task, leaving the PS in charge of suspend it.

The hST IP stores the information related to all created timer (the timer's period, callback function and if it is an auto-reloaded timer or not) in the "Timer Info list". Activated timers are saved in a doubly linked list called "Timer List". The Timer List sorts the activated timers by their expiration time. When a timer expires, the hST IP sends a "resume" command to the hTM IP (resumetimer_out) in order to place the timers handler task in ready-to-run state. Also, provides the timer's callback function to the PS (addrTimer_out). If the expired timer is an auto-reload timer, then it is again placed in the Timer list with a new expiration time. In addition to the timer's start and "create" commands, the hST IP allows commands such as timer's stop (stopTimer_in), change timer's period (changePeriod_in), and delete (deleteTimer_in).

The principle is the same as in Task Delay sub-module of hTM IP, sorting the active timers by ascending order of expiration time. Figure 4.16 depicts the behaviour of the list against expired

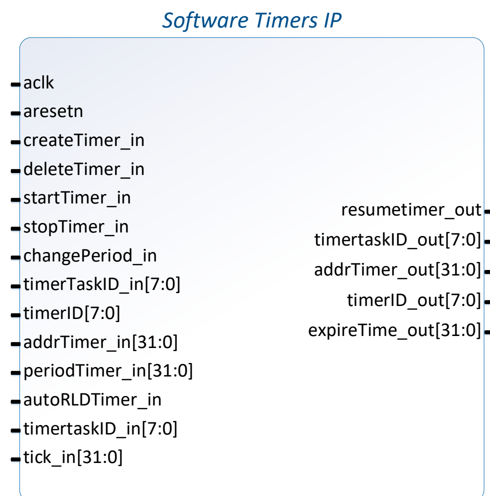


Figure 4.15: Software Timers IP.

values after overflow of the timer.

4.2.6 Synchronization Service

Mutexes, binary semaphores, and count semaphores, are quite similar in terms of implementation. The hardware Semaphore IP (hSmphr IP) implements a count semaphore that can be used as a mutex or a binary semaphore. Each semaphore has a number of maximum counts. If it is a binary semaphore, the semaphore only has one count available.

To create a semaphore, the createSemaphr_in needs to be set. Also, the hSmphr IP receives the maximum counting value that the semaphore can "give" (countmax_in) and the number of counts available in the initialization (countInit_in). Each semaphore has an ID that is provided also in the creation (semphoreID_in). The inputs take_in and release_in, correspond to the "take" and "release" operations, respectively. When one of these inputs is set, the hSmphr IP also receives

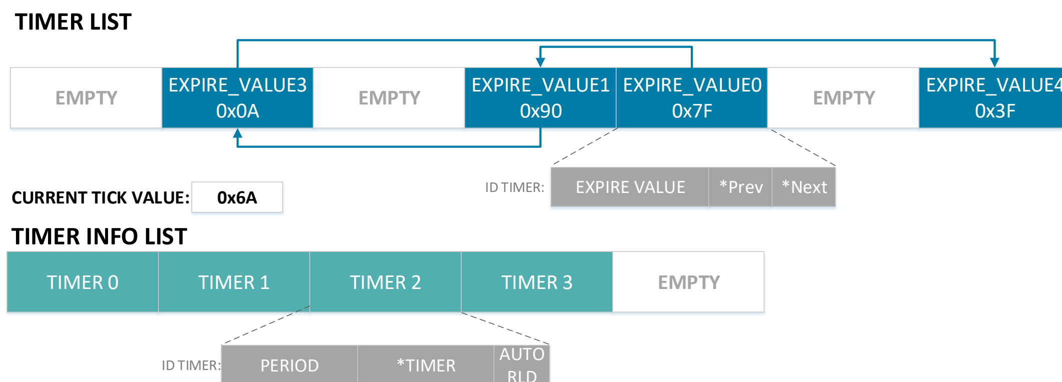


Figure 4.16: Software Timers List.

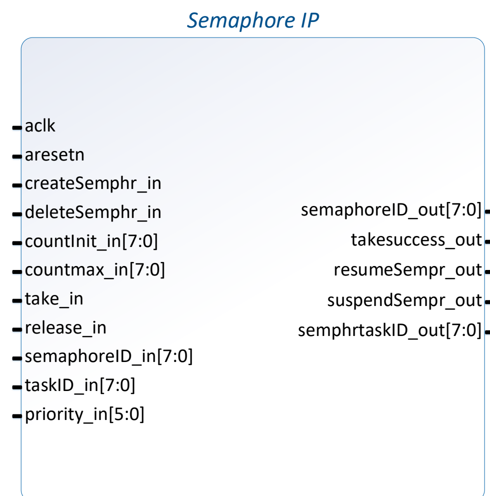


Figure 4.17: Semaphore IP.

the semaphore ID (semphoreID_in), and the identification of the task that calls the "take" or "release" (taskID_in), as well as its priority (priority_in). Furthermore, the Figure 4.17 illustrates the I/O of the Semaphore Module.

The hSmphr IP stores the semaphores information (maximum of counts and present value of releases) in the Semaphore List. If a "take" occurs the value of releases is decremented. If a "release" command occurs the value of releases is incremented without exceeding the maximum value. In case of the counting value is null, meaning that the semaphore cannot perform "take" operations, the hSmphr IP suspends the task that request the "take" operation and inserts the task in the Waiting List. The waiting list sorts the blocked tasks by their priority. When a "release" command is invoked, the semaphore resume the higger priority task that was waiting for a release, and not the task that was waiting the longest. Each semaphore as an associated waiting list, then multiple tasks can be waiting for a release. However, a task can not be waiting for more than one semaphore in an instant.

Figure 4.18 depicts an example of list behaviour when there are three semaphores, with tasks on hold, and a semaphore with availability to run a "take" command.

4.2.7 Secure Guest OS modifications

As FreeRTOS was used as the secure guest, it was needed to make some modifications to it so it will execute using FIQs instead of IRQs. As such, the core interrupt handling code was modified and the GIC CPU interface configured to signal secure interrupts as FIQs. The latter is simply done by setting the FIQEn bit in the CPU Interface Control Register (ICCICR).

The major issue with this modifications is the fact that, in the Zynq implementation, FIQs are non-maskable from the interrupt perspective. It is only set by hardware when certain exceptions

occur, such as the triggering of a FIQ. This seriously complicates the FreeRTOS interrupt code with respect to interrupt nesting, as this code originally enables and disables IRQs by masking them in the I bit of the CPSR. Furthermore, this problem is aggravated by the fact that internally, the implementation of the yield operation is done through the use of a SVC (supervisor call) instruction to directly jump to the exception vector and execute the needed yield operations. When using IRQs, this code assumes their automatic masking when the trap is triggered, which does not happen for FIQs. Not having FIQs maksed in these critical sections, removes the guarantee for their atomic execution, which might totally disrupt the correct execution of the RTOS. To solve this, the yield macro was replaced for a software generated interrupt, namely SGI 0, previously configured as secure, by writing to the ICDSGIR register of the distributor. As such, the FIQ handling code only needs to redirect control flow to the yield code (and first acknowledge and end the interrupt) in case the acknowledge interrupt is "0", and proceed normally otherwise. Nesting code however, becomes permanently "damaged" as are we not able to unmask and mask FIQs at will, so this functionality will not be completely available in the pure software version of the secure guest RTOS.

A final note regarding interrupts in the secure FreeRTOS guest: when configuring them, care must be taken so that they always have a higher priority than non-secure IRQs. This is simply done by assigning a priority in the lower half of the spectrum (as ARM interrupt priority scale is inverted), as TrustZone hardware forces secure software to configure priorities in the higher half of the spectrum.

4.3 Non-Secure VM (Linux with HDMI video output)

The non-secure world host the general-purpose guest OS, Linux. The massive support community and user base for this GPOS allows the existence of documentation, software modules and FPGA reference designs compatible, mitigating the engineering effort. GPOS are useful for running human-machine interfaces as well as internet-based applications and services. In order

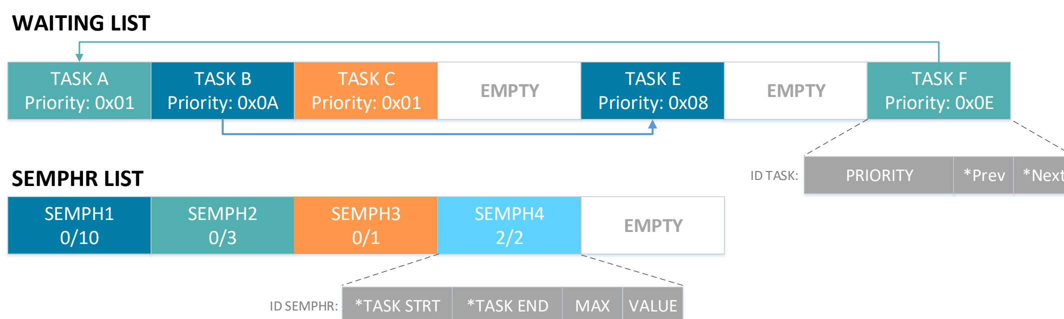
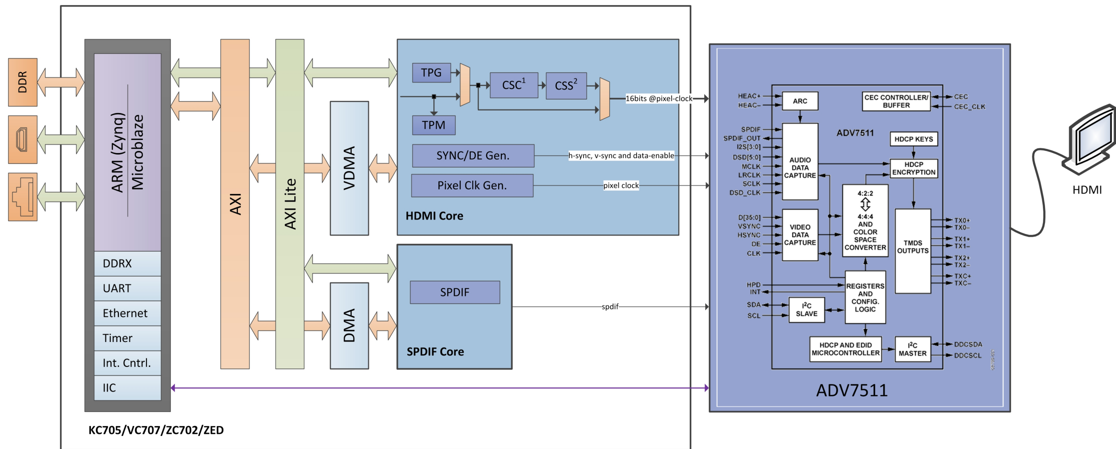


Figure 4.18: Semaphore Lists.



1. RGB to YCbCr Color space conversion (not applicable to VC707).
2. 444 to 422 Subsampling (not applicable to VC707).

Figure 4.19: ADV7511 reference design

to enrich the multimedia support, Analog Devices provides a reference HDL design which contain support for generating the necessary video and audio as well as support for interfacing as well as Linux driver for the High-Definition Multimedia Interface (HDMI) transmitter, ADV7511.

The reference design illustrated in figure 4.19 [HDM18] consists of two independent core modules. The video part consists of a Xilinx VDMA interface and the ADV7511 video interface. The ADV7511 interface consists of a 16bit YCbCr 422 (a format of color spaces used as a part of the color image pipeline in video and digital photography systems) with separate synchronization signals. The Video Direct Memory Access (VDMA) streams frame data to this core. The internal buffers of this core are small (1k) and do not buffer any frames as such. Additional resources may cause loss of synchronization due to DDR bandwidth requirements. The video core is capable of supporting any formats through a set of parameter registers (given below). The pixel clock is generated internal to the device and must be configured for the correct pixel frequency. It also allows a programmable color pattern for debug purposes. A zero to one transition on the enable bits trigger the corresponding action for HDMI enable and color pattern enable.

The reference design defaults to the 1080p video mode. The video settings can be change. The (HDMI Core) requires a corresponding pixel clock to generate the video. The reference design reads 24bits of RGB data from DDR and performs color space conversion (RGB to YCbCr) and down sampling (444 to 422). If bypassed, the lower 16bits of DDR data is passed to the HDMI interface as it is. A color pattern register provides a quick check of any RGB values on the monitor. If enabled, the register data is used as the pixel data for the entire frame.

The audio part consists of a Xilinx DMA interface and the ADV7511 spdif audio interface. The audio clock is derived from the bus clock. The audio data is read from the DDR as two 16bit words for the left and right channels. It is then transmitted on the SPDIF frame. The reference design defaults to 48KHz.

The ADV7511 driver is implemented as a IP encoder slave driver. In a typical board design the ADV7511 is not used as a standalone component but rather as a HDMI encoder fronted for some other devices with a graphics core, like a SoC or a FPGA. Implementing the ADV7511 driver as a IP encoder slave driver allows to reuse the driver between different platforms which use the ADV7511.

4.3.1 Non-Secure Guest OS modifications

The Linux runs in the supervisor mode of the non-secure world side. The non-secure VM is completely isolated from the privileged software running on the secure world side. The main limitation posed on the OS hosted on the non-secure side is that it can no longer use the TrustZone features by itself. The virtual architecture is not completely identical to the physical one, but it is identical to the bare architecture without TrustZone enhancements with an offset due of the reserved non-secure space. Notwithstanding, all the hardware modules in the reconfigurable hardware that pertain to the GPOS are declared as non-secure slaves of the secure AXI Master Bus.

Follow, some of the modifications made after setting the Xilinx original Linux with the default configuration provided by Analog Devices:

- FIQ stack initialization were removed, due to IRQs being used instead of FIQs, according to the LTZVisor interrupt model;
- Updates Linux device tree with specific LTZVisor bootargs (e.g. clock frequency, memory limitation);
- Clean the Filtering_Start_Address_Register, so the DDR start from 0.

Listing 4.2: Offtsets Declaration

```
SECTIONS {
    . = PHY_OFFSET;
    .texts : {
        _start:
        start.o(.text);
        start.o(.rodata);
    }
    . = NS_OFFSET + DTS_OFFSET;
    .textd : {
        d.tmp(.data);
        . = . + 0x1000;
    }
    . = NS_OFFSET + ZIMAGE_RAMDISK;
    .textr : {
        r.tmp(.data);
    }
    . = NS_OFFSET + ZIMAGE_OFFSET;
    .textz : {
        _start_linux:
```

```
        z.tmp(.data);
    }
}
```

After building the modified Linux, the reserved section for the compiled kernel (zimage), the device tree struct (dts) and the root file system (ramdisk) are declared in the non-secure space. Listing 4.2 illustrate the linker script file used to declare those offsets.

4.4 LTZVisor Modifications

This section intend to describe some modification made to the original lightweight TrustZone hypervisor (LTZVisor) code regarding:

- decouple the secure guest, which was tightly entangled with the hypervisor itself being compiled together as a single image.
- endowing LTZVisor with multi-processing VM support and so the hardware-software VM . While the secondary core executes in a normal state hosting exclusively the non-privileged software (i.e. the Linux guest), the primary core executes the privileged real-time guest in the secure state, scheduling the non-secure guest on this core in the idle period of the hardware-software RTOS.

Before describing the LTZVisor modifications carried out on this implementation, this section will delve a little bit deeper on the internal implementation details and build system of the LTZVisor, for which an overview is given on section 3.4.

LTZVisor achieves its main objective of dual guest execution by carefully assigning as non-secure only the needed resources for the non-secure guest to execute. This is accomplished by configuring the TrustZone module registers [Xil14] according to the needs of the non-secure guest's peripheral and memory use. Regarding memory, TrustZone segment assignment is done according to a predefined memory layout, as the non-secure guest must be previously compiled to execute specifically within the bound of this configuration. Furthermore, registers of the APU mpcore module, namely the SNSAC (SCU Non-secure Access Control Register) and the ICDISRX (Interrupt Security Register) registers of the GIC distributor, are configured to assign internal core components (e.g. the global timer) and non-secure interrupts to this guest, respectively. At the same time, the core's GIC interface is configured to signal secure interrupts as FIQs (in the CPU Interface Control Register or ICCCICR).

LTZVisor code executes in the monitor mode (which is always considered as running in secure state independently of the NS bit of the CP15 Secure Configuration Register), and performs the context switch between the non-secure and secure guests. After the aforementioned system

initialization, the context of both guests is initialized. Then LTZVisor executes the following steps to execute the context-switch to the secure-guest:

1. Sets the NS bit to secure.
2. Configures the FIQ bit of the SCR to route FIQs directly to FIQ mode (in the secure exception vector).
3. Restores secure guest context and transfers control flow to it (the first time this occurs, the hypervisor directly branches to the main function of the non-secure guest).

After this, the core is in completely control of the processor. It may execute until it seems fit. When entering an idle state it must emit a SMC to trigger a trap to the hypervisor which will then perform the sequence:

1. Sets the NS bit to non-secure.
2. Configures the FIQ bit of the SCR to route FIQs directly to Monitor mode (in the monitor exception vector).
3. Restores non-secure guest context and transfers control flow to it.

The non-secure guest will then execute until a FIQ, configured previously by the secure guest, is triggered. When this happens, the hypervisor acknowledges the interrupt and branches to a secure guest defined interrupt service routine. After the guest returns, the steps for the secure-guest context restore are repeated in the whole cycle repeats. This asymmetric scheduling scheme gives complete control of the system the secure RTOS guest, which will always execute when it deems fit by accordingly configuring it interrupts.

4.4.1 Secure Co-Design Guest Decoupling

One important aspect of the original LTZVisor implementation is that the secure guest is compiled with the hypervisor in a single image. This allows the aforementioned direct branches (at start-up and for interrupt servicing) from the hypervisor to the secure guest, which entangles both implementations, making the guest's interrupt servicing and nesting software highly dependent on the hypervisor. For this reason, the makefile and FIQ routine of the hypervisor were altered so that the secure guest is compiled completely from the remain system. In this way no direct branches happen from the hypervisor to the secure guest - both are completely agnostic of each others implementation. As such the secure guest will be included as a raw binary in the final system image, in the same way the non-secure guest is. This as two implications in the functioning of the hypervisor described above:

- The hypervisor cannot directly jump to the main routine of the secure guest at start-up. It will instead jump to the reset expection vector entry, which must coincide with the base of the guest image.
- The hypervisor cannot directly branch to the guest's interrupt service routine when a FIQ is triggered while the secure guest executes. Instead, it will be agnostic of the FIQ being serviced and not acknowledge it but directly restore the context of the secure guest. As of this moment, the FIQ mask in the CPSR is cleared, and the interrupt is still pending, immediately after the guest is resumed, the hardware will jump to the guests secure exception vector, which will service the interrupt as normally happens when executing standalone.

4.4.2 Multi-Processing

The adaptation of LTZVisor to a multicore-processing configuration is relatively simple. The main idea here is that while the primary core behaves as the single-core implementation of LTZVisor, by giving almost full control to the secure RTOS guest and scheduling the non-secure guest on its idle periods, the secondary core is dedicated to exclusively execute the non-secure guest,

To achieve this, only small modifications are needed at the start-up and initialization code of LTZVisor, as well as duplicate parts of the context of the non-secure guest. First, the primary core run all TrustZone configurations as it was in the original version. Then, when the non-secure VM initialization starts and attempt to initialize the secondary core, the hypervisor first takes place and executes the needed TrustZone configurations in order to declare the secondary core as non-secure, before handling execution control to the non-secure guest. Some care is needed also to initialize some core specific structures such as the GIC CPU interface in both cores.

5. Evaluation

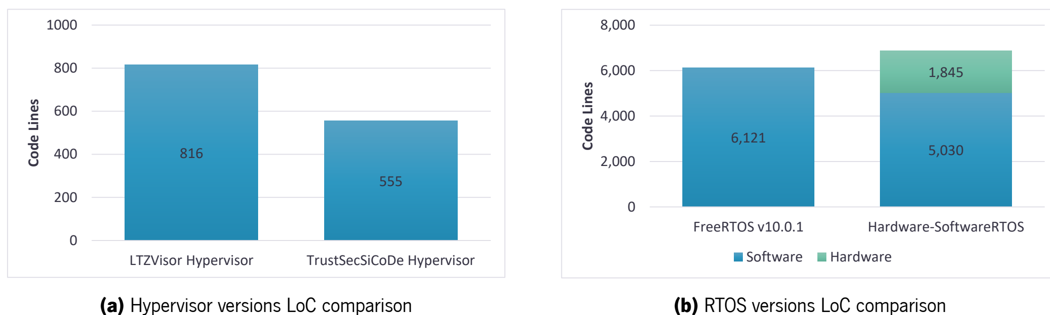
The Chapter 4 has describe the implementation of the TrustZone-assisted Secure Silicon Co-Design framework. This chapter goes through an extensive evaluation. The implemented solution was evaluated on a ZedBoard evaluation board targeting a dual ARM Cortex-A9 running at 667 MHz and 100 MHz on the Series-7 Programmable Logic. This evaluation focused on three metrics: memory footprint, performance overhead and real-time behaviour.

5.1 Engineering effort

The engineering effort required for the implementation of the developed framework was measured using the Understand software tool. This tool provide pertinent metric and reports regarding the developed code. In the context of this thesis, we have focused on the number of lines of code (LoC) of the TCB of the system (libraries and drivers were not taking in consideration).

Figure 5.1a illustrate the comparison between number of LoC of the LTZVisor and the modified version for this developed framework. It is noted that there is a reduction in terms of LoC between hypervisor. This reduction is related to the fact that in this developed version the secure guest is no longer part of the hypervisor's TCB.

Figure 5.1b depicts the comparison between number of LoC of FreeRTOS v10.0.1 and the hardware-assisted developed version, and so the LoC written in HDL language for the developed hardware modules. It was expected a reduction in terms of LoC in the version developed due the offloading of several OS services to hardware. It is noteworthy that since this approach rely



(a) Hypervisor versions LoC comparison

(b) RTOS versions LoC comparison

Figure 5.1: Engineering effort results

on the structure of FreeRTOS and attempt to maintain its architecture, the code refactoring could never be excessive. Thereupon, the conjunction of software and hardware code presents values slightly higher than the original version of FreeRTOS.

5.2 Memory Footprint

In order to assess the memory footprint of each software component of the implemented architecture, we used the size tool of the Arm GNU toolchain. The original and hardware-software co-design version of FreeRTOS (v10.0.1), as well as the original and modified version of LTZVisor hypervisor were evaluated without taking in consideration the boot code, libraries and drivers. Table 5.1 presents the collected measurements. As it can be seen the memory overhead introduced by the hypervisor remains very small, just like the original LTZVisor, i.e., 2816 bytes. As it can be seen, the hardware-software version of the secure VM RTOS has smaller memory footprint compared to the original version. This reduction comes from hardware offloading of the components that caused the most overhead to the real-time operating system.

Table 5.1: TrustSecSiCode memory footprint (bytes)

Software	Memory Footprint			
	.text	.data	.bss	Total
LTZVisor Trust SecSi Code	2816	0	260	3076
LTZVisor	2368	0	512	2880
Hardware-software RTOS	18554	0	120	18674
FreeRTOS v10.0.1	21526	0	360	21886

5.3 Hardware Costs

Since both VMs employ services on the reconfigurable hardware layer, it is worth to mention the hardware costs for the developed framework. Vivado post-implementation offers a set of reports regarding timing, power and utilization. Figure 5.2 depicts the Vivado utilization report. Vivado utilization report parameters indicates the number of lookup tables (LUTs), LUTRAM, flip-flops (FFs), buffer random access memory BRAM, I/Os and mixed-mode clock manager (MMCM) of the current design required in order to implement the secure VM hardware RTOS services (illustrated in grey colour) and the HDMI transmitter module used by the non-secure VM (illustrated in cyan colour).

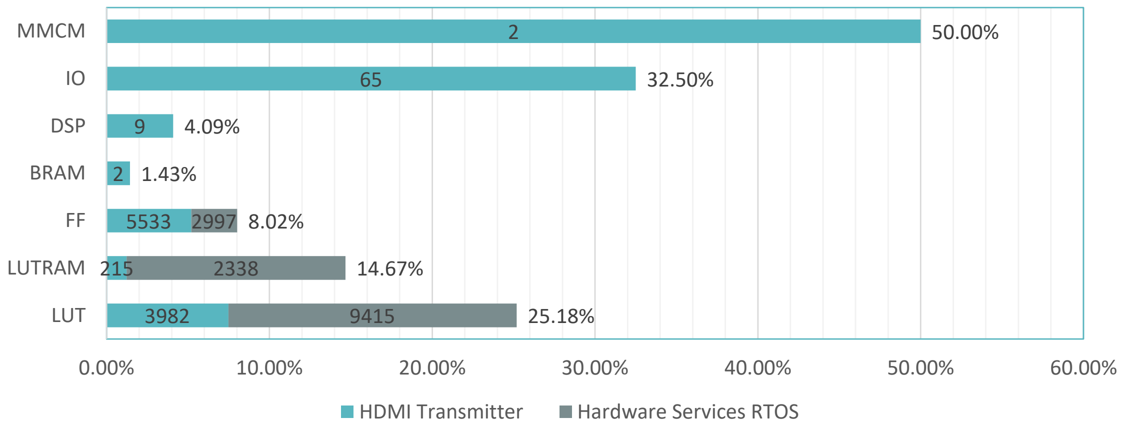


Figure 5.2: TrustSecSiCoDe framework hardware costs

There is not much to point out about the hardware costs pertaining to the HDMI transmitter because because it is a third-party implementation. Nevertheless, it is relevant to present them together with the results of the migrated RTOS services towards analysing the overall hardware costs. The offloaded RTOS services requires a significant amount of LUT and LUTRAM, due to the fact that in most cases these services are practically list management operations. This lists are stored in memory resources. In terms of PL resources, the overall system does not have an excessive consumption of resources available in the evaluated Zynq platform. Since the consumed resources do not exceed much more than the 20% of resources available in this platform, leaving enough development space for applications to be developed on top of this evaluation board.

5.4 Performance

The performance evaluation process was split into two different test case scenarios. First, evaluate the improvements introduced by the implemented hardware-software co-design RTOS (using the Thread Metrics Suite) as well as the latency deviation over the secure VM (Hardware-software RTOS). Then, evaluate the overhead over the non-secure VM (GPOS) using the LMBench.

5.4.1 Secure VM (Hardware-Software RTOS)

Thread-Metric Benchmark Suite provides a set of benchmarks to evaluate the RTOS real-time capabilities. As already described in section 3.5.1, the suite comprises seven benchmarks, evaluating the most common RTOS services and interrupt processing: cooperative context switching (CS); preemptive context switching (PS); interrupt processing (IP); interrupt processing with preemption (IPP); semaphore processing (SP); message passing (MP); and memory allocation and deallocation (MA). Each benchmark outputs a counter value, the higher the value, the smaller RTOS impact on the running application.

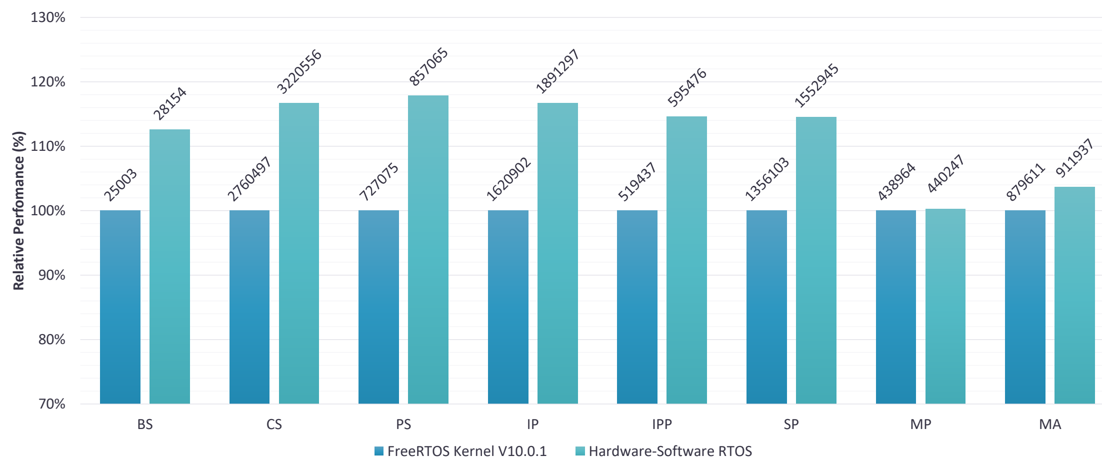
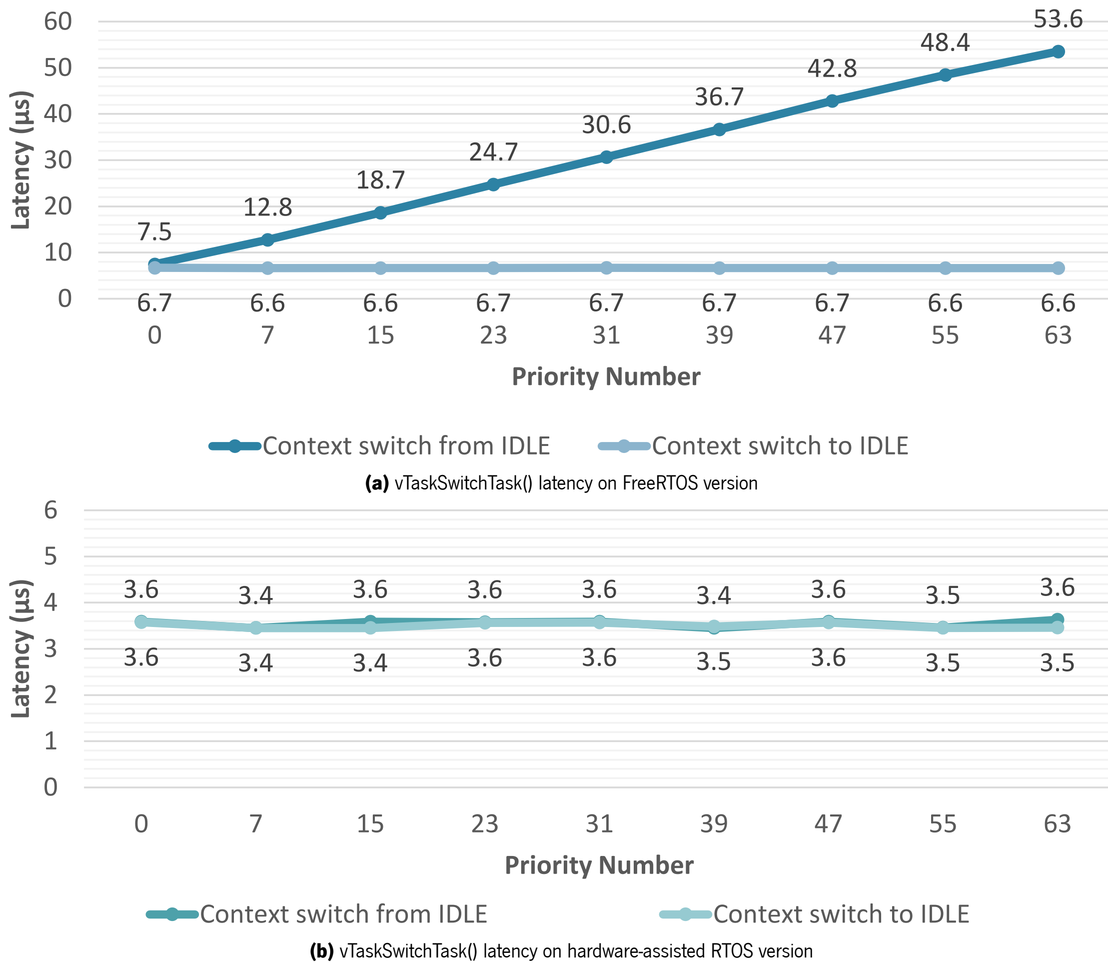


Figure 5.3: Thread-Metric benchmarks results

Benchmarks were executed on the original version of FreeRTOS (FreeRTOS Kernel v10.0.1), and on the hardware-software RTOS version (Hardware-Software RTOS). In both versions the interrupts are handled as FIQs. Because either versions are declared as secure guests on the Trust SecSi CoDe. Figure 5.3 presents the achieved results, corresponding to the average relative performance (as well as the average absolute performance) of 20 collected samples for each benchmark. Each sample reflects the benchmark score for a 30 seconds execution time, encompassing a total execution time of 500 minutes, per benchmark. In accordance with Figure 5.3 the hardware-software co-design version of FreeRTOS compared to the original version has an overall better performance. It is also possible to verify that the improvements in memory allocation and deallocation and message passing benchmarks were not as significant as the remaining tests. This phenomenon occurs because no messaging or memory allocation service has been offloaded to the reconfigurable hardware.

The hardware accelerators implemented to support the RTOS are directly related to the task switching mechanism. This way, to infer the benefits obtained from the hardware approach, the evaluation and validation was realized by measuring the latency and jitter in the manipulation of the various kernel data structures. Both hardware-software RTOS and FreeRTOS Kernel v10.0.1 were measure through a specific hardware counter module, called performance monitor unit (PMU).

Figure 5.4a and Figure 5.4b illustrate the task switch latencies on the FreeRTOS Kernel v10.0.1 and on the hardware-software RTOS, respectively. This test measures the latency of a switch context operation from the idle task to a task with the priority: 0, 7, 15, 23, 31, 39, 47, 55, 63. Also, this test measures the switch latency after on task (with the priorities previous mentioned) As can be seen on the collected data, the mean latency in the hardware-software RTOS is significant lower than the original FreeRTOS. More than that, regardless of the priority that the task has, this operation takes 1200 system ticks. This means hardware-software RTOS presents more determinism than the original version of FreeRTOS.



(a) vTaskSwitchTask() latency on FreeRTOS version

(b) vTaskSwitchTask() latency on hardware-assisted RTOS version

Figure 5.4: Context switch (vTaskSwitchTask()) latency results.

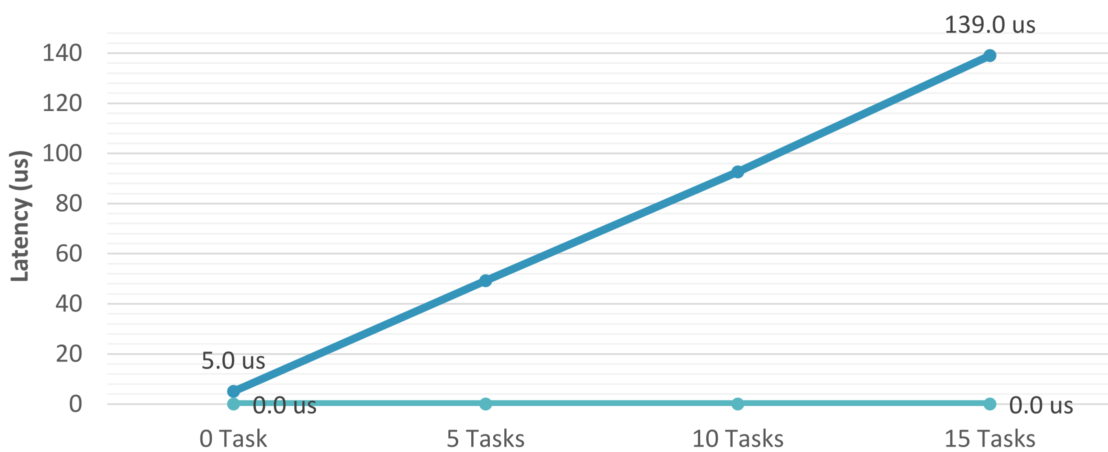
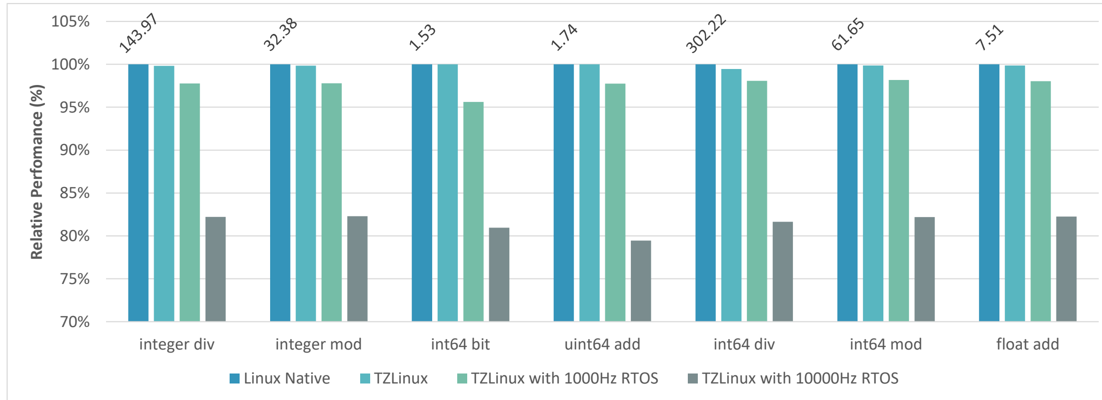
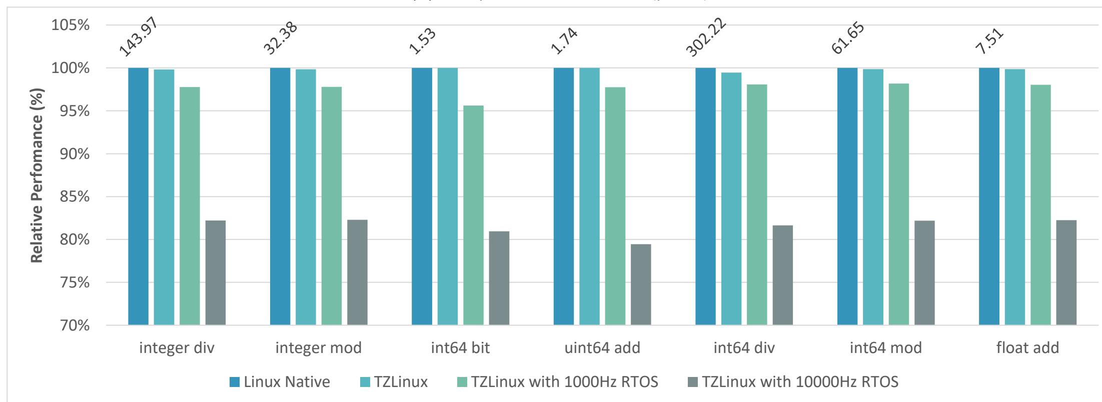


Figure 5.5: xIncrementTick() latency results

Through the results illustrated in the Figure 5.5, was possible to verify that the "xIncrementTick()" operation is a costly and time consuming procedure. Since, in the developed approach this procedure was completely moved to hardware, the RTOS processing system will be entirely released of it and, therefore, the latency is eliminated.



(a) lat_ops benchmark results (part 1)



(b) lat_ops benchmark results (part 2)

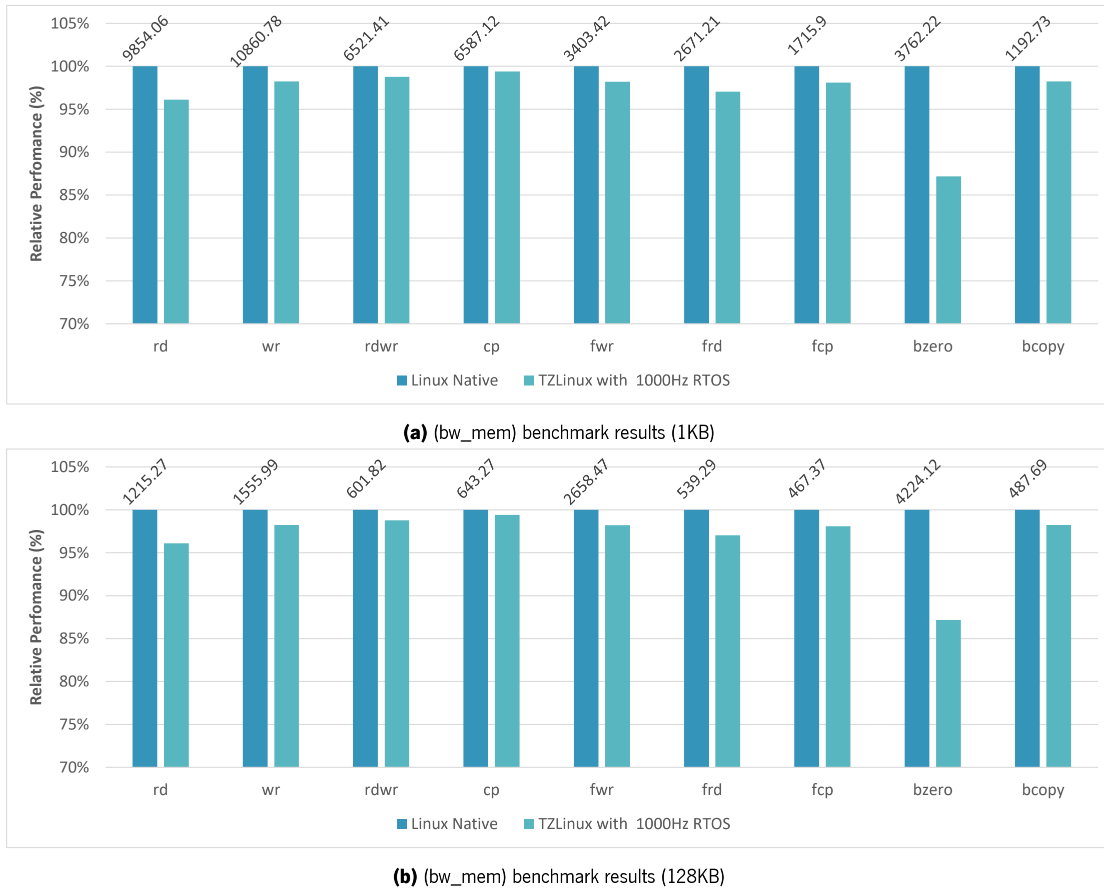
Figure 5.6: LMBench arithmetic operations latency (lat_ops) benchmark results.

5.4.2 Non-Secure VM (Linux)

LMBench is a widely used suite of micro-benchmarks that measure a variety of important aspects of system performance, such as latency and bandwidth. The LMBench 3.0 suite includes more than forty micro-benchmarks within three different categories: bandwidth, latency, and other. The evaluation was focused on two specific benchmarks:

- lat_ops: Arithmetic operations latency, to evaluate general CPU performance
- bw_mem: Memory operations bandwidth for different blocks size to evaluate the interference of the TrustZone Address Space Controller (TZASC);

The presented evaluation exhibit the results taken from four different application scenarios. For the first scenario, the micro-benchmarks were ran in the native version of Linux (Linux Native). Second, the micro-benchmarks in virtualized version (TZLinux) without any secure VM executing. For the third and fourth scenario, the micro-benchmarks were ran with the secure VM configured with a 1 MHz tick frequency and with 10MHz, respectively. Its worth to mention that no real time tasks were added to the secure VM. This means, in the case of the implemented hardware-assisted RTOS will not trigger the timer interrupt since there is no need to context switch. Therefore, for



(a) (bw_mem) benchmark results (1KB)

(b) (bw_mem) benchmark results (128KB)

Figure 5.7: LMBench memory bandwith (bw_mem) benchmark results

the purpose of this evaluation, a real-time task was created, this task simply suspends it self until next tick. This way the non-secure VM evaluation for the Trust SecSi Code framework can come closer to a real typical framework use since to run applications in RTOS it is necessary to associate them with tasks. Presented results correspond to the average relative performance and variation (as well as the average absolute performance) of the 10 consecutive experiments, encompassing a total of 1000 samples.

Figure 5.6 presents the achieved results for the arithmetic operations latency benchmark. The values on top of the bars correspond to the average latency, in nanoseconds. As it can be seen, the virtualized version of Linux with a 1 MHz tick frequency of the secure VM only presents an average performance degradation of 3%, when compared to its native execution. This value is practically uniform among all micro-benchmarks (apart from the small variations due to the benchmark's lack of accuracy). For these arithmetic operations latency -benchmarks, the achieved results do not reflect the real performance penalty, due to the lack of precision. The relative performance of GPOS at different RTOS tick rates it has an exponential decay behaviour for the core 0. Despite this it will not happen starvation on the non-secure world side since the core 1 is reserved for non-secure virtual machine use. Moreover, at 100 μs RTOS tick rate, the relative performance of one core used by the GPOS is 85%.

Figure 5.7 presents the achieved results for the memory bandwidth benchmark. The values on top of the bars correspond to the average memory bandwidth, in megabytes per second (MB/s). Figure 5.7a and Figure 5.7b depict the assessed results for a memory block size of 2KB and 128KB, respectively. These memory block sizes were selected with the intention to fit and not fit within the L1 and L2 cache sizes, respectively. Looking at the two figures, it is clear the relative performance of the system is practically uniform among all micro-benchmarks, presenting an average performance degradation of 3% when comparing to the virtualized version of Linux with the native one.

In summary, the non-secure VM's performance is inversely proportional to the secure VM's tickrate, but only in the primary core. Since the secondary core run exclusively the non-secure SMP operating system. Thereby the non-secure VM experience lack of performance on one of its cores (on the primary core to be more precise), which is much better than suffering completely starvation, which makes the purpose of this guest ineffective.

6. Conclusion

Embedded systems are proliferating at a rapid pace in our everyday life, representing a huge part of our key infrastructures. The trend nowadays goes towards the consolidation of a wide range of functions into the same hardware platform, while reducing size, weight, power and cost (SWAP-C) budget.

Solutions that guarantee the deadlines of real-time tasks, while at the same time, integrating rich environments for monitoring and network purposes are gaining momentum in embedded systems field. Virtualization technology allows the co-existence of mixed-criticality systems environments on a single physical platform. Hardware virtualization solutions are capable of providing efficient hypervisors and real-time guarantees. TrustZone-assisted virtualization has been seen as a promising approach, due to the ubiquitous presence of TrustZone-enabled processors.

Co-designed systems that exploit reconfigurable hardware technology are able to ensure the rigid real-time constraints of embedded domain. Virtualization solutions that rely on asymmetric scheduling policy can lead to the starvation over one of the guests. Multi-processing configuration demonstrate to be a viable solution to prevent the drawback of asymmetric designs.

This thesis proposed the TrustZone-assisted Secure Silicon Co-design framework (Trust SecSi CoDe). Trust SecSi CoDe is a hardware-software co-design framework for easing the economy of building the new generation of embedded devices. By bringing together TrustZone technology, virtualization, multiprocessing, and RTOS acceleration via configurable hardware, that embedded system developers and hobbyists will find into Trust SecSi CoDe framework a valuable resource to speed-up the development of current embedded applications on Xilinx Zynq-based devices.

6.1 Future Work

Although the developed framework is already at a good stage of development, there is still room for improvements and development. A good example is the implementation of an inter-partition communication (IPC) mechanism using VirtIO [RSPT18].

Our current priority is the implementation of use cases which might demonstrate the full potential of the framework across different embedded industries. Among existing embedded industries, we are particularly interested on the development of a demo for the Industrial IoT.

Figure 6.1 depicts a use case scenario of a conventional smart factory and respective production facilities. At the heart of the Industrial IoT are the Automation and Industrial Control Systems, which are computing platforms that monitor and control physical processes. In the presented use case scenario, the highlighted ICS unit is responsible for monitoring, controlling and connecting a robotic arm to the Internet. The Trust SecSi Code framework provides all technologies needed to address such requirements.

Currently, we are developing a rough prototype of such an application with the DOBOT Magician[1]. The framework is used to build a system that runs Ubuntu for monitoring (using a QT GUI), as well as the accelerated RTOS which interfaces with the robotic arm when the buttons available on the Zedboard are pressed.

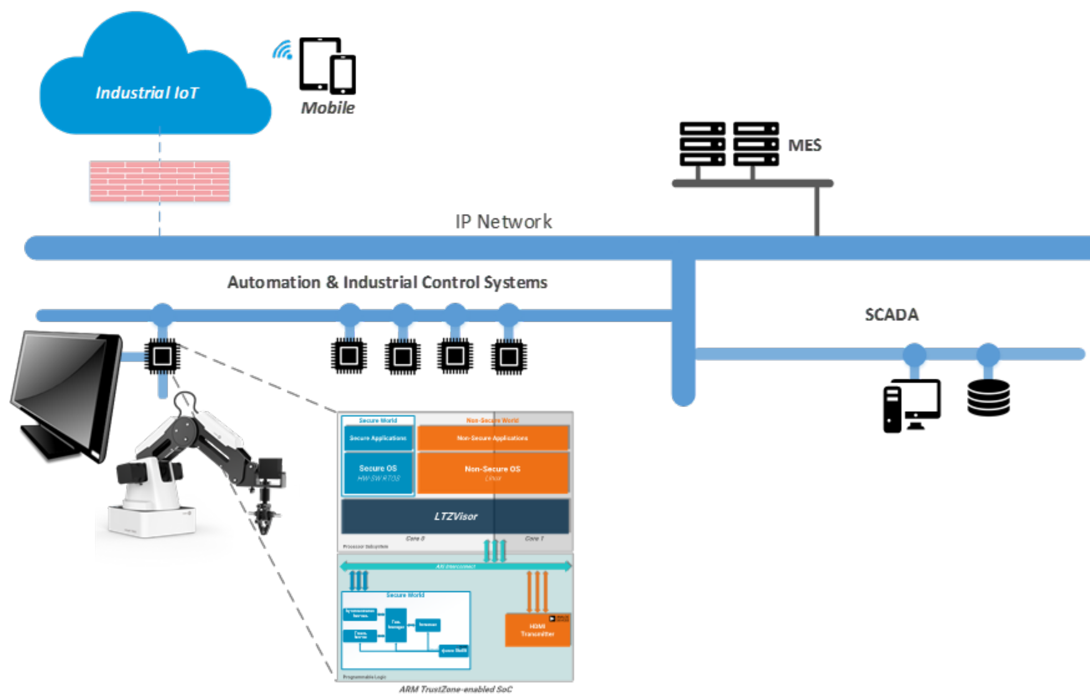


Figure 6.1: Trust SecSi Code for Industrial Control Systems.

[1] https://www.dobot.cc/dobot-magician/product-overview.html

References

[AG09] F. Armand and M. Gien. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In 2009 6th IEEE Consumer Communications and Networking Conference, pages 1–7, January 2009.

[AH10] A. Aguiar and F. Hessel. Embedded systems' virtualization: The next challenge? In Proceedings of 2010 21st IEEE International Symposium on Rapid System Protyping, pages 1–7, June 2010.

[APA+05] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass. hthreads: a hardware/software co-designed multithreaded RTOS kernel. In 2005 IEEE Conference on Emerging Technologies and Factory Automation, volume 2, pages 8 pp.–338, Sept 2005.

[ARM09] ARM. ARM Security Technology: Building a Secure System using TrustZone Technology. PRD29-GENC-009492C, April 2009.

[Bar09] R. Barry. FreeRTOS reference manual: API functions and configuration options. Real Time Engineers Limited, 2009.

[Bar10] R. Barry. Using the FreeRTOS Real Time Kernel - a Practical Guide. 01 2010.

[BBO+16] P. Burgio, M. Bertogna, I. S. Olmedo, P. Gai, A. Marongiu, and M. Sojka. A software stack for next-generation automotive systems on many-core heterogeneous platforms. In 2016 Euromicro Conference on Digital System Design (DSD), pages 55–59, Aug 2016.

[CEES14] L. Crockett, R. Elliot, M. Enderwitz, and R. Stewart. The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc. Strathclyde Academic Media, UK, 2014.

[CI18] Giulio Corradi and Dan Isaacs. Python Based Framework for Analytics in Industrial IoT. In Proceedings of Embedded World Conference, Nuremberg, Germany, March 2018.

[CSA+16] A. Carvalho, V. Silva, F. Afonso, P. Cardoso, J. Cabral, M. Ekpanyapong, S. Montenegro, and A. Tavares. Full virtualization on low-end hardware: A case study. In

IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, pages 4784–4789, October 2016.

[DN14] C. Dall and J. Nieh. KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor. SIGPLAN Not., 49(4):333–348, February 2014.

[FLWH10] T. Frenzel, A. Lackorzynski, A. Warg, and H. Härtig. ARM Trustzone as a Virtualization Technique in Embedded Systems. In Proceedings of Twelfth Real-Time Linux Workshop, Nairobi, Kenya, 2010.

[G. 18] G. Corradi. The Value of Python Productivity: Extreme Edge Analytics on Xilinx Zynq Portfolio. Xilinx, White Paper, WP502 (v1.0), June 2018.

[GGP+16] T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares. Bringing Hardware Multithreading to the Real-Time Domain. IEEE Embedded Systems Letters, 8(1):2–5, March 2016.

[GP09] Y. Gosain and P. Palanichamy. TrustZone Technology Support in Zynq-7000 All Programmable SoCs. 2009.

[GPG+15] T. Gomes, J. Pereira, P. Garcia, F. Salgado, V. Silva, S. Pinto, M. Ekpanyapong, and A. Tavares. Hybrid real-time operating systems: deployment of critical FreeRTOS features on FPGA. 8, 01 2015.

[GZ12] Z. Gu and Q. Zhao. A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization. 5:277–290, 01 2012.

[HDM18] Adv7511 xilinx evaluation boards reference design. [Online]. Available:, Accessed on: September 14, 2018.

[Hei08] G. 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, New York, NY, USA, 2008. ACM.

[Hei11] G. Heiser. Virtualizing embedded systems - why bother? In 2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC), pages 901–905, June 2011.

[Kai09] R. Kaiser. Complex embedded systems - A case for virtualization, 07 2009.

[KGJ03] P. Kohout, B. Ganesh, and B. Jacob. Hardware support for real-time operating systems. In First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721), pages 45–51, October 2003.

[KHV14] B. Kauer, H. Haertig, and P.E. Veríssimo. Improving System Security Through TCB Reduction. Saechsische Landesbibliothek- Staats- und Universitaetsbibliothek

Dresden, 2014.

[LASS12] Anders Blaabjerg Lange, Karsten Holm Andersen, Ulrik Pagh Schultz, and Anders Stengaard Sørensen. HartOS - a Hardware Implemented RTOS for Hard Real-time Applications. IFAC Proceedings Volumes, 45(7):207 – 213, 2012. 11th IFAC,IEEE International Conference on Programmable Devices and Embedded Systems.

[Lie95] J. Liedtke. On Micro-kernel Construction. SIGOPS Oper. Syst. Rev., 29(5):237–250, 1995.

[Lin91] L. Lindh. Fastchart-a fast time deterministic CPU and hardware based real-time-kernel. In Proceedings. EUROMICRO '91 Workshop on Real-Time Systems, pages 36–40, June 1991.

[LMP+05] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks, 01 2005.

[LP09] E. Lübbers and M. Platzner. ReconOS: Multithreaded Programming for Reconfigurable Computers. ACM Trans. Embed. Comput. Syst., 9(1):8:1–8:33, 2009.

[MAC+17] J. Martins, J. Alves, J. Cabral, A. Tavares, and S. Pinto. RTZVisor: A Secure and Safe Real-Time Hypervisor. 6:93, 10 2017.

[MNK14] H. Shinya T. Hiroaki M. Naotaka, I. Takuya and S. Katsunobu. ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems,. pages 9–16, 2014.

[MS96] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[NA07] S. Nordstrom and L. Asplund. Configurable Hardware/Software Support for Single Processor Real-Time Kernels, November 2007.

[OK13a] R. Oshana and M. Kraeling. Chapter 1 - Software Engineering of Embedded and Real-Time Systems. Newnes, Newton, MA, USA, 1st edition, 2013.

[OK13b] R. Oshana and M. Kraeling. Chapter 25 - linux for embedded systems. Newnes, Newton, MA, USA, 1st edition, 2013.

[Pal03] S. Palnitkar. Verilog®Hdl: A Guide to Digital Design and Synthesis, Second Edition. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2003.

[PKR+13] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere. Formal Virtualization Requirements for the ARM Architecture. J. Syst. Archit.,

59(3):144–154, March 2013.

[POP+14] J. Pereira, D. Oliveira, S. Pinto, N. Cardoso, V. Silva, T. Gomes, J. Mendes, and P. Cardoso. Co-Designed FreeRTOS Deployed on FPGA. In 2014 Brazilian Symposium on Computing Systems Engineering, pages 121–125, November 2014.

[POP+17] S. Pinto, A. Oliveira, J. Pereira, J. Cabral, J. Monteiro, and A. 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+17] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral. LTZVisor: TrustZone is the Key. In Marko Bertogna, editor, 29th Euromicro Conference on Real-Time Systems, volume 76 of Leibniz International Proceedings in Informatics (LIPIcs), pages 4:1–4:22, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[PRAM17] M. Valdes Pena, J. Rodriguez-Andina, and M. Manic. The Internet of Things: The Role of Reconfigurable Platforms. IEEE Industrial Electronics Magazine, 11(3):6–19, Sept 2017.

[PS] S. Pinto and N. Santos. Demystifying Arm TrustZone: A Comprehensive Survey. ACM Computing Surveys, preprint.

[PS12] S. Panneerselvam and M. Swift. Operating Systems Should Manage Accelerators. In Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism, HotPar'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[PTM16a] S. Pinto, A. Tavares, and S. Montenegro. Hypervisor for Real Time Space Applications. In The 4S Symposium, 2016.

[PTM16b] S. Pinto, A. Tavares, and S. Montenegro. Space and time partitioning with hardware support for space applications. DASIA 2016, Proceedings of the Conference, May 2016.

[QLG+15] E. Qaralleh, D. Lima, T. Gomes, A. Tavares, and S. Pinto. HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM multicore. In 2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA), pages 1–4, Sept 2015.

[RSPT18] J. Ribeiro, N. Silva, S. Pinto, and A. Tavares. A trustzone-assisted hypervisor supporting dynamic partial reconfiguration, 02 2018.

[SGB+16] J. Shuja, A. Gani, K. Bilal, A. Khan, S. Madani, S. Khan, and A. Zomaya. A Survey of Mobile Device Virtualization: Taxonomy and State of the Art. ACM Comput. Surv., 49(1):1:1–1:36, 2016.

[SHT10] D. Sangorrin, S. Honda, and H. Takada. Dual Operating System Architecture for

Real-Time Embedded Systems. In Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT), Brussels, Belgium, pages 6–15, 2010.

[SK10] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In Proceedings of the 5th European Conference on Computer Systems, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM.

[SOLMTS04] M Sindhwani, Timothy Oliver, D L Maskell, and And T. Srikanthan. RTOS Acceleration Techniques-Review and Challenges. 01 2004.

[SOSA08] N. Silva, A. Oliveira, R. Santos, and L. Almeida. The OReK real-time micro kernel for FPGA-based systems-on-chip. In 2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, pages 75–80, October 2008.

[SR04] J. Stankovic and R. Rajkumar. Real-Time Operating Systems. Real-Time Syst., 28(2-3):237–253, 2004.

[SWP04] C. Steiger, H. Walder, and M. Platzner. Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks. IEEE Transactions on Computers, 53(11):1393–1407, November 2004.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. Computer, 33(3):78–85, March 2000.

[WCW+09] Y. Wang, W. Chen, X. Wang, H. You, and C. Peng. The Hardware Thread Interface Design and Adaptation on Dynamically Reconfigurable SoC. In 2009 International Conference on Embedded Software and Systems, pages 173–178, May 2009.

[Xil14] Xilinx. Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC. User Guide, UG1019 (v1.0), May 2014.

[XPN15] T. Xia, J. Prevotet, and F. Nouvel. Mini-NOVA: A Lightweight ARM-based Virtualization Microkernel Supporting Dynamic Partial Reconfiguration. In 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pages 71–80, May 2015.

[ZMH15] S. Zampiva, C. Moratelli, and F. Hessel. A hypervisor approach with real-time support to the MIPS M5150 processor. In Sixteenth International Symposium on Quality Electronic Design, pages 495–501, March 2015.

[ZQCP05] B. Zhou, W. Qiu, Y. Chen, and C. Peng. SHUM-uCOS: A RTOS using multi-task model to reduce migration cost between SW/HW tasks. In Proceedings of the Ninth International Conference on Computer Supported Cooperative Work in Design, 2005., volume 2, pages 984–989 Vol. 2, May 2005.