MASTER's Degree Thesis

Porting FreeRTOS Symmetric Multiprocessing support to the Zynq-7000 SoC

Supervisors

Prof. STEFANO DI CARLO

Prof. ALESSANDRO SAVINO

Doctor. ENRICO MAGLIANO

Doctor. ALESSIO CARPEGNA

Candidate

MATTEO FRAGASSI

JULY 2025

Summary

Real-Time Operating Systems (RTOSs) are widely employed in time-sensitive applications. However, since keeping costs low is crucial, the adopted hardware is usually not state-of-the-art, even though safety and responsiveness are often among the requirements of the system. Therefore, it is important to exploit all the resources the hardware has to offer. From the perspective of an Operating System (OS), it has become almost imperative to be able to support multi-core processing with a single instance of the OS. In this context, the thesis focuses on porting the Symmetric Multiprocessing (SMP) functionality of FreeRTOS, a popular open-source RTOS, on the multi-core processor embedded in the Zynq-7000 System-on-Chip (SoC). The process is detailed in both the hardware-dependent and hardware-independent steps and the result is compared to other existing multiprocessing options.

Acknowledgements

I would like to thank all my professors for the knowledge and guidance they have provided during my academic journey. A special mention goes to the supervisors and co-supervisors who supported me throughout the development of this thesis.

Un grazie speciale va poi alla mia famiglia, senza la quale non avrei affrontato questo percorso con la serenità e la consapevolezza che lo hanno contraddistinto.

Un pensiero va anche a nuovi e vecchi amici, che hanno saputo rendere leggeri e spensierati i momenti più duri, a prescindere dalla distanza che ci separava.

Infine, un ringraziamento particolare va a Michela, la mia ragazza, che ha contribuito a rendere questo viaggio un ricordo indelebile.

Table of Contents

List of Tables

List of Figures

List of Listings

Acronyms

AAPCS

ARM Architecture Procedure Call Standard.

AMP

Asymmetric Multiprocessing.

AP

Application Project.

API

Application Programming Interface.

APU

Application Processing Unit.

BSP

Board Support Package.

CA9

Cortex-A9.

CPSR

Current Program Status Register.

CPU

Central Processing Unit.

CU

Computational Unit.

DDR

Double Data Rate.

DMB

Data Memory Barrier.

DSB

Data Synchronization Barrier.

EABI

Embedded Application Binary Interface.

ES

Embedded System.

FP

Floating Point.

FPGA

Field Programmable Gate Array.

FSBL

First Stage Boot Loader.

GIC

Generic Interrupt Controller.

GUI

Graphical User Interface.

HDL

Hardware Description Language.

ISB

Instruction Synchronization Barrier.

L1

Level 1.

L2

Level 2.

MCU

Microcontroller Unit.

MMU

Memory Management Unit.

MP

Multiprocessing.

MPU

Microprocessor Unit.

OS

Operating System.

PL

Programmable Logic.

PoC

Point of Coherency.

PoU

Point of Unification.

PP

Platform Project.

PPI

Private Peripheral Interrupt.

PS

Processing System.

RTOS

Real-Time Operating System.

SCU
: Snoop Control Unit.

SDT
: System Device Tree.

SEV
: Send Event.

SGI
: Software Generated Interrupt.

SMP
: Symmetric Multiprocessing.

SoC
: System-on-Chip.

SP
: Stack Pointer.

SPI
: Shared Peripheral Interrupt.

SPSR
: Special Program Status Register.

TMR
: Technical Reference Manual.

TTC
: Triple Timer Counter.

UART
: Universal Asynchronous Receiver-Transmitter.

WFE
: Wait For Event.

Chapter 1

Introduction

In the last decades, the number of devices surrounding everyone has drastically increased, along with their complexity. These devices are usually designed to perform a few specific functions, and they are referred to as Embedded Systems (ESs). Technological improvements have made the definition of ES increasingly vague. For example, a smartphone is usually considered an ES, but today its functionality makes it more similar to a general-purpose computer. Even if this is less noticeable in other devices (e.g., appliances, cars), everything has become so smart that it contains at least a Microcontroller Unit (MCU) or/and a Microprocessor Unit (MPU). Obviously, the hardware still has a non-negligible cost, so an ES is rarely implemented with state-of-the-art technology. Because of this, exploiting all the available resources becomes crucial.

The software design of simple systems can be coded in bare metal to optimize resource usage, thus saving on hardware costs. However, as the complexity of the applications grows, the required effort increases drastically. In this scenario, having a software layer that manages the underlying hardware is useful so programs can be more easily developed and ported to other devices. In the embedded domain, as many applications are time-sensitive or need to be predictable, the most common intermediate layer is a Real-Time Operating System (RTOS).

For many years, creating devices with multiple instances of a Computational Unit (CU) has been the main strategy to overcome the performance barrier imposed by the limits of single-core processors. Systems that can manage many CUs support some form of Multiprocessing (MP), but how this is exploited by the software directly depends on the target hardware. The most common approaches are known as Symmetric Multiprocessing (SMP) and Asymmetric Multiprocessing (AMP). Historically, the latter has been more widely used in the embedded domain because it can manage heterogeneous groups of processors, thus allowing better allocation of a device's resources to specific parts of an application. SMP can, in principle, provide better performance. Still, its design is less flexible than AMP, since the processors must comply with the same architecture and the software has inherently more dependencies. Moreover, because supporting SMP on a processor introduces some overhead, developing multi-core processors for ESs has encountered some difficulties. Despite these initial problems, many embedded devices are now equipped with multiple CU [1] to meet the requirements of modern applications.

This thesis faces the problem of porting the SMP support of an RTOS to hardware that is equipped with symmetric CUs. The scope is to show that even older platforms can benefit from improved software support, leading to better utilization of the systems' resources. Nevertheless, the process is not straightforward, since the code created for a single-core device cannot run on multiple cores without being modified. If more than one CU is considered, the software structure must be enhanced with appropriate synchronization mechanisms to ensure that no race condition can disrupt the correct execution of an application.

More specifically, the thesis illustrates the process of adapting the SMP functionality of FreeRTOS, a popular open-source RTOS, to Zynq-7000 System-on-Chips (SoCs), a family of devices released in 2011. This part is preceded by general concepts, which provide the theoretical knowledge required to understand the porting procedure. The description of the Zynq-7000 SoC, the Cortex-A9 (CA9) processor that powers it, and the tools used during the development are only some of the treated arguments. Furthermore, the steps that lead to the final implementation of the SMP port for the Zynq-7000 SoC are presented in a way that eases the comprehension, since the discussion is divided into topics rather than being a mere changelog. As a result, the focus is placed on the hardware-software dual-core configuration and the changes that address it. Some time will also be spent to describe the additional primitives that enable the execution in SMP mode. Finally, the leveraged testing tools and programs are presented, before briefly showcasing the achieved performance of the SMP port compared to an AMP configuration.

Chapter 2

Background

This chapter provides the foundation for the port implementation described in Chapter 3. Initially, an overview of the multiprocessing techniques considered in this document is included, followed by a brief comparison. Afterwards, both the RTOS and the SoC are presented in depth so that the reader can grasp more easily the porting process. In this sense, the architecture of the SoC, its components, and the software that powers them are detailed more than FreeRTOS itself. Indeed, while the internal functioning of the RTOS constrains the structure of the low-level code, there are often several ways to implement a port function given a specific hardware. Nevertheless, in case a non-trivial concept was inadvertently overlooked, FreeRTOS provides complete documentation [2]. The chapter concludes with a description of the development environment.

2.1 Multiprocessing

The term Multiprocessing commonly refers to the ability of a system to simultaneously use more than one Central Processing Unit (CPU) to perform the required computation. The definition can be further adapted to a specific context, since it is functional to describe the possible interactions between the different CU of a system. Multiple CPUs on many interconnected printed circuit boards, multiple cores of a single CPU, and different processors with specialized architectures (e.g., graphic and neural processing units) on an SoC are only some of the possible configurations that can be observed in modern devices. For this thesis, a CU is identified either as a core or a processor, and MP is defined as the ability of a CU to coordinate with other CUs to fulfill the function of a system. Different MP solutions can be implemented on a single device depending on the symmetry of the CUs. Some of the most common approaches are described below, other than being pictured in Figure 2.1:

- **Asymmetric Multiprocessing**

  In an AMP system, the CUs are typically heterogeneous at both the hardware and software levels. The processors have distinct architectures and mostly separate memory address spaces while the programs they run can be completely different (e.g., bare-metal code and an Operating System). The CUs communicate with each other through shared memories and interrupt-driven messaging mechanisms (e.g., hardware mailbox), which can be abstracted by using proper frameworks (e.g., OpenAMP) or ad hoc software solutions.

  A device that integrates an MCU to deal with sensors and other peripherals and an MPU to run a lightweight Operating System (OS) is a classic AMP system. The strength of such designs lies in their flexibility and adaptability to a wide range of requirements. In the previous example, the MPU-MCU pairing may be a low-power solution for an application that needs to fetch and label data before transmitting it to the cloud for further processing.

- **Symmetric Multiprocessing**

  An SMP system is equipped with CUs that are homogeneous in terms of hardware and software. The architectural uniformity ensures that the entire set of CUs can be managed by a single instance of an OS and most of the address space is shared, or at least coherent, to ease the interaction of the CUs. When it comes to software design, it is still possible to assign additional tasks to each CU, but, compared to the whole application code, the exclusive code a CU has to execute is generally a small fraction. This approach ensures that the OS can effectively distribute the system's workload on all the provided units. However, the fact that CUs share most of the data structures is also a disadvantage because the skills and effort required to produce functionally correct SMP software are much higher than the ones needed to code an application for a single CU.

  A multi-core MPU is the most obvious example of SMP system, even though devices equipped with multiple instances of a single-core processor can be found in the embedded domain.

SMP can potentially achieve higher performance than AMP because the homogeneous architecture enables designers to tightly couple the CUs [3]. Faster inter-core communication mechanisms and workload distribution across the CUs are only some of the advantages SMP offers. However, there are some significant drawbacks, as the additional complexity needed to realize such systems must be addressed in hardware and software. The architecture becomes more elaborate in terms of area and power consumption, due to the CUs' duplication and the supplementary units that provide the actual SMP support. For example, ARM Cortex-A MPUs integrate a module called Snoop Control Unit (SCU) that maintains the L1-cache coherence for the cores participating in this process. From a software perspective, especially when the hardware support is not enabled, the application writer has numerous restrictions that would not be present with a single CU. In general, the higher the number of CUs that interact with the same data structures, the higher the sections of code that need proper synchronization mechanisms to preserve the application logic. Furthermore, as hinted before, if SMP is not properly supported at the hardware layer, the software may need to deal with several other hardware-related issues (e.g., memory coherency). Because of these problems, AMP has been historically more popular than SMP in ESs. AMP also has the advantage of better fitting the needs of many embedded applications, since it allows to assign a separate part of the application to each part of the system. This characteristic can be beneficial in mixed-criticality systems as every domain can be certified separately.

Despite their advantages and disadvantages, AMP and SMP are rarely interchangeable. Instead, they are usually employed to face different problems. For this reason, in some systems, selecting one of the two approaches would not benefit the application as much as combining them in a hybrid form of MP, which can be referred to as hybrid multiprocessing. This choice is particularly convenient when clusters of symmetric processors can be managed asymmetrically so that each cluster can be specialized in a task (AMP) and completely exploit the resources of the processors within the cluster to efficiently execute the task (SMP).

2.2 Zynq-7000 SoC

The Zynq-7000 SoC is a family of devices developed by Xilinx, now part of AMD, offering hardware and software programmability on a single chip. These features are achieved through the integration of a Processing System (PS), which embeds an ARM-based processor, and a Programmable Logic (PL), which provides the functionality of a Field Programmable Gate Array (FPGA). The heterogeneous architecture is especially suited for applications that need a compromise between hardware acceleration and the flexibility of a powerful processor.

SoCs in this family are divided into two series: Zynq-7000 and Zynq-7000S. This thesis focuses on the former architecture, as all the PSs feature an Application Processing Unit (APU) equipped with a dual-core ARM CA9 processor. Zynq-7000S SoCs are not considered, since the PS embeds a single-core CA9. Aside from the PS, the hardware configuration of both the board and the PL is not relevant to the thesis' objectives, so it will be presented in detail during the dissertation if deemed necessary. For reference, an internal view of the PS is depicted in Figure 2.2.

The cores in the APU are connected in an MP configuration, sharing a unified

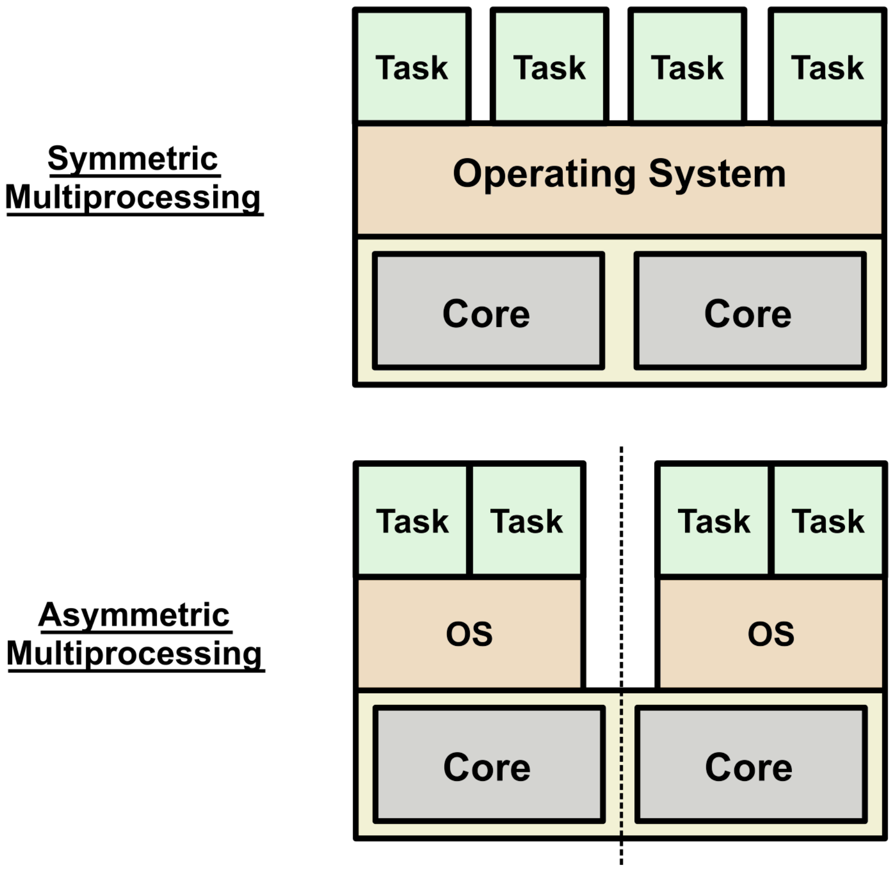

Figure 2.1: Comparison of SMP and AMP systems

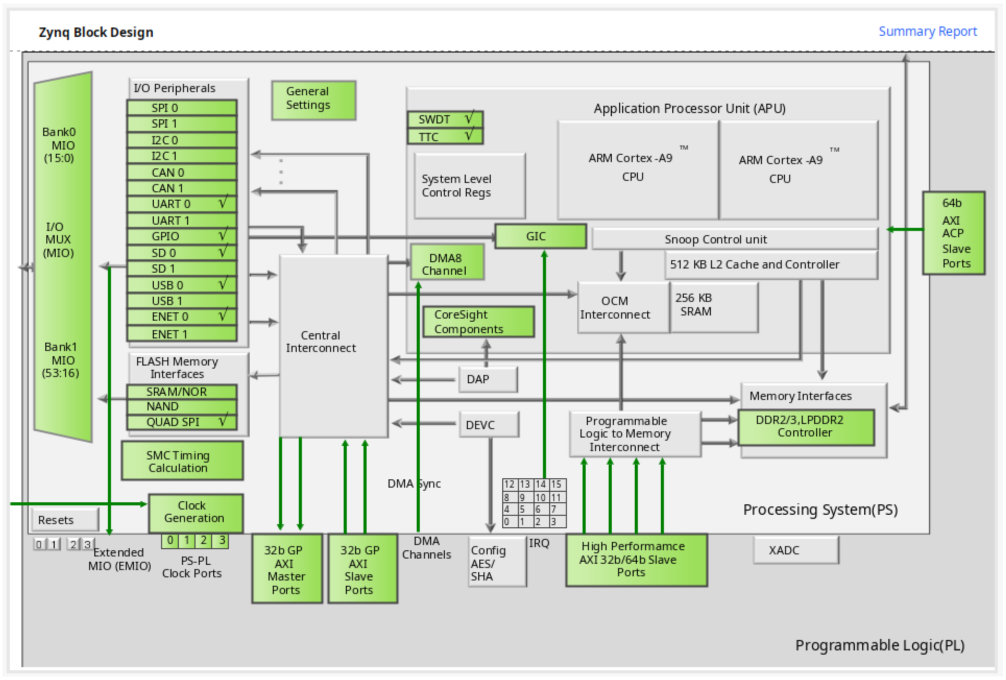

Figure 2.2: Zynq-7000 SoC Processing System internal view (Vivado)

512 kB L2-cache and an SCU, which is responsible for maintaining the L1-cache coherency. Despite the architectural symmetry, the CUs can assume separate roles, as reported in Section 2.1. For example, the system boot is usually executed on one of the available cores (i.e., the primary core). In contrast, the others (i.e., the secondary cores) are started from the primary core after the system has been initialized. Therefore, for clarity, the two cores will hereafter be referred to as CPU0 and CPU1.

2.2.1 PS Software Boot Process

The complete Zynq-7000 SoC boot is a complex process that involves several hardware and software passages [4, ch. 6]. However, the stages required to boot the PS are sufficient to understand the arguments exposed in the following chapters.

The Zynq-7000 SoC PS boot [5, ch. 3] is a process divided into a maximum of three steps: stage-0 boot, First Stage Boot Loader (FSBL), and, optionally, second stage boot loader. The stage-0 code stored in the on-chip ROM is the first software to be executed on the PS and it also known as BootROM code. Among its tasks, this program sets up the APU, configures the system peripherals according to the selected boot mode (e.g., JTAG, Quad-SPI, and SD memory card), and moves the FSBL from the boot device to the on-chip memory. The FSBL continues the PS' setup and eventually programs the PL before loading a second stage boot loader (e.g., U-Boot) or a bare-metal application in Double Data Rate (DDR) memory. The FSBL that is executed after the BootROM phase can be entirely customized depending on the user's needs, but AMD provides an example, which is enough for most applications.

2.2.2 Application Processing Unit Modules

Besides the CA9 processor, the APU includes several hardware units that provide core functionalities of the system. Among them, two modules are particularly relevant for the work presented in this thesis: Generic Interrupt Controller (GIC) and Triple Timer Counters (TTCs).

A TTC comprises three independent 16-bit timers, each equipped with a 16-bit prescaler, whose input clock can be selected from the available sources. Every counter can generate an output waveform (e.g., Pulse Width Modulation) and a periodic interrupt. Such action is triggered by an overflow, either on the entire counting range, a shorter interval, or a match with a configurable value. The APU features two TTCs for a total of six timers.

The interrupt controller incorporated in the APU is a PrimeCell GIC (PL390), based on the non-vectored ARM GIC Architecture version 1.0. The GIC is logically divided into a global Distributor and a series of local CPU interfaces.

The Distributor centralizes the interrupt sources and delivers the highest-priority pending interrupt to each interface. In contrast, the CPU interfaces allow cores to manage the interrupts routed through the GIC. When active, a CPU interface evaluates the associated core's priority mask and preemption policy to determine whether to signal the interrupt delivered by the Distributor. The interface is also used to acknowledge an interrupt and notify of its completion after it has been handled.

The GIC supports the configuration of security state, priority level, and target processor for an interrupt, as well as their activation and deactivation. Furthermore, the managed interrupts are classified in three groups: Software Generated Interrupts (SGIs), Private Peripheral Interrupts (PPIs), and Shared Peripheral Interrupts (SPIs). The GIC integrated in the Zynq-7000 SoC accepts signals coming from PL, I/O peripherals, and CA9 cores and routes the resulting interrupts to PL and CA9 cores.

On the Zynq-7000 SoC, each core has 16 private interrupt sources to generate an SGI, which can target itself and/or other cores, and some PPIs. Private peripheral sources are the CPU private timer and watchdogs, the global timer, and an IRQ and a FIQ, both coming from the PL. Moreover, since SPIs and PPIs are private, the Distributor registers associated with them are banked. In contrast, SPIs are a group of signals produced by several modules that the GIC delivers to CPU and PL, even though it does not manage reception and prioritization for the latter. An SPI can be assigned to multiple cores, but only one really handles it, and its sensitivity might also be configurable (e.g., rising-edge or high-level).

The Zynq-7000 SoC Technical Reference Manual (TMR) offers further information on both TTC [4, pp. 253-257] and GIC [4, ch. 7]. The GIC is also extensively described in the ARM manuals [6] [7].

2.3 ARM Cortex-A9

The ARM CA9 is a highly configurable processor optimized for power and performance, part of the Cortex-A series. The processor is implemented with the ARMv7-A architecture, but besides the ARMv7-A instruction set, it supports Thumb and Thumb-2 and can be extended with Floating Point (FP) and single instruction multiple data support. A single MPU can host a maximum of 4 cores, each provided with a Memory Management Unit (MMU); however, as already stated in Section 2.2, the CA9 processor embedded in the Zynq-7000 SoC integrates only two cores. The following subsections discuss significant CA9-related concepts, which are helpful to understand the work presented in this thesis.

2.3.1 Relevant features in the ARMv7-A architecture

The ARMv7-A architecture defines several features that enable software and hardware designers to create complex yet well-structured systems. The most significant concepts are briefly introduced in this subsection to facilitate the reading of Chapter 3.

Cache Maintenance Operations

In a memory hierarchy that employs cache memories, maintenance operations might be required to ensure that caches do not hold stale data due to changes in external levels of memory or activity in the MMU (e.g., modification of memory attributes). This function is provided through the "clean" and "invalidate" CP15 operations, which respectively write dirty lines to more external levels of memory and remove data from the cache. Nevertheless, these maintenance steps are not always necessary. For example, cleaning a cache line is only relevant for data caches that use a write-back policy.

Both clean and invalidate can be performed with cache set, way, or virtual address granularity. However, while set/way based operations target a specific level of cache, for virtual addresses two points are defined: the Point of Coherency (PoC) and the Point of Unification (PoU). The former is the point that allows all units that can access memory (e.g., direct memory access and cores) to have a coherent memory view. The PoU is the point where the data and instruction caches of a core can observe the same copy of a memory location. The concepts of PoC and PoU are equivalent for the CA9, as they both correspond to the Level 2 (L2) interface [6, ch. 8, pp. 19-20] because the processor does not incorporate an L2 cache. For more information on cache maintenance operations, refer to the ARM documentation [8, sec. B2.2.7] [6, ch. 8, pp. 17-21].

Processor Modes

The ARM architecture provides multiple processor modes to handle different scenarios that can occur during the operation of a system. These modes are classified as either Privileged or Unprivileged, and aside from User mode, all modes are privileged, so their ability to perform certain operations is not restricted. An essential aspect of processor modes is that some of the provided 16 32-bit general-purpose registers and the Current Program Status Register (CPSR) are shared across modes. Therefore, registers may need to be saved before being used. Nevertheless, some modes (e.g., FIQ) still have banked registers, meaning the software might access different physical locations depending on the active mode. Stack Pointer (SP) and Special Program Status Register (SPSR) are the only registers banked in all modes, except User and System mode, which share the same set of registers. Since some processor modes are associated to exception events, the SPSR is used by the processor to automatically save the CPSR of the mode that was active before the arrival of an exception, while banked SPs allow to allocate separate stacks for each mode. The topic becomes even more complex when additional architectural extensions, like the Virtualization Extensions, are introduced. These cases are beyond the scope of this thesis, but they can be explored further in the ARM manuals [8, sec. B1.3] [6, ch. 3].

Exception Handling

According to ARM's definition [8, sec. B1.8-B1.9] [6, ch. 11], an exception is a condition that disrupts the normal execution of a program due to an external event. When the processor receives such a signal, it immediately manages the request by executing the handler associated with the exception in the vector table. This step is preceded by other operations, which ensure the correct resumption of normal execution after the event [6, sec. 11.2]. Among the different types of exceptions, interrupts are particularly meaningful to understand some of the topics presented in this thesis. ARMv7-A divides interrupts into "IRQ" and "FIQ", so that, when triggered, all exceptions disable IRQ, but only FIQ and reset disable FIQ.

Since ESs often execute code in response to interrupts, the speed at which the processor reacts to external events is a crucial factor, referred to as interrupt latency. Consequently, other than the classic exception management, the software can define a reentrant interrupt handler to support nested interrupts [6, sec. 12.1]. This feature allows to prioritize interrupts, reducing latency for higher priority events.

Memory Barriers

Memory barriers are instructions that generate synchronization events to constrain the execution of memory operations that appear before and after the barrier. ARM provides three instructions of this kind to support its memory ordering model: Data Synchronization Barrier (DSB), Data Memory Barrier (DMB), and Instruction Synchronization Barrier (ISB). DSB and DMB can be applied to different "shareability domains" and only affect memory accesses generated by load/store instructions and cache maintenance operations. However, the CA9 support exclusively system-wide DSBs [9, sec. 7.5]. DMB ensures that any explicit memory access preceding it is observed before any subsequent memory access. DSB imposes the same conditions of DMB, but it also enforces the completion of memory operations that appear before the barrier. Finally, ISB flushes the pipeline, forcing it to freshly fetch the following instructions. This is especially useful when context-altering operations (e.g., translation lookaside buffer or branch predictor operations) before ISB must be visible to instructions executed after the barrier.

Additional information is reported in the ARM documentation [8, sec. A3.8.3] [6, sec. 10.2].

ARM Architecture Procedure Call Standard

As applications are composed of multiple code modules, often written in different languages, a standard set of rules is required to ensure interoperability in ARM executables. For ESs, these rules are collected under the ARM Embedded Application Binary Interface (EABI). However, since the comprehension of the next chapter does not need such extensive knowledge, the attention can be solely focused on the ARM Architecture Procedure Call Standard (AAPCS). This is a part of the EABI that defines conventions on registers and stack usage, which must be respected by compilers and routine calls to guarantee the correct execution of a program. Key concepts are the relation between stack organization and function calls and the classification of registers in three functional groups: argument or caller-saved, callee-saved, and special registers. In particular, the rules related to memory organization specify the layout of data in memory and argument registers for function parameters, and that 64-bit types must be aligned to 8-byte boundaries. More details are discussed in the ARM Cortex-A Series Programmer's Guide [6, ch. 15].

2.3.2 Memory Management Unit

An MMU is a hardware unit that acts as an interface between the processor and the memory system to manage memory accesses. This task is primarily fulfilled by denying illegal memory operations and translating the virtual addresses from the processor domain into physical addresses, which are used to access memory locations. For these reasons, the MMU can operate in conjunction with various levels of cache (e.g., Level 1 (L1) and L2), other than external memory. The MMU integrated in the CA9 is compatible with the Virtual Memory System Architecture version 7; therefore, it supports a variety of features, as described in the ARM Architecture Reference Manual (ARMv7-A) [8].

To understand the configuration of an MMU in an SMP system, examining how the address translation is performed is essential. However, the project developed in this thesis uses a 1:1 mapping, so there is no distinction between virtual and physical addresses. The process is based on hardware structures, called translation tables or page tables, which host a series of entries whose primary function is to transform a virtual address into a physical address. Attributes related to memory sections and access permissions are also encoded together. Each entry refers to a portion of the system's memory, but the actual size of such a block, also called page, depends on the specific architecture. This parameter is critical, since the correct sizing can drastically influence the performance of memory accesses. The CA9 MMU supports 4 kB, 64 kB, 1 MB, and 16 MB pages, as well as two-levels page tables.

Memory types and attributes are concepts strictly tied to the managed memory section of a memory map [8, sec. A3.5] [4, pp. 59-62]. Three mutually exclusive memory types exist: Normal, Device, and Strongly-ordered. Memory that stores data and programs (e.g., DRAM, SRAM, ROM, and flash) is generally marked as Normal. Differently, I/O peripherals are tagged as Device or Strongly-ordered memory because they need to comply with more restrictive access rules. The "shareability" and "cacheability" attributes also affect the memory access order of the types that define them. The former divides the system into different domains, each with specific features related to the predictability and coherence of the memory operations. In the Zynq-7000 SoC, the inner shareable domain ensures coherency between the CA9 L1 caches. In contrast, the outer shareable domain provides the exact property to CPU, L2 cache, and other peripherals (e.g., direct memory access) which may access a shared memory. Cacheability is somewhat linked to shareability, since it allows to control the coherency of observers that are not part of the shareability domain of a memory region. In the Zynq-7000 SoC, memory regions marked as inner cacheable can be fetched in both L1 and L2 cache, while the ones tagged as outer cacheable can reach only the L2 cache.

2.3.3 Exclusive Access

As explained in Section 2.1, SMP systems require a non-negligible software development effort. Hardware cannot always guarantee the functional correctness of application-level logic, which typically follows a hardware-independent design model. A concurrent access to the same resource by multiple CUs defines a region of code referred to as critical section and is a primary example of this concept.

The solution to such an issue is mutual exclusion. However, while in single-core systems mutual exclusion can often be achieved by disabling interrupts, in a multi-core device this is not possible because other CUs might still use the shared resource. Instead, these systems resort to hardware-software mechanisms named spinlocks.

ARMv7-A provides three basic instructions to implement mutual exclusion: LDREX (Load-Exclusive), STREX (Store-Exclusive), and CLREX (Clear-Exclusive). These primitives rely on the ability of exclusive access monitors to track the accessed memory addresses. Furthermore, the specific monitor used in a particular scenario depends on the shareability attribute that marks the tagged address. Non-shareable memory regions are only tracked with the local monitor of a core, while shareable sections are additionally checked by a global monitor, which also considers other observers' activity in the shareability domain. Monitors can assume either the "open access" or the "exclusive access" state and can transition between these states in response to exclusive access instructions. The CA9 includes a local monitor in each L1 cache [9, ch. 7, p. 8], whereas the presence of a global monitor is never explicitly mentioned. This is consistent with the possibility of integrating both local and global monitors into a single unit [8, ch. A3, p. 116]. Furthermore, the Zynq-7000 SoC provides two additional monitors in the DDR Controller [4, pp. 142-145].

LDREX performs a load and tags the address in the exclusive monitor, while STREX conditionally executes a store when the location is marked as exclusively monitored by the core. Therefore, LDREX and STREX must be used as a pair. CLREX clears the state of a monitor; thus, it can be employed to ensure that an STREX-LDREX pair does not depend on the result of previous exclusive accesses. Naturally, the behavior of these instructions and their variants is slightly different depending on the shareability attribute of the target memory. Moreover, the number of bits for a tag is implementation-defined, but always limited between 29 and 21 bits. The size of the tagged memory block, known as Exclusives Reservation Granule, is equal to 8 words, aligned on an 8-word boundary, for the CA9 [9, ch. 7, p. 8], and can be retrieved in the CP15 Cache Type Register (c0) [8, sec. B4.1.42] as $\log_2 \mathrm{ERG}$.

Further information on the topic can be retrieved from the ARM documentation [6, sec. 18.8] [8, sec. A3.4.1-A3.4.5].

2.3.4 Global and Private Timers

Hardware timers are essential components that enable designers to implement time-related functions in their applications. The CA9 integrates a private timer and watchdog block per core, in addition to a global timer accessible by all cores. The private timers and watchdogs are 32-bit decrementing counters capable of generating interrupts when the count reaches zero. The counting mode is set to single-shot or auto-reload, and both prescaler and initial value can be configured via software. In contrast, the global timer is a 64-bit incrementing counter, split into two separately accessed 32-bit registers, that can generate interrupts with an interval shorter than its counting period. Although the global timer is shared, the registers to configure this feature are banked, making the interrupt private to each core. All these modules are clocked by the PERIPHCLK signal, which corresponds to half the CPU frequency on the Zynq-7000 SoC [4, p. 245]. More details on the timers' registers can be found on the CA9 MPCore TMR [10, ch. 4].

2.4 FreeRTOS

FreeRTOS is a market-leading open-source RTOS developed by Real Time Engineers Ltd., now maintained by Amazon Web Services as part of the AWS FreeRTOS project [11]. It stands out from other competitors (e.g., ThreadX and Zephyr) thanks to its small memory footprint and simplicity without giving up the numerous features of the kernel. This is reflected by the variety of supported task scheduling strategies (preemptive, cooperative, and hybrid), inter-task communication (queues, task-notifications, event groups, stream, and message buffers), and synchronization primitives (mutexes and semaphores). Reliable time-sensitive applications can be implemented thanks to the deterministic real-time scheduler, the software timer Application Programming Interface (API), and the deferred interrupt handling, all while maintaining a low power consumption in tickless idle mode. The memory management is highly flexible, providing both static and dynamic approaches. This allows various design choices tailored to the specific application and target device.

FreeRTOS features a layered architecture, which separates hardware-dependent components, commonly referred to as port layer or port, from hardware-independent code. Combined with its well-structured design, it helps the kernel to be highly portable and support a broad range of architectures: ARM Cortex-M/A/R, RISC-V, and many others.

FreeRTOS is mainly present in Internet of things, industrial automation, and automotive domains where real-time performance and reliability are essential. Its adoption is backed up by many libraries, which extend the basic functions of the kernel (FreeRTOS-Plus) and allow a deep integration with the AWS cloud environment.

2.4.1 Symmetric Multiprocessing in FreeRTOS

Since 2003, FreeRTOS has been developed as a single-core RTOS. This was reasonable in the project's early days because ESs were not equipped with symmetric CUs. However, AMP still enabled to achieve a certain degree of MP in the developed systems. Following the design improvements in the embedded domain, FreeRTOS has been upgraded to officially support SMP in its kernel. Therefore, starting from version 11.0.0, along with the options already available, the FreeRTOS configuration file `FreeRTOSConfig.h` also provides SMP-related macros: `configNUMBER_OF_CORES`, `configRUN_MULTIPLE_PRIORITIES`, `configUSE_CORE_AFFINITY`, etc.

The design of the new features is coherent with the description of SMP provided in Section 2.1, but it also integrates some additional options, as hinted by the names of the listed macros. In fact, while the traditional definition of SMP implies that each core can execute every task, FreeRTOS can be configured to bind some tasks to specific cores or, if preemptive scheduling is active, to prevent tasks with different priorities from running on multiple cores simultaneously. Furthermore, the SMP kernel enables individual tasks to be configured either in preemptive or cooperative mode, leading to hybrid forms of MP even within a symmetric set of CUs. Although the listed features introduce some variations to the original concept of SMP, they are not uncommon in modern OSs. In the context of ESs, they are instrumental when an application is migrated from a single-core to a multi-core environment because they allow the restriction of the software-level symmetry.

The FreeRTOS ports that have already been upgraded to integrate SMP are located in the `portable/ThirdParty/` directory of the FreeRTOS-Kernel repository [12]. However, these ports are not always developed by the FreeRTOS team, as partners and community members can also undergo the contribution process. Among the available ports, the most notable are the XCORE AI and Raspberry Pi Pico (FreeRTOS Team Supported), the Texas Instruments Cortex-A53 (Partner Supported), and the Zynq UltraScale+ MPSoC Cortex-A53 (Community Supported).

Despite the popularity of FreeRTOS and the significant potential of using SMP in embedded devices that support it, the number of ports that implement this form of MP remains limited. Many factors contribute to the problem, such as limited documentation, the required development effort, and the availability of more mature alternatives (e.g., Zephyr).

About the documentation, aside from the numerous individual posts on the FreeRTOS forum, the two primary sources of information are the XMOS SMP design guide [13] and a popular thread discussing the Cortex-A53 SMP port [14]. It is important to note that these resources contain conflicting guidelines due to the evolution of some constraints over time. The forum post should be trusted in such cases, as it is more recent than the design document.

2.4.2 FreeRTOS on ARM Cortex-A9

The FreeRTOS documentation provides a dedicated page to illustrate the characteristics of the CA9 port for devices that incorporate a GIC [15]. The most relevant features of this port are the hardware FP support and the implementation of the interrupt nesting model.

The context-switch logic can be configured to save the FP registers for all tasks or only when explicitly enabled by calling `vPortTaskUsesFPU()` within a task. Furthermore, FP operations cannot be used by default inside interrupt handlers. This functionality must be manually set up by modifying the FreeRTOS handler associated with the IRQ exception in the vector table (`FreeRTOS_IRQ_Handler`), so that `vApplicationFPUSafeIRQHandler` is called instead of `vApplicationIRQHandler`. Additionally, care must be taken to avoid the usage of functions such as `memcpy()` and `memset()`, as well as any FreeRTOS API that may rely on them (e.g., `xQueueSend()`), inside both tasks and interrupt handlers. In fact, compilers may exploit FP registers to optimize these functions. The kernel also enables the definition of a maximum priority that allows interrupts to call faster interrupt-safe APIs, whose name ends with `fromISR`.

The implementation of nested interrupt handling is analyzed in Section B.6, while further details on the configuration and customization of the CA9 port can be found in the FreeRTOS documentation [15].

2.5 Development Tools

The development of modern applications for ESs often requires the assistance of an environment that automates and simplifies some steps of the design flow. Being equipped with both a PS and a PL, Xilinx devices are no exception to this rule, as they require a hardware-software co-design platform. Moreover, in case the application is designed to run with the support of an OS, debugging it becomes challenging, especially if the software is executed on multiple CUs. In this scenario, an additional OS-aware trace tool may be needed to capture the application's behavior at runtime.

2.5.1 Hardware-Software Design Platforms

Vitis and Vivado are development platforms that support the design and implementation of hardware and software designs on Xilinx FPGAs and SoCs.

Vivado offers a comprehensive tool suite covering the entire hardware development workflow. The design begins with the description of a circuit using an Hardware Description Language (HDL), such as Verilog and VHDL, or by connecting various intellectual properties and custom blocks, which Vivado can group into an HDL wrapper. The flow proceeds through synthesis, place, and route, which can be automated to streamline the design iterations. In the end, a bitstream is generated before programming the target FPGA or exporting the generated hardware.

Vitis is a software platform that provides end users with several tools to integrate their designs on Xilinx devices. To start building an application, a target system must be selected to create a Board Support Package (BSP). Vitis comes with various predefined devices, but it is also possible to create and import one from Vivado. An application can then be developed in bare metal or on top of an OS, as both FreeRTOS and Linux are supported. The designed software can finally be cross-compiled, using one of the preinstalled toolchains, and debugged, resorting to GDB or the Eclipse TCF Agent. Vitis also handles deployment by loading the produced binary on the PS and programming the PL with a bitstream.

Figure 2.3 visually represents the aforementioned co-design flow.

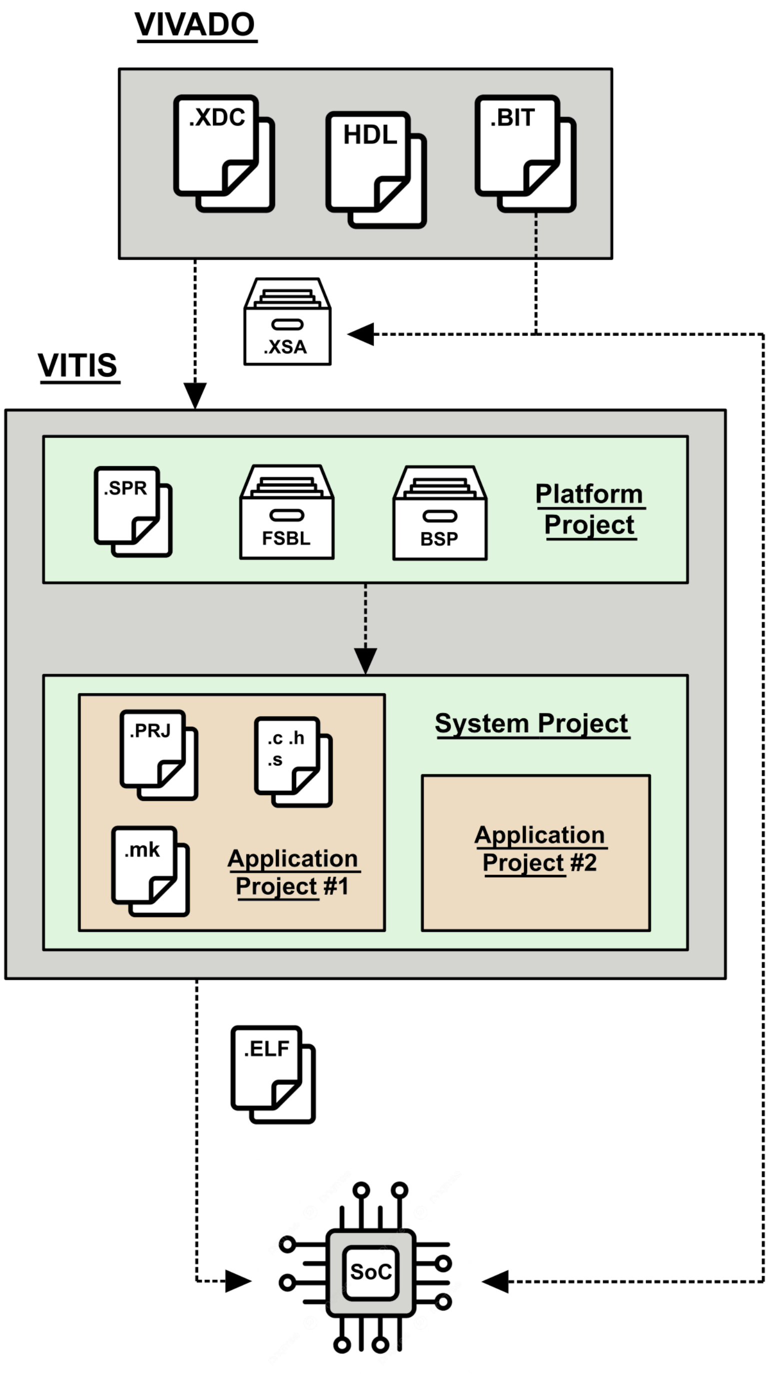

Figure 2.3: Hardware-software design flow

2.5.2 Xilinx Software Libraries

Most of the software components that power the design experience offered by Vitis are sourced from the Xilinx "embeddedsw" software library [16]. This open-source repository contains a collection of libraries and drivers that provide essential code to build BSPs and ease the interaction with various hardware modules.

The BSP configuration changes according to the available hardware and the abstraction level of the designed application. When the latter aspect is considered, the most relevant packages for the scope of this work are the "standalone" and "FreeRTOS" BSPs. The standalone BSP is a low-level software library exploited in bare-metal designs to manage the PS and its hardware modules (e.g., caches and timers). On the other hand, the latest version of the Xilinx FreeRTOS BSP includes both the APIs of the standalone BSP and the features of release 10.6.1 of the FreeRTOS kernel [12, tag `V10.6.1`], adequately adapted to the MPUs embedded in the supplied SoCs.

The drivers included in the BSPs facilitate the interaction between the application code and the modules in the PS. Obviously, each hardware unit is used to an extent that depends on the characteristics of system and application. For this reason, some modules have more optimized APIs (e.g., GIC and private timer) while others may not even need a driver (e.g.,). Additionally, as explained in Section 2.1, despite the presence of symmetric CUs, ESs are frequently configured in single-core or AMP. This often leads designers to ignore issues strictly related to SMP during the engineering of the driver.

The FreeRTOS portable code for the Zynq-7000 SoC uses substantially two drivers: `scugic` and `scutimer`. The latter supports the CA9 private timers, together with its features (e.g., hardware interrupts), whereas the former provides an interface to interact with the GIC. Although it is not present in the hardware-dependent layer, the `ttcps` driver is also relevant in this thesis, as it has been employed in the test demo to control the TTC module.

ScuGic driver

This subsection details the design of the `scugic` driver, as a solid knowledge of such library is essential to understand the design choices described in Subsection 3.4.2.

The `scugic` library is based on three main data structures, declared in `xscugic.h`: `XScuGic`, `XScuGic_Config` and `XScuGic_VectorTableEntry`. Each structure contains the preceding one as a member in the listed order. An `XScuGic_Config` instance stores information on the device managed by the driver. More precisely, it holds the ID assigned to the GIC in the BSP, the base address of Distributor and CPU interface, and an array of `XScuGic_VectorTableEntry`, named `HandlerTable`, which records the key-value pairs (Interrupt ID, Handler Address).

The library allocates a single XScuGic_Config variable in xscugic_g.c, which is referenced through a pointer whenever the GIC configuration is retrieved with a call to XScuGic_LookupConfig(). The execution of this preliminary step is essential, as the function XScuGic_CfgInitialize() assigns the XScuGic_Config* pointer to a XScuGic instance during its initialization. Once configured, each XScuGic variable can be used by a core as an interface to interact with the selected GIC through the scugic driver.

The library also includes logic to handle multi-core scenarios, since it defines the internal variable CpuId to identify the target core and guards the GIC shared registers, which are part of the Distributor, using the spinlock API contained in the standalone BSP.

Further information can be found in the library's source code inside the directory XilinxProcessorIPLib/drivers/scugic of the embeddedsw repository [16].

Xilinx spinlock API

The standalone BSP provides a simple spinlock for CA9 processors to deal with critical sections in developed applications. However, the implementation is limited to a single non-recursive spinlock, making it suitable primarily for Xilinx APIs that can leverage it and for elemental bare-metal designs.

The library declares the internal variables Xil_Spinlock_Addr, which stores the address of the spinlock, and Xil_Spinlock_Flag_Addr, which holds the address of the spinlock's status. The flag is set to either XIL_SPINLOCK_ENABLED or 0, depending on whether the lock has been initialized. Such action is performed by calling Xil_InitializeSpinLock(), while to release a spinlock, the function Xil_ReleaseSpinLock() is used. An important detail is that the lock and its status can reside within the same 32-byte memory region because the flag is not accessed using the exclusive operations described in Subsection 2.3.3.

After being initialized, the spinlock can be claimed with Xil_SpinLock() and unlocked with Xil_SpinUnlock(). As the lock does not support recursive acquisitions, it cannot be claimed multiple times by its owner; hence, a Xil_SpinLock() must always be followed by a Xil_SpinUnlock().

For a more detailed description of the API's usage, refer to the files lib/bsp/standalone/src/arm/common/xil_spinlock.* provided with the Xilinx embeddedsw library [16].

2.5.3 Trace and Debug

Vitis provides native OS-aware debug options, but only Linux is entirely supported. Nevertheless, FreeRTOS can be debugged by other means, as it integrates numerous trace macros in the kernel that the user can selectively implement to extract relevant runtime information (e.g., context-switch actors). Furthermore, Percepio Tracealyzer and its free version, Percepio View, can exploit these trace macros to collect statistics during the execution of a program and analyze them through a practical Graphical User Interface (GUI). More specifically, trace data is stored inside a static buffer in either "Snapshot" or "Streaming" mode. Unfortunately, Percepio View offers only a subset of the features available in Tracealyzer. Despite its limitations, for the aims of this thesis, Percepio View has been revealed to be a valuable tool for achieving observability in a system running FreeRTOS in SMP mode.

Despite the availability of trace tools, debugging must also be supported by a well-designed set of tests to be effective. FreeRTOS provides a variety of predefined demos, each targeting a specific device, along with a template to create new demos, respectively located under `Demo/` and `Demo/ThirdParty/Template` in the FreeRTOS Github repository [17].

Chapter 3

Implementation

This chapter details the process of porting the FreeRTOS kernel's SMP support to the Zynq-7000 SoC. The discussion begins with the setup of hardware and software environments, followed by the analysis and update of the existing port for Zynq-7000 SoC from version 10.6.1 to version 11.1.0 of the kernel. After this initial phase, which focuses purely on the single-core functions, the implementation of the SMP port is presented. More specifically, the description starts by explaining the dual-core boot process of the Zynq-7000 SoC, immediately followed by the multi-core configuration of FreeRTOS, to provide a complete picture of the CPU1's bring-up procedure. Ultimately, the main changes introduced in the port are illustrated, before showing the debug tools and the demo application exploited to validate the final implementation.

It is essential to underline that although this chapter includes some configuration steps when the discussion requires them, its objective is to explain the design choices behind the implementation of the SMP port and its test demo. Detailed guides to achieve functioning projects based on the Zynq-7000 SMP port [18] and the related test demo [19] can be found in the `README.md` files published on the "FreeRTOS Community Supported" repositories.

3.1 Development Environment

The hardware provided for developing the FreeRTOS SMP port is a TUL PYNQ-Z2, which integrates a Zynq-7020 SoC. As discussed in Section 2.2, this chip is part of the Zynq-7000 series and is equipped with a dual-core ARM CA9.

As for the development platforms, general concepts about the tools adopted to manage the hardware and software layers, which serve as the foundation of the entire project, have been introduced in Section 2.5. The specific version used for both Vitis and Vivado is 2021.1, as it is more lightweight than newer releases but still provides the needed development features. However, concerning Vitis, the "Classic" edition does not integrate the System Device Tree (SDT) flow as the "Unified" one. Consequently, the parts that support the SDT flow have not been upgraded with the rest of the single-core port and are thus overlooked in the following discussion. Furthermore, Vitis uses a version of the embeddedsw library aligned to its release. This introduces the risk of working with outdated BSPs and drivers. The release tagged `xilinx_v2024.1` [16] was preferred to address this issue. Compared to the library bundled with Vitis 2021.1, the standalone BSP contains more useful drivers to interact with the Zynq-7000 SoC and fixes for specific CA9 operations.

Creating a hardware design for the PYNQ-Z2 in Vivado is thoroughly documented online [20]. It is therefore not presented here, as it does not influence any of the topics discussed in this thesis. For reference, Figure 3.1 shows the minimal set of hardware modules required to work with the PS. The generation of Platform Project (PP) and Application Project (AP) in Vitis is omitted as well, as this process is documented both in the `README.md` file provided with the SMP port [18] and in the official Vitis manual [21, pp. 24–69]. Nevertheless, it should be underlined that FreeRTOS is included as an external resource in an AP based on the standalone BSP. This choice stems from the impossibility of integrating BSPs with custom names and structures into the Vitis GUI without editing its back-end scripts. In conclusion, the cross-compilation is handled using the `arm-none-eabi` toolchain shipped with Vitis, so, unless explicitly stated, it is implied that the referenced FreeRTOS ports are compiled with version 10.2.0 of the GNU compiler (GCC) [22].

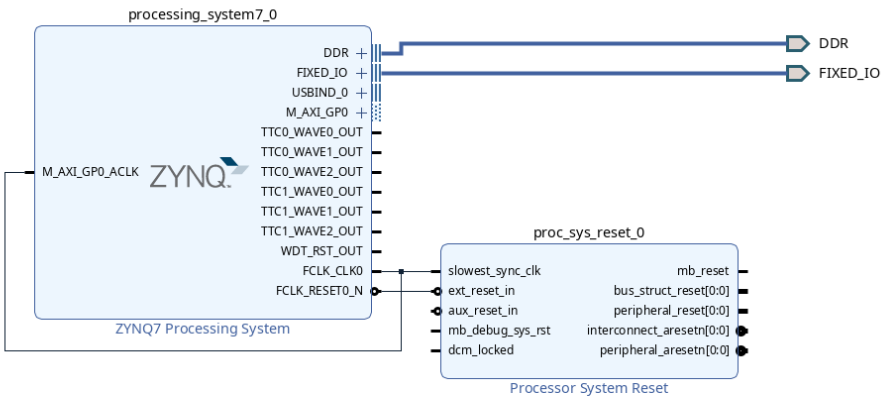

Figure 3.1: Block Design view of the smallest design that includes a Processing System (Vivado)

3.2 FreeRTOS Port Update

Updating the official FreeRTOS BSP for Zynq-7000 SoCs requires modifying the hardware-dependent layer. Considering that, as stated in Subsection 2.5.2, the latest available release of the Xilinx FreeRTOS BSP is based on version 10.6.1 of the kernel [12, tag `V10.6.1`], the official CA9 portable files of such version have been compared with the portable layer of the FreeRTOS BSP to discover potential differences. The analysis led to a better understanding of the changes performed on the basic CA9 port to adapt it to the Zynq-7000 SoC and on the degree of integration the BSP has with the Vitis back-end. The most notable aspects are listed below, grouped by file:

- `port.c` implements the port functions that interface FreeRTOS with the host hardware. Compared to the CA9 template in the original kernel, the management of the runtime statistics, enabled by setting `configGENERATE_-RUN_TIME_STATS` to 1, is embedded in the port layer and causes the system tick to be generated ten times faster. Furthermore, additional functions have been defined to ease the interrupt management in the application code, preventing the direct usage of `xInterruptController`, the driver's instance used to configure the GIC. Finally, `prvTaskExitError`, assigned to `configTASK_-RETURN_ADDRESS` inside `FreeRTOSConfig.h`, has been redefined to terminate a task instead of triggering an assertion when it attempts to return.

- `portmacro.h` defines macros employed across the entire portable layer. In the FreeRTOS BSP, they have been moved in `FreeRTOSConfig.h` to better integrate the FreeRTOS configuration into the Vitis GUI.

The BSP is also supplied with two auxiliary files, which ease the setup steps needed by FreeRTOS to correctly interact with both the Zynq-7000 SoC and the user-defined application:

- `portZynq7000.c` provides the application interrupt handler `vApplication-IRQHandler`, the functions used to set up the interrupt that periodically invokes the scheduler, referred to as "system tick" or "SysTick", and a weak definition of the FreeRTOS hooks to prevent possible linker errors.

- `port_asm_vectors.S` overrides the definition of the CA9 vector table and of the exception handlers provided by the standalone BSP file `asm_vectors.S`. Even though `_vector_table` is defined multiple times, the linker does not produce any error and resolves the symbol to the one in `port_asm_vectors.S`. Moreover, both files are linked inside the executable, as the handlers in `port_asm_vectors.S` wrap the ones in `asm_vectors.S`. The function `vPort-InstallFreeRTOSVectorTable` is still present in `port_asm_vectors.S` to set the Vector Base Address Register to the correct address, but it is not used in the current port.

The FreeRTOS configuration file `FreeRTOSConfig.h` has been analyzed alongside the port layer because, as explained before, its generation is deeply integrated in the Vitis environment. Unfortunately, FreeRTOS 10.6.1 did not include a template to perform a proper comparison. Moreover, the back-end script that manages `FreeRTOSConfig.h` selectively includes definitions according to the features enabled in the BSP configuration. Therefore, the template located under `examples/template_configuration` in the FreeRTOS-Kernel repository [12, tag V11.1.0] has been preferred over the configuration file provided in the FreeRTOS BSP as a starting point for future applications.

Updating the port to version 11.1.0 did not require any particular effort, since the basic functions needed to execute FreeRTOS in single-core remained consistent across the two releases. The only significant contributions have been the relocation of some macros from the BSP `FreeRTOSConfig.h` to `portmacro.h` and the addition of `vApplicationDaemonTaskStartupHook` inside `portZynq7000.c`. The SoC-specific sections of `FreeRTOSConfig.h` have also been added at the end of the template.

3.3 Secondary Core Bring-up Procedure

This section presents the steps required to boot the dual-core CA9 embedded in the Zynq-7000 SoC. The parts concerning the Xilinx BSP and FreeRTOS are discussed separately to improve the clarity.

3.3.1 Dual-Core Boot on the Zynq-7000 SoC

The PS's boot process, presented in broad terms in Subsection 2.2.1, needs to be discussed in more detail, since understanding the role of each core is essential to achieve a multi-core boot.

In a dual-core Zynq-7000 SoC, the BootROM code begins at address 0x0 on both cores [4, pp. 151-152]. Nevertheless, while CPU0 performs the boot procedure, CPU1 parks itself in a low-power state using the Wait For Event (WFE) instruction. Therefore, to start executing code on CPU1 after the stage-0 boot, it is necessary to write the application's entry point at address 0xFFFFFFF0 and awaken the core with a Send Event (SEV) instruction [4, p. 160]. As a safety measure, even if a SEV is mistakenly issued before the content of 0xFFFFFFF0 is overwritten, CPU1 remains blocked in the same section of code. Section B.2 provides more details in this regard.

After the FSBL hands off control to the bare-metal application, the standalone BSP code in `boot.S` is executed by the core domain selected during the creation of the AP. For instance, choosing CPU1 as the AP domain leaves CPU0 in a low-power state. The second-stage boot sets up numerous modules of the APU (e.g., SCU, cache, MMU, and processor registers) to initialize the primary core. This procedure is outlined in the Zynq-7000 SoC TMR [4, p. 104] and in the ARM Cortex-A Series Programmer's Guide [6, ch. 13, p. 2]. From now on, APs will have CPU0 as a domain, unless specified otherwise.

The primary core's setup sequence in `boot.S` contains sections of code that ease the configuration of the APU in SMP, even though, by default, the standalone BSP can exploit only one core. In particular, the primary core is set in SMP mode and the MMU can be configured to coherently access memory with multiple symmetric cores by defining the preprocessor directive `SHAREABLE_DDR`.

To achieve coherency at the core level, the developed SMP system must properly set the DDR memory's attributes in the MMU, since it is the only external memory accessed by both cores. As hinted in Subsection 2.3.2, this can be done by tagging as "shareable" the page table entries that perform address translation for the DDR. Failing to do so forces to manage coherency with explicit software operations [6, ch. 10, pp. 4-5]. Enabling the S bit of an MMU descriptor is a necessary step [4, ch. 3, p. 74], but it is not enough to achieve memory coherency, as the cores need to actively participate in the process by informing the SCU. For this reason, they have to set the SMP bit in the Auxiliary Control Register [9, ch. 4, pp. 27-29]. Listings B.3 and B.4 show the relevant code in `boot.S`, while Section B.3 further analyzes the DDR configuration inside the MMU.

At the end of the bare-metal boot, the execution proceeds with the C startup code in `xil-crt0.s`, where the application is finally started. The FreeRTOS documentation on CA9-based systems [15] suggests calling the `main()` function in supervisor mode, but the processor mode is set to system mode.

The boot process provided by the standalone BSP concludes with CPU1 stuck in WFE and, by default, the code to manage the wake-up mechanism of the secondary core is not implemented in bare metal. Using the FreeRTOS BSP, the result would be the same, since, as described in Subsection 2.5.2, the library is based on the standalone BSP. It is possible to unlock a domain that configures the CA9 APU in SMP mode without user intervention, but it is required to select the Linux OS during the creation of a PP.

The only way to wake up CPU1 with the adopted AP structure is to follow the steps mentioned earlier in this subsection. However, since the FreeRTOS scheduler must be started inside `xPortStartScheduler()` on all the cores that run the application [13, p. 2], it does not make sense to wake up CPU1 in any of the BSP files. This consideration aligns with a key decision that will influence many design choices: to modify and exploit the BSP code as little as possible to minimize the effort required to execute the port on future BSP releases. The new port file `smpPortZynq7000.S` implements the process to boot CPU1 along with other functions required to operate in SMP.

3.3.2 Secondary Core Setup in FreeRTOS

The FreeRTOS kernel provides `vTaskStartScheduler()` to start the scheduler after configuring the application inside the `main()` function. This API initializes several kernel objects (e.g., idle task) and ultimately calls `xPortStartScheduler()`, which performs architecture-specific steps before restoring the context of the first task. However, as explained in Section 2.1, when SMP is adopted, the symmetry that characterizes this MP paradigm does not inherently apply to the entire kernel structure. Therefore, in FreeRTOS, the setup of the primary core differs from that of the secondary cores, as observed in Subsection 3.4.1.

Nevertheless, in SMP mode, `xPortStartScheduler()` must start the scheduler on all the available cores. In the Zynq-7000 SoC SMP port, the wake-up process of the secondary core begins with a call to `vPortLaunchSecondaryCore`, whose code is reported in Listing 3.1 for clarity. In this function, CPU0 stores the entry point of CPU1 at address 0xFFFFFFF0 and then manually cleans the dirty line of its L1 data cache to prevent the presence of stale data in CPU1's L1 data cache. As a result, since DDR addresses are marked with a write-back cache policy and CPU1 did not take part in hardware coherency yet, the memory views of the two cores differ. The cache line is cleaned by explicitly writing to the CP15 because the DCCMVAC pseudo-instruction [8, ch. B3, p. 1491] is not supported by the assembler. Moreover, as stated in Subsection 2.3.1, PoC and PoU are equivalent for the CA9, since the cache maintenance operations end in L2 cache or, in case it was inactive, in main memory. The instruction is guarded with two DSBs; the first makes sure that all prior operations are complete before proceeding, while the second forces the line clean to conclude before the execution of SEV. No other barrier is required in this case, as DMB only constraints the ordering of memory instructions without guaranteeing their completion [23], and ISB does not affect the visibility of data [8, ch. A3, pp. 148-152].

In the second part of `vPortLaunchSecondaryCore`, CPU0 continuously polls the shared variable `ulSecondaryCoreAwakeConst` to determine when it can stop issuing SEV instructions. The variable is updated once CPU1 has been completely configured, so the primary core does not start executing the application before the secondary core is ready. Alternative designs for the code are possible, as explained in Section B.4.

The entry point of CPU1 is the address associated with the symbol `_boot_core1`. This label is defined in `smpPortZynq7000.S` and marks the beginning of a section of code that replicates the boot of the primary core in `boot.S`, following the ARM's guidelines [24, ch. 5, pp. 10-11]. Naturally, the initialization procedures for the SCU and L2 cache are skipped, as they are shared resources. Another substantial

```
.type vPortLaunchSecondaryCore, %function
vPortLaunchSecondaryCore:
  /* Write the start address of CPU1 */
  mov  r0, #CORE1_START_VECTOR
  ldr r1, =_boot_core1
  str r1, [r0]
  dsb
  mcr p15, 0, r0, c7, c10, 1 /* Clean address to PoC */
  dsb

  /* Wake up CPU1 and wait for it to be ready */
  ldr r2, ulSecondaryCoreAwakeConst
pollForCore:
  sev
  ldr r1, [r2]
  cmp r1, #0
  beq pollForCore

  bx lr
```

Listing 3.1: CPU1 wake-up process

distinction can be observed in the definition of the labels that identify the processor mode stacks, as they are different from the ones used in `boot.S`. This is necessary because both cores operate concurrently, so the stack would be corrupted if they happened to be in the same processor mode simultaneously. For this reason, the linker script `lscript.ld`, which is automatically generated during the creation of an AP, has been modified to allocate `.stack1`. This additional memory section hosts the processor modes' stack of CPU1. In the entire project, this is the only scenario in which the user is obliged to substitute or modify a BSP file.

After the initial boot phase, CPU1 must be properly set up to work under FreeRTOS. Therefore, before signaling to CPU0 that the core is ready to execute the application, CPU1 configures the GIC interface by jumping to `vPortInit-CoreInterruptController()`. This function is defined in `portZynq7000.c` and, other than initializing the `scugic` driver instance of CPU1, links `prvYieldCore-Handler()` to the SGI with number `portSGI_ID`. As explained in Subsection 2.2.2, SGIs allow the cores connected to the GIC to communicate with each other. In the FreeRTOS SMP port, this feature is exploited by defining the macro `port-YIELD_CORE(x)` as the `scugic` function `XScuGic_SoftwareIntr()`; its objective is to notify another core that it needs to yield the task it is currently executing. The operation is performed inside `prvYieldCoreHandler()` by setting the variable `ulPortYieldRequired[xCoreID]` for the running core. Since this is a frequent operation, one could consider integrating it in the `FreeRTOS_IRQ_Handler`, as discussed in Section B.5.

Although the end of the code section that starts with `_boot_core1` features several calls to C functions, no further action is needed to comply with the AAPCS, since the stack pointer is always 8-byte aligned.

After the setup steps described in these sections, the scheduler can run on the secondary core, so CPU1 can finally execute its first task.

3.4 Symmetric Multiprocessing Port Upgrade

The previous section discussed the bring-up procedure of the Zynq-7000 SoC's secondary core without investigating the improvements introduced in the FreeRTOS port layer that enable this process. These changes are both numerous and diverse, so only the most important ones are presented in the following subsections.

3.4.1 Primary Core Setup in FreeRTOS

Subsection 3.3.2 mentions that CPU0 and CPU1 have different roles during the startup of FreeRTOS. Still, it does not anticipate that the primary core executes additional setup phases both in the hardware-dependent and independent layers.

Nevertheless, the current discussion solely focuses on the port code, specifically on the operations performed inside `xPortStartScheduler()`.

In SMP, although the scheduler is started on all the available cores, a single core must handle the system tick. This ensures the correct management of `xTickCount`, the variable that tracks the number of OS ticks since the start of the scheduler, and the FreeRTOS APIs that rely on it (e.g., `vTaskDelay()`). However, unlike in the single-core port, the installation of the SysTick interrupt in `FreeRTOS_SetupTick-Interrupt()` is not followed by restoring the first task's context. Indeed, CPU0 enables the SGI that implements the inter-core communication. Nevertheless, differently from CPU1, it calls `vPortSetupYieldRequestHandler()`, a function defined in `portZynq7000.c`, since it already initialized its CPU interface before configuring the system tick. After the previous step, the primary core checks if the spinlock API presented in Subsection 2.5.2 is disabled and, eventually, sets it up. At this point, as thoroughly described in Subsection 3.3.2, CPU0 jumps to `vPortLaunchSecondaryCore()` and starts executing its first task, once CPU1 finishes its boot sequence.

3.4.2 Multi-core Interrupt Management

The hardware-dependent layer handles the interaction between cores and GIC through the `scugic` driver, whose functioning is mostly described in Subsection 2.5.2. Nevertheless, its usage has not been flawless, since, even though multiple cores are supported, the driver's design is not SMP-oriented.

Subsection 2.2.2 states that the registers of the CPU interfaces are banked; hence, in principle, a single `XScuGic` instance could be used to set up both cores. However, upon its first call, `XScuGic_CfgInitialize()` configures the passed `XScu-Gic` instance so that it cannot be used to initialize more CPU interfaces. While it would technically be possible to overcome this limitation by manually modifying the `IsReady` field, this approach has been discarded to avoid potential issues in future library versions. Instead, the two-element array `xInterruptControllers[-configNUMBER_OF_CORES]` has been used to allocate a separate `XScuGic` variable per core.

Another substantial limitation lies in the interrupt handlers' management because, although an `XScuGic` instance is assigned to each core, different cores cannot have different handlers associated with the same interrupt ID. As already explained in Subsection 2.5.2, the driver only defines a single `XScuGic_Config` variable, thus a single `HandlerTable`. Addressing this issue would require modifying the `scu-gic` library or directly manipulating its variables in the port code. To preserve compatibility, the design has been adapted to follow the current structure of the driver. For this reason, the interrupt handling in the SMP port complies with the following rules:

- A single handler must service an interrupt on all the cores that enable it, no matter the interrupt type (i.e. SPI, PPI or SGI).
- After an interrupt handler is installed using `XScuGic_Connect()`, no call to `XScuGic_Disconnect()` or `XScuGic_Connect()` with another handler as argument must be performed as long as at least one core needs the handler associated with that interrupt.

These rules prevent to incur in some of the limitations of the `scugic` driver. Still, when multiple cores interleave in the usage of the library, further precautions are needed due to the internal management of `CpuId`. Supposing the hardware has version 1.0 of the GIC, like the Zynq-7000 SoC, the driver assigns the value returned by `XGetCoreId()` (i.e., the running core) to `CpuId` only at the beginning of `XScuGic_CfgInitialize()`; all other functions that use this variable never evaluate again the active core number. While this is not an issue for PPIs and SGIs, enabling or disabling an SPI on a certain core is not safe by default. Therefore, it is advised to check the value of `CpuId` with `XScuGic_GetCpuID()` and eventually change it with `XScuGic_SetCpuID()`. Specifically, the functions that require this measure are `XScuGic_Enable`, `XScuGic_Disable`, and `XScuGic_Stop`.

Even though the driver's design does not easily allow to manage SPIs, the configuration of these interrupts is flexible due to their nature. Since SPIs are shared among all cores, if a core enables one of such interrupts, the corresponding bit in the Distributor Interrupt Set-Enable Register [25, sec. 4.3.5] is observed by other cores too. The same applies to similar Distributor registers (e.g., Set-Pending Register, Clear-Pending Register). However, an SPI is forwarded only to the CPU interfaces of the cores selected as targets in one of the Interrupt Processor Targets Registers [25, sec. 4.3.11]. Therefore, every core can configure an SPI for any other core. Considering the behavior of `CpuId` described in the previous paragraph, an SPI can be enabled using the `scugic` driver by setting the target core with `XScuGic_SetCpuID()` before calling `XScuGic_Enable()`, independently from the running core.

Interrupt Management API for Applications

The application-level API provided to handle interrupts when FreeRTOS is configured in SMP is different from the single-core API included in the official Xilinx release. Although the interactions with the `scugic` driver are still completely managed through the API, the redundant checks on the `XScuGic` instance have been removed, and the various operations have been split into multiple functions to improve flexibility. Therefore, users need to manually perform some setup steps, which are hidden in the single-core API, but can also customize further their applications. For example, `vPortConfigureInterrupt()` allows to configure sensitivity and priority for a certain interrupt and `vInitialiseInterruptController()` enables to initialize the `XScuGic` of the executing core without the need to install a handler.

With regard to multi-core, the correct management of `CpuId` is handled by both `vPortEnableInterrupt()` and `vPortDisableInterrupt()`. Nevertheless, the requirements imposed by the usage of the `scugic` library in SMP must be dealt with by the applications, as enforcing those rules within the API would require additional logic.

3.4.3 Recursive Spinlocks

FreeRTOS requires two recursive locks to coherently manage mutual exclusion when operating in SMP, due to constraints imposed by the internal functioning of the kernel [13, pp. 4-5]. Nevertheless, the spinlocks' structure is the same; thus, a single set of functions must be implemented to handle both the "ISR" and "Task" locks. The spinlock mechanism provided in the FreeRTOS SMP port for the Zynq-7000 SoC is based on the `lock_t` data structure and on the `vPortGetLock()` and `vPortReleaseLock()` primitives, which respectively allow to acquire and release a lock. Implementing these spinlocks heavily relies on the exclusive access features detailed in Subsection 2.3.3.

`lock_t` is declared in `portmacro.h` and, as shown in Listing 3.2, it is composed of two fields: `ulLock` and `ulRecursionCount`. The latter holds the number of times its owner has recursively acquired the lock. At the same time, the former is the actual lock, hence the memory location targeted by the exclusive accesses, and stores the state of the spinlock. Its value can be either `portCORE0` (lock owned by CPU0), `portCORE1` (lock owned by CPU1) or `portLOCK_FREE` (lock free). The ISR and Task locks are defined by the array `xLocks[portLOCK_COUNT]` and, since the DDR memory's attributes analyzed in Section B.3 are compatible with the ones required to use the L1 cache monitors [4, pp. 142-143], the exclusive accesses can correctly end both in L1 cache and DDR memory. Moreover, considering that the elements of an array are allocated in subsequent memory locations, the GCC attribute at line 22 could be optionally included to align each `lock_t` variable to a 32-byte boundary. As the structure size is 8 bytes, such alignment ensures that different locks are stored in memory regions identified by different tags in an exclusive monitor. The final version of the SMP port does not implement this feature because, despite the warnings in the ARM documentation [8, ch. A3, p. 121], several tests have demonstrated that the spinlocks are not affected by starvation or livelock. Nevertheless, these conditions cannot be excluded a priori in a system that relies on the spinlock API to handle additional recursive locks.

The functions to acquire and release a spinlock are provided in `smpPortZynq-7000.S`, and while `vPortReleaseLock()` is implemented once, `vPortGetLock()`

```
/*** port.c ***/

/* Recursive ISR and TASK spinlocks */
volatile lock_t xLocks[portLOCK_COUNT] = { [0 ... (portLOCK_COUNT - 1)]
↪  = {portLOCK_FREE, 0}};

/*** portmacro.h ***/

#define portCORE0       0U
#define portCORE1       1U

/* Lock macros */
#define portLOCK_COUNT  2U

#define portISR_LOCK    0U
#define portTASK_LOCK   1U

/* The value of this macro has to be the same as the one
defined inside the assembly files. */
#define portLOCK_FREE   2U

typedef struct{
    uint32_t ulLock /*__attribute__( (aligned(32)) )*/ ;
    uint32_t ulRecursionCount;
} lock_t;
```

Listing 3.2: Recursive locks variables and macros

has two slightly different variants. Listing 3.4 reports only one of the available versions, but a user can choose between the two by properly setting the macro `configGET_LOCK_VERSION` in `FreeRTOSConfig.h`. Since they are hardware-dependent, the primitives to handle the recursive locks are not directly manipulated by the FreeRTOS kernel, but are hidden through the macros reported in Listing 3.3.

```c
extern void vPortGetLock( volatile lock_t* xLockAddr );
extern void vPortReleaseLock( volatile lock_t* xLockAddr);
extern volatile lock_t xLocks[portLOCK_COUNT];

#define portGET_TASK_LOCK()     vPortGetLock(&xLocks[portTASK_LOCK])
#define portGET_ISR_LOCK()      vPortGetLock(&xLocks[portISR_LOCK])
#define portRELEASE_TASK_LOCK()     vPortReleaseLock(&xLocks[portTASK_LOCK])
#define portRELEASE_ISR_LOCK()  vPortReleaseLock(&xLocks[portISR_LOCK])
```

Listing 3.3: FreeRTOS spinlock interface

`vPortGetLock()` is called with a single argument of type `volatile lock_t*`; thus, according to the AAPCS, the address of the lock is passed in the scratch register R0. The core ID is retrieved before entering the lock acquisition loop, which starts at line 9. The instructions executed inside the loop are similar to the basic example provided by ARM [8, ch. D7, p. 2446], but, since the lock can assume three values, an additional branch operation is required. As the spinlock supports recursive acquisitions, if the lock is not free, it is impossible to know which core owns it without further testing the held value. Consequently, when the lock is unavailable, the program jumps to line 30 to verify the lock's ownership. If the value stored in memory is equal to the ID of the running core, the execution proceeds by claiming the lock once again; otherwise, the core remains stuck in the loop at line 35, waiting for the complete lock's release. If the lock is free, the described branch is not taken by any core that might be trying to acquire the lock until one of them successfully executes the STREX instruction. Upon the acquisition of a lock, DMB is issued to ensure the state of the lock is written in memory and the recursion count is successively increased. Furthermore, when `configTEST_RECURSIVE_SPINLOCK` is set, `vPortGetLock()` includes some test code which can be used to catch errors during the usage of the API.

```
1 vPortGetLock:
2     /*   r0: lock address */
```

```

    /* Get core ID */
    mrc     p15, 0, r1, c0, c0, 5
    and     r1, r1, #AFF_MASK

    /* Try to claim the lock */
tryLockLoop:
    ldrex   r2, [r0]
    cmp    r2, #portLOCK_FREE   /* Check if the lock is free */
    bne    checkOwnership

    strex   r3, r1, [r0] /* Try to claim the lock */
    cmp   r3, #0  /* Check if the operation updated the memory (r2=0)
    ↪ */
    bne   tryLockLoop  /* If not loop until one claimer gets the lock
    ↪ */

    /* Ensures that all subsequent accesses are observed after the
    gaining of the lock is observed. */
    dmb

#if ( configTEST_RECURSIVE_SPINLOCK == 1 )
    ldr    r2, [r0, #4]
    cmp    r2, #0 /* Recursive count == 0 when first acquired */
    bne    .
#endif

    b    1f

checkOwnership:
    cmp    r2, r1
    beq    1f /* If the core owns the lock just reclaim it */

    /* Continue to loop until the lock is free */
waitLockLoop:
    ldrex   r2, [r0]
    cmp    r2, #portLOCK_FREE
    strexeq r3, r1, [r0]
    cmpeq   r3, #0
    bne   waitLockLoop

    /* Ensures that all subsequent accesses are observed after the
    gaining of the lock is observed. */
    dmb
```

```

1:
    /* From now on shared data are protected from concurrent accesses.
     ↪  */

#if ( configTEST_RECURSIVE_SPINLOCK == 1 )
    ldr    r2, [r0, #4]
    cmp    r2, #255 /* Recursive count < 255 when reclaimed */
    bhs    .
#endif

    /* Increase claim count */
    ldr    r2, [r0, #4]
    add    r2, r2, #1
    str    r2, [r0, #4]

    bx lr
```

Listing 3.4: Recursive spinlock acquisition function v1

`vPortReleaseLock()` is more aligned than `vPortGetLock()` to the example code in the ARM documentation [8, ch. D7, p. 2446]; nevertheless, it still presents some changes due to the recursive nature of the locks. The code in Listing 3.5 shows that the lock is freed only if the recursion count has reached zero. Furthermore, the barrier is placed before the STREX instruction to prevent the preceding memory operations from being observed after the lock's release.

3.5 FreeRTOS SMP Port Trace and Debug

The SMP port includes a configurable trace library that embeds the tools described in Subsection 2.5.3 to ease the debugging of both hardware-dependent and independent components. However, while trace macros are more suited to debug specific low-level issues, Percepio View offers tracing features that facilitate the analysis of wide applications. Moreover, unlike the trace macros, Percepio View must be properly set up both in the application code and library configuration files. Detailed information on the matter is illustrated in the Percepio documentation [26] [27] and in the "User manual" shipped with Percepio View.

Aside from some code in the portable layer, the API is entirely defined by the files in the directory `utility/` of the Zymq-7000 SoC SMP port [18].

```
.type vPortReleaseLock, %function
vPortReleaseLock:
    /*   r0: lock address */

#if ( configTEST_RECURSIVE_SPINLOCK == 1 )
    /* Get core ID */
    mrc      p15, 0, r1, c0, c0, 5
    and      r1, r1, #AFF_MASK

    ldr     r2, [r0]
    cmp     r1, r2      /* Core ID == Lock owner ID */
    bne     .        /* Error */

    ldr     r2, [r0, #4]
    cmp     r2, #0      /* Recursion count != 0 */
    beq     .        /* Error */
#endif

    /* Decrease the recursion count */
    ldr     r2, [r0, #4]
    sub     r2, r2, #1
    str     r2, [r0, #4]

    cmp     r2, #0

    /* Allow all the previous memory operations to be observed
    before releasing the lock */
    dmb

    /* Release the lock if all the claims have
    been released */
    moveq  r1, #portLOCK_FREE
    streq  r1, [r0]

    bx lr
```

Listing 3.5: Recursive spinlock release function

3.5.1 Trace Macros

The debugging of the SMP features integrated in the port has been initially aided by the trace macros scattered in the FreeRTOS kernel. A few of them have been used, as they were the most suited to discover subtle implementation errors: context-switch and recursive lock macros. The former appear in the code as `traceTASK_SWITCHED_IN()` and `traceTASK_SWITCHED_OUT()`, while the latter have been introduced in the port layer to test the locks at the kernel level. In particular, tracing the activity of the locks required to wrap `vPortGetLock()` and `vPortReleaseLock()` in the C function `vPortRecursiveLock()`, as well as to redefine the spinlock macros presented in Subsection 3.4.3. Furthermore, the context-switch macros have been implemented, so that two subsequent calls to `traceTASK_SWITCHED_IN()` on the same core block the program in a loop. This tweak has been crucial for finding a critical error in the port functions `ulPortSet-InterruptMask()` and `vPortClearInterruptMask()` [28].

While the single-core usage of the trace macros is straightforward, the design choices are not so immediate in multi-core, since the trace analysis can be either static or dynamic, and the structures that store the collected data can be either shared or dedicated. About these aspects, the SMP port instantiates a statically-allocated circular buffer for each core, whose size can be configured by the user. This solution has proven to be the most efficient, as it is not affected by the Universal Asynchronous Receiver-Transmitter (UART)'s issue reported in Section B.1 and is hence less invasive than dynamically transmitting data through a shared UART peripheral [29]. Moreover, while creating a thread-safe `xil_printf()` function is technically possible, doing so at the kernel level using the spinlock provided in the standalone BSP or recursive spinlocks can lead to deadlock, since both APIs are leveraged in the port layer.

3.5.2 Percepio View

Percepio View 4.10.3 has been a fundamental tool for performing offline analyses of the tested applications' behavior. However, tracing the activity of FreeRTOS running on a CA9 configured in SMP has required significant changes to the portable interface, as Percepio View does not natively support multi-core on CA9 processors. The necessary modifications are reported in a brief guide by Percepio [30].

An essential step in the multi-core configuration of Percepio View is the definition of the additional macros `TRC_CFG_CORE_COUNT` and `TRC_CFG_GET_CURRENT_CORE()` in `trcConfig.h`. The former must be set to the same value of `configNUMBER_OF_-CORES`, as it specifies the number of cores to trace, while the latter can be defined as `portGET_CORE_ID()`, since it serves as API to retrieve the ID of the running core. `TRC_CFG_HARDWARE_PORT` must also be defined as `TRC_HARDWARE_PORT_ARM_-CORTEX_A9` to include the portable code to work with the CA9. About `trcKernel-PortConfig.h`, `TRC_CFG_RECORDER_MODE` and `TRC_CFG_FREERTOS_VERSION` have to be respectively defined as `TRC_RECORDER_MODE_STREAMING` and `TRC_FREERTOS_-VERSION_11_1_0` to set the correct operating mode and FreeRTOS version. Only Streaming mode is capable of operating in SMP because, other than being deprecated [27], Snapshot mode integrates fewer features [31] and cannot be configured to trace multiple cores. The remaining settings are still important to customize the runtime data collection, but they are not critical to understand the next steps.

To complete the configuration, it is possible to modify `trcHardwarePort.c` and `trcHardwarePort.h`, the library files that handle the hardware-dependent operations. `trcHardwarePort.c` requires adapting `cortex_a9_r5_enter_critical` and `cortex_a9_r5_exit_critical` to use the critical section handling primitives provided in the SMP port, since their implementation is different from the single-core version. On the other hand, while `trcHardwarePort.h` needs to change the declaration of the functions modified in `trcHardwarePort.c`, it must also redefine the timestamp-generation logic, as every traced core must be able to access the used hardware timer [30]. The global timer has been the obvious choice for this purpose because, as described in Subsection 2.3.4, it is embedded directly in the CA9 and can be accessed by all the available cores. Although Xilinx provides a way to access the counter register, a small library that further eases the configuration of the global timer and the setup function `vTraceSetupGlobalTimer()` have been developed respectively in `gtimer.h` and `trace.c`. More specifically, the sequence of instructions implemented in `vTraceSetupGlobalTimer()` configures the prescaler to zero to allow both the global timer and the private timers to run at the same frequency, since, as explained in Subsection 2.3.4, they are clocked with the same signal. Ultimately, `configCPU_CLOCK_HZ` must be defined to `XPAR_CPU_CORTEXA9_CORE_CLOCK_FREQ_HZ / 2UL` in `FreeRTOSConfig.h` to let Percepio View know the clock frequency of the timer.

The changes described in this section are provided with the port in the `Percepio View 4.10.3 patch/` directory [18]. Furthermore, the configuration of Percepio View for an application that uses the SMP port is shown in the `main.c` file of the Zynq-7000 SoC test demo [19].

3.6 FreeRTOS SMP Port Test Demo

The upgrade of the FreeRTOS port for Zynq-7000 SoC has been debugged and tested along its development using several ad hoc programs, which targeted specific port functions without considering special scenarios (e.g., interrupt nesting) or interacting in particular ways with the FreeRTOS features. Therefore, such programs have not been deemed sufficient to validate the port, requiring the adoption of a comprehensive test suite.

The FreeRTOS repository [17] already includes a demo that targets the Zynq-7000 SoC in the directory `Demo/CORTEX_A9_Zynq_ZC702`. However, although the demo integrates a complete set of tests, its template is outdated, and the target board differs. Consequently, rather than adapting it to both the SMP port and the TUL Pynq-Z2, a new test demo was developed using the template under `Demo/ThirdParty/Template`. Nonetheless, the demo applications for the ZC702 Evaluation Kit and the Zynq UltraScale+ MPSoC (`Demo/CORTEX_A53_64-bit_UltraScale_MPSoC`) have been used as a reference during the development.

Considering `Demo/ThirdParty/Template` as root directory, while the general setup of the demo is common to all devices and is thoroughly described in `README.md`, the implementation of `IntQueueTimer.*` and `RegTests.*` depends on the target system. Specifically, the latter test contains the logic to verify the correct preservation of the task context, while the former acts as an interface between the hardware and the high-level code in `IntQueue.c`. This file, together with the sources and headers respectively located in the `Demo/Common/Minimal` and `Demo/Common/include` folders, provides the actual tests that are common to all the demos created with the FreeRTOS template. Regarding its functioning, the demo operates by setting up the test routines enabled in `FreeRTOSConfig.h`, as well as the control task `prvCheckTask()`, before starting the scheduler. This is done with a call to the `vStartTests()` function, which is defined in `TestRunner.c`. Such a file contains the entire mechanism for starting the selected tests and periodically monitoring their status, reporting eventual errors.

As for the test files to be implemented, the sequence of instructions to prove the correct preservation of the tasks' context is illustrated inside `RegTests.c` and is not influenced by the number of cores. The test logic consists of filling the general-purpose and FP registers with known values, before entering an infinite loop until one register is corrupted. If this happens, the appropriate `ulRegister-TestXCounter` is set and, when `prvCheckTask()` is resumed, the issue is reported to the user. The test has been coded in the assembly file `RegTest.S` instead of `RegTests.c` to exploit the greater controllability over the registers' usage.

Implementing the nested interrupts' test is more complex than the previous one, since it combines the usage of the FreeRTOS queues with the nesting support offered by the CA9. Moreover, due to its logic and the usage of port-level functions, this test depends on the number of running cores. As for the other tests, the setup is performed inside `vStartTests()` using `vStartInterruptQueueTasks()`, which creates the kernel objects instantiated in `IntQueue.c`. Nevertheless, since this function does not interact in any way with the portable layer, the configuration is completed with a call to `vInitialiseTimerForIntQueueTest()` in the higher-priority task "H1QRx". `vInitialiseTimerForIntQueueTest()` is defined in `IntQueueTimer.c` and its function is to set up the GIC and a TTC module to properly manage the interrupts required for the test. However, while the configuration of the TTC is independent of the number of available cores, that of the GIC is not, since, in the multi-core version, the issues of the `scugic` driver are addressed through the API presented in Subsection 3.4.2. Regarding the configuration of these units, the TTC is set up to generate three periodic interrupts: two with comparable periods and one that fires ten times more frequently than the others. Furthermore, the GIC is configured so that the two low-frequency interrupts can safely use the FreeRTOS API, while the remaining one cannot, since it is assigned a priority below `configMAX_API_CALL_INTERRUPT_PRIORITY`. Such a setting allows the interrupt not to be masked during the execution of critical sections, hence to be leveraged to stress the implementation of `FreeRTOS_IRQ_Handler()`. Aside from configuring their priorities, `vInitialiseTimerForIntQueueTest()` also associates the interrupt to `prvTimerHandler()`. Each time the handler is invoked, it records the maximum nesting depth and counts the number of times it is executed. Only when a lower-frequency interrupt is serviced, it calls either `xFirstTimerHandler()` or `xSecondTimerHandler()`. `prvTimerHandler()` could also be used to test FP operations in interrupt service routines; however, this feature has not been included in the final demo. Section B.8 explains the reasons behind this choice and provides useful insights on the usage of FP registers within interrupt handlers.

Although the described implementation is fully functional on the classic CA9 port, running in SMP mode requires some additional tweaks because the test logic was initially developed for a single-core scenario. For example, even though the TTC interrupts can be managed by multiple cores, simultaneously executing `xFirstTimerHandler()` and `xSecondTimerHandler()` makes one of the tested conditions fail, causing `prvQueueAccessLogError()` to flag an error. Enabling other tests revealed that `StreamBufferDemo.c` and `MessageBufferDemo.c` are also affected by similar issues.

After several hours of tracing, the flawed code sections were spotted and properly fixed with solutions aimed at preserving the original single-core logic while avoiding unnecessary assumptions about the tasks' state. More specifically, the former objective has been achieved by binding certain tasks to a single core and limiting the kernel's ability to concurrently execute tasks with different priorities. Additional information about these solutions can be found in the `README.md` file within the test demo repository [19]. Moreover, a detailed discussion of the errors in `IntQueue.c` is available on the FreeRTOS forum [32].

Chapter 4

Results

As anticipated in Chapter 3, the Zynq-7000 SMP port for FreeRTOS [18] and its test demo [19] are already available on the official FreeRTOS repositories. After its release, the port has been upgraded to version 11.2.0 of the FreeRTOS kernel [12, tag `V11.2.0`], but this process has not been documented, as it only required minor changes.

To assess whether running FreeRTOS in SMP mode on the Zynq-7000 SoC brings some advantages, it is possible to compare it to an equivalent AMP configuration. For this purpose, a simple benchmark program has been created for both SMP and AMP modes, as explained in Chapter C. More specifically, the application uses version 11.2.0 of the FreeRTOS API to create two tasks and a queue, which implement a producer-consumer scenario. The evaluation focuses on both the size of the generated executables and the performance of the two MP configurations.

The size of the programs is shown in Table 4.1. While for SMP the measurements correspond to a single executable, for AMP they incorporate the Executable and Linkable Format files for both cores. Depending on the optimization flag, the binary of the SMP program is on average 9% bigger than the equivalent binary produced for a single core configured in AMP. Nevertheless, independently of the optimization level, the SMP program has a smaller memory footprint than the combined AMP binaries. The benchmark application reaches the maximum of a 47% size reduction when the compilation is oriented to limit the executable's dimension (`-Os` flag). Therefore, the SMP configuration can more easily be stored by memory-constrained systems.

Concerning performance, Tables 4.2 and 4.3 report the average execution timings over ten samples of various test cases, which differ in the number of exchanged items, queue size, and optimization flags. In AMP, each task is bound to a core, whereas in SMP the scheduler freely performs the workload distribution. The two MP configurations deliver similar performance, since, comparing the best results for the 10k items scenario, the difference is a mere 4% in favor of

SMP. Table 4.4 shows that this result could stem from the slight difference in the programs' setup time, caused by the longer execution of the `main()` in the AMP configuration. Unfortunately, running in AMP with a queue size higher than one creates synchronization issues. However, this limitation should not significantly interfere with the proposed conclusions, since tasks can only gain a minor performance improvement from having a larger queue. This is confirmed by the results reported for SMP. Furthermore, increasing the number of exchanged items does not yield new insights, as the execution time increases linearly regardless of the adopted MP strategy.

When a third higher-priority task is introduced, the gap between AMP and SMP becomes more evident. In this scenario, the SMP program delivers the items from task "Tx" to task "Rx" up to 33% faster than AMP. Increasing the queue size in AMP is only helpful if the producer can store items in the queue while the consumer cannot receive them. If the bottleneck task shares the same core as the producer, the queue size does not impact the final execution time. In contrast, the SMP kernel can execute the high-priority task on one core and let the producer and consumer run on the other. Therefore, the performance benefits from a bigger queue, since each task can process more items in its execution slot. Limiting the tasks' schedulability (e.g., tasks with different priorities cannot run simultaneously) inevitably reduces the performance.

Although the analysis in this chapter is based on a simple benchmark, it clearly shows the advantages of using the FreeRTOS SMP port on the Zynq-7000 SoC. Obviously, the numerical results are strictly tied to the considered application; however, other aspects are also worth considering. As discussed in Appendix C, the AMP setup is quite tricky and must be tailored to both a specific program and a target system. On the other hand, in SMP, the memory layout, the kernel-level synchronization on shared resources, and task scheduling can be managed entirely by the OS. Therefore, portability and maintainability are improved, as designers can develop applications without considering low-level details.

Multiprocessing configuration	**Optimization**	**Size (B)**
Asymmetric Multiprocessing	-O0	405438
	-O3	378522
	-Os	360430
Symmetric Multiprocessing	-O0	220711
	-O3	211475
	-Os	196591

Table 4.1: Program size comparison

#Items	**Queue size**	**Optimization**	**Avg. Exec. Time (ms)**
10000	1	-O0	33.4
		-O3	21.0
		-Os	22.1
	10	-	-
100000	1	-O0	-
		-O3	2047
		-Os	-

Table 4.2: AMP benchmark execution times

#Items	**Queue size**	**Optimization**	**Avg. Exec. Time (ms)**
10000	1	-O0	29.8
		-O3	21.1
		-Os	20.5
	10	-Os	20.2
	100	-Os	20.1
1000000	1	-O0	-
		-O3	-
		-Os	1983

Table 4.3: SMP benchmark execution times

MP	**Sampling start**	**Optimization**	**Avg. Exec. Time (ms)**
AMP	`main()`	-O3	21.0
	`prvTxTask()`	-O3	21.6
SMP	`main()`	-Os	20.5
	`prvTxTask()`	-Os	20.1

Table 4.4: Impact of benchmark program's setup on the final execution time for different MP configurations

MP	**Queue size**	**Optimization**	**Avg. Exec. Time (ms)**
AMP	1	-O3	30.9
	10	-	-
SMP	1	-Os	28.5
	10	-Os	25.3
	100	-Os	23.1
	1000	-Os	23.0

Table 4.5: Benchmark execution time with 10000 items and additional high-priority task

Chapter 5

Conclusion

This thesis investigated the usage of SMP in applications based on an RTOS and in embedded devices that support such MP paradigm.

The main point was understanding whether SMP could benefit the embedded domain as it is in others. More specifically, the research aimed at verifying if SMP could improve the design of MP applications even in older platforms, for which AMP had historically been the obligated choice.

The thesis focused on porting the SMP support of FreeRTOS, a Real-Time Operating System, on the Zynq-7000 SoC, an embedded platform that integrates a dual-core processor.

The work started with the analysis of Xilinx's development tools. This allowed us to set up the hardware-software environment and to discover the available options to integrate FreeRTOS in the existing software structure. The choice was to include it as an external library in a bare-metal application to improve compatibility with future versions of the tools.

After this preliminary phase, the existing FreeRTOS port for Zynq devices was inspected, allowing us to familiarize ourselves with the FreeRTOS portable layer and understand the changes introduced to the CA9 template available on the FreeRTOS repository. Examining the port revealed that several device-specific adjustments were necessary to adapt FreeRTOS to the Zynq-7000 SoC.

Subsequently, the required steps to boot the system in dual-core mode were collected. While the BSP set up the primary core to work in SMP, no already functioning driver was provided to wake up the secondary core. For this reason, the bring-up procedure was completely implemented following the Zynq-7000 SoC documentation. The instructions enabled waking up the secondary core from the primary, allowing it to execute an initial configuration. Moreover, the bare-metal environment was modified to allocate a dedicated memory section for the correct execution of the secondary core.

The dual-core boot process was successively incorporated in the FreeRTOS portable layer. Nevertheless, further steps were added to initialize both cores to ensure the FreeRTOS kernel could correctly control them. Therefore, the cores were provided with a way to configure the GIC, to protect critical sections in the Xilinx drivers, and to communicate with each other. A final call to the scheduler was inserted to start it on all the available cores, as required by FreeRTOS.

Additional changes were introduced in the port layer to adapt it to handle the execution of multiple cores, among all the implementation of the recursive locks. An API was also included to allow users to manage application interruptions.

The porting process concluded with its validation using a test demo based on a template provided on the FreeRTOS repository. However, the tests required some changes, as they were developed for single-core scenarios.

In the final phase, a benchmark program assessed whether Symmetric Multiprocessing improved the results obtained with AMP. The outcome demonstrated that running FreeRTOS in SMP mode allows achieving better execution timing and memory footprint on the tested application. The configuration of the programs to operate in SMP and AMP also proved that SMP offers better maintainability and portability on systems that support it.

In conclusion, this thesis confirms that configuring embedded platforms equipped with symmetric CUs in SMP offers significant advantages over other MP options. These benefits apply even to older devices with a lower core count. Nevertheless, the availability of a software layer to manage the architectural complexity of such systems remains crucial. Therefore, providing application designers with these tools is fundamental to fully exploit the hardware's potential.

Appendix A

Reference Manuals

The following tables report the manuals consulted during the development of this thesis, properly grouped according to the treated topic.

Unit	**Supplier**	**Version**
Zynq 7000 SoC TMR [4]	AMD	1.14
Zynq 7000 SoC Software Developers Guide [5]	AMD	13.0

Table A.1: Zynq-7000 SoC: Reference manuals

The versions of the manuals in Table A.1 are consistent with the ones listed in the Zynq-7000 SoC TMR [4, pp. 1934–1935]. In case of discrepancies in the versioning of a manual, the most appropriate document has been chosen:

- Arm Cortex-A9 MPCore Technical Reference Manual, Revision r3p0 (DDI0407F)

  The values of the revisions are incompatible; revision F corresponds to the first release of r2p2 and revision G corresponds to the first release of r3p0.

- Arm Generic Interrupt Controller v1.0 Architecture Specification (IHI 0048B)

  The architecture's version and the revision are incompatible; GIC v1.0 corresponds to revision A and v2.0 corresponds to revision B.

Unit	Supplier	Version
Cortex-A9 MPCore TMR [10]	ARM	G
Cortex-A9 TMR [9]	ARM	G
ARM Cortex-A Series Programmer's Guide [6]	ARM	4.0
ARM Architecture Reference Manual (ARMv7-A) [8]	ARM	C
PrimeCell Generic Interrupt Controller (PL390) TMR [7]	ARM	B
ARM Generic Interrupt Controller v1.0 Architecture Specification [25]	ARM	A

Table A.2: Zynq-7000 SoC: Third-party IP and standards documents

Unit	Supplier	Version
Cortex-A9 MPCore TMR [24]	ARM	G
PYNQ-Z2 Reference Manual [33]	-	1.0
FreeRTOS SMP Change Description [13]	XMOS	-
Vitis Unified Software Platform Documentation: Embedded Software Development [21]	Xilinx	2021.2

Table A.3: Reference manuals: Miscellaneous

Appendix B

Implementation Details

This chapter provides insights on the most relevant topics faced while developing the FreeRTOS SMP port for the Zynq-7000 SoC.

B.1 Multiple UART Connections on PYNQ-Z2

Configuring multiple UART connections on the PYNQ-Z2 can be cumbersome. While UART0 is connected through MIO 14 and 15 to the FTDI chip, making it accessible from the Micro-USB port, UART1 must be routed via EMIO (i.e., I/O between the PS and PL) into the PL and then out through one of the available external interfaces (e.g., a PMOD port) [34].

B.2 CPU1 BootROM Code

The debug of an AP based on a PP that uses CPU0 as domain reveals that, after the FSBL phase, the program counter of CPU1 is set to 0xFFFFFF34. This indicates that the processor is executing the instruction stored at address 0xFFFFFF32. Keeping in mind that, in the ARMv7-A architecture, the instruction memory is always mapped in little-endian [8, ch. A3, pp. 109], it is possible to decode such instruction (0xE320F002) as WFE (0x02F020E3).

Considering that 0xFFFFFFF0 holds the address 0xFFFFFF20, Listings B.1 and B.2 report the content of the memory region executed after the issue of a SEV instruction, to further explain the safety mechanism mentioned in Subsection 3.3.1.

```
 Address   0 - 3     4 - 7     8 - B     C - F
 FFFFFF10  00000000  00000000  00000000  00000000
 FFFFFF20  F57FF04F  E320F002  EAFFFFFC  F57FF04F
 FFFFFF30  E320F002  E3E0000F  E590E000  E37E00D4
 FFFFFF40  0AFFFFF9  EE070F15  EE070FD5  EE080F17
 FFFFFF50  E3A04000  EE014F10  E12FFF1E  00000000
 FFFFFF60  00000000  00000000  00000000  00000000
```

Listing B.1: Memory dump of CPU1 boot instructions

```
ffffff20:   dsb     sy
ffffff24:   wfe
ffffff28:   b       -16          ; branch at address 0xffffff20
ffffff2c:   dsb     sy
ffffff30:   wfe
ffffff34:   mvn     r0, #0xf     ; r0 = not(0xF) = 0xFFFFFFF0
ffffff38:   ldr     lr, [r0]     ; load value in 0xFFFFFFF0 to LR
ffffff3c:   cmn     lr, #0xd4    ; compare LR and (-0x4D) = 0xFFFFFF2C
ffffff40:   beq     -28          ; branch at address 0xffffff2c
; invalidate the entire instruction cache
; r0 is ignored and opc2 is 0, since it is omitted
ffffff44:   mcr     p15, 0, r0, c7, c5
; invalidate the branch predictor
ffffff48:   mcr     p15, 0, r0, c7, c5, 6
; invalidate the entire unified TLB
ffffff4c:   mcr     p15, 0, r0, c8, c7
ffffff50:   mov     r4, #0
; move r4 to the SCTLR in c1 (CP15)
ffffff54:   mcr     p15, 0, r4, c1, c0
ffffff58:   bx      lr  ; branch to the address in 0xFFFFFFF0
```

Listing B.2: ARMv7-A instructions decoded from Listing B.1

B.3 MMU Configuration for SMP

An in-depth analysis of the MMU configuration reported in Subsection 3.3.1 is necessary to verify that the DDR memory is correctly set up to work in SMP.

The code shown in Listing B.4 configures 1 GB of memory; enough to cover the entire 512 MB of DDR3 memory included on the PYNQ-Z2 [33, p. 10]. The linker script `lscript.ld` actually sizes the DDR region `ps_ddr_0` as 511 MB (0x1FF00000) because it begins at address 0x100000 (1 MB), in accordance with the Zynq-7000 SoC system memory map [4, p. 106].

Since the number of modified L1 page table entries is 1024 and the two least significant bits are 0b10 [4, pp. 67-68], each entry identifies a 1 MB page. Starting from the string 0x15de6, every iteration of the loop adds 0x100000 to the Section Base Address to apply the memory attributes to the next page. Besides the S bit, other important fields are Domain, TEX, B, and C. The Domain field, together with the Domain Access Control Register [8, ch. B4, p. 1554], is configured to prevent the generation of any permission fault during a memory access. At the same time, the TEX, B and C bits define the memory as cacheable with write-back/write-allocate inner and outer policy. For additional information refer to the Zynq-7000 SoC TMR [4, pp. 67-75].

```
/* Write to ACTLR */
mrc  p15, 0, r0, c1, c0, 1   /* Read ACTLR*/
orr  r0, r0, #(0x01 << 6)    /* set SMP bit */
orr  r0, r0, #(0x01 )        /* Cache/TLB maintenance broadcast */
mcr  p15, 0, r0, c1, c0, 1   /* Write ACTLR*/
```

Listing B.3: Core takes part in the SCU cache coherency protocol

B.4 Acknowledge CPU1 Boot End from CPU0

After the execution of the SEV instruction, CPU1 jumps to the value of the memory location 0xFFFFFFF0 [4, p. 160]. Meanwhile, CPU0 cannot start the first task, so it needs to wait for CPU1 to finish its initialization. In principle, the solution presented in Subsection 3.3.1 is not unique, since the procedure could also be implemented differently. A possible alternative is to immediately set a shared variable to allow the primary core to enter a low-power state after the secondary core wakes up. The primary core would then be awakened by generating an event just before CPU1 executes the application. This would prevent CPU0

```
#ifdef SHAREABLE_DDR
  /* Mark the entire DDR memory as shareable */
  ldr  r3, =0x3ff         /* 1024 entries to cover 1G DDR */
  ldr  r0, =TblBase       /* MMU Table address in memory */
  ldr  r2, =0x15de6       /* S=b1 TEX=b101 AP=b11, Domain=b1111, C=b0,
  ↪  B=b1 */
shareable_loop:
  str  r2, [r0]           /* write the entry to MMU table */
  add  r0, r0, #0x4       /* next entry in the table */
  add  r2, r2, #0x100000  /* next section */
  subs  r3, r3, #1
  bge  shareable_loop     /* loop till 1G is covered */
#endif
```

Listing B.4: Page table configuration for DDR memory

from spamming event signals, potentially triggering other hardware units in the PS and in the PL [4, pp. 45-46].

This solution has been discarded because the memory configuration after the awakening of CPU1 does not allow to easily manage a shared variable. In fact, at this point, the L2 cache has already been enabled during the standalone BSP's boot and, while the cache coherency of CPU0 is handled by the SCU, that of CPU1 is not. The code that implements this solution is reported in Listing B.5. However, despite the absence of known errors, it has not been considered safe due to the author's lack of information on the access order to the L2 cache. For example, CPU1 might write to a shared variable while CPU0 performs maintenance operations on the cache line that hosts it.

The uncertainty that characterizes some parts of such code is underlined by the need to clean and invalidate the line holding the shared variable in the L2 cache inside the loop that starts at line 31. Theoretically, these operations should not be necessary because CPU1 should write directly in the L2 cache, so, after CPU0 invalidates the shared variable in its L1 cache, the SCU should correctly fetch this line from the second level of cache.

A solution to the problematic memory configuration could be not to enable the L2 cache during the boot of the standalone BSP. Nevertheless, while a multi-core boot does not impose any constraint in this regard [24, ch. 5, pp. 10-11], doing so would require to modify `boot.S`, thus the BSP. Moreover, it is not possible to exclude an additional issue that might be present in the code shown in Listing B.2, since there is a DSB before the WFE instruction but not after it. In case speculation was allowed, this could lead CPU1 to see stale data in its L1 cache because the load instruction might be executed before both the WFE and the cleaning of the L1 cache of CPU0.

```
 1  // CPU1
 2  _boot_core1:
 3      /* Notify CPU0 that CPU1 is awake */
 4      /* Data L1 is not active yet so the write goes either in
 5       * L2 or main memory. */
 6      mov  r1, #1
 7      ldr r2, =ulSecondaryCoreAwake
 8      str r1, [r2]
 9      dsb
10
11      /* Initialize CPU1 */
12
13      sev   /* Wake up CPU0 */
14      sev
15      sev   /* Issue more than one just to be safe */
```

```

// CORE0
vPortLaunchSecondaryCore:

    /* Code common to Listing 3.1 */

    /* Wake up CPU1 */
    ldr r3, =L2CCCleanPa
    ldr r4, =L2CCInvPa
    ldr r5, =L2CCCleanInvPa
    ldr r2, =ulSecondaryCoreAwake
    mcr p15, 0, r2, c7, c10, 1 /* Clean address to PoC */
    dsb
    str r2, [r3] /* Clean address in L2 */
    dsb
sendEvent:
    sev
    /* Hardware coherency through the SCU is still not active
     * on CPU1 so CPU0 may read stale data in its D-L1 cache if the
     * line holding ulSecondaryCoreAwake is not invalidated */
    mcr p15, 0, r2, c7, c6, 1 /* Invalidate address to PoC in D-L1 */
    dsb
    str r2, [r5] /* Clean and Invalidate address in L2 */
    dsb
    ldr r1, [r2] /* Fresh read from L2 or main memory */
    cmp r1, #0
    beq  sendEvent

    /* Wait for CPU1 initialization */
    wfe

    bx lr
```

Listing B.5: Alternative CPU1 wake-up process

B.5 Core Yield's Integration in FreeRTOS Interrupt Handler

A way to optimize the management of yield requests, which come from external cores, could be to integrate the logic inside `FreeRTOS_IRQ_Handler` instead of associating the SGI to a handler with the `scugic` driver. In practice, the idea is to skip the section of code that starts at line 11 and ends at line 25 in Listing B.7.

However, the logic required to implement it would use additional registers and control sequences, increasing the handling time of other interrupts (e.g., system tick interrupt). Due to these concerns, the definitive adoption of the proposed solution should be preceded by profiling sessions. In case the tests were to produce mixed results, the macro in Listing B.6 could be used to distinguish between the new and the classic version of `FreeRTOS_IRQ_Handler` depending on the specific application.

```
/* Set configUSE_YIELD_CORE_HANDLER to 0 to integrate the
↪  portYIELD_CORE(x) logic into FreeRTOS_IRQ_Handler, to 1 otherwise.
↪  */
#define configUSE_YIELD_CORE_HANDLER        0
```

Listing B.6: Macro to select the `FreeRTOS_IRQ_Handler` version

B.6 Cortex-A9 Nested Interrupt Handling in FreeRTOS

According to Subsection 2.4.2, FreeRTOS supports a full interrupt nesting model. However, although this might appear to be confirmed by examining the code of `FreeRTOS_IRQ_Handler`, such a handler is slightly different from the one reported as an example in the official ARM documentation [6, sec. 12.1.3]. Specifically, the I bit of the CPSR remains set when control is transferred to the application IRQ handler, as it is not cleared after being set by the processor upon entering the IRQ exception. Therefore, while interrupt nesting is supported, its usage is not imposed, since the user must consciously enable it by clearing the interrupt masking within the application-level handler. The SysTick handler `FreeRTOS_Tick_Handler()` is an exception to this rule because, by default, it reactivates the reception of interrupts through a call to `portCPU_IRQ_ENABLE()`.

B.7 Exclusive Access Tests on the Zynq-7000 SoC

The implementation of the primitives illustrated in Subsection 3.4.3 has been extensively tested to investigate the behavior of LDREX and STREX on the Zynq-7000 SoC; the results are reported below:

- Two cores access two locks with the same tag:

```
.type FreeRTOS_IRQ_Handler, %function
FreeRTOS_IRQ_Handler:

    /* Code not relevant */

    /* Read value from the interrupt acknowledge register, which is
    ↪  stored in r0 for future parameter and interrupt clearing use.
    ↪  */
    LDR     r2, ulICCIARConst
    LDR     r2, [r2]
    LDR     r0, [r2]


    /* Ensure bit 2 of the stack pointer is clear.  r2 holds the bit 2
    ↪  value for future use. _RB_ Does this ever actually need to be
    ↪  done provided the start of the stack is 8-byte aligned? */
    MOV     r2, sp
    AND     r2, r2, #4
    SUB     sp, sp, r2

    /* Call the interrupt handler. R4 originally pushed to maintain
    ↪  alignment. It contains the core ID in the SMP port. */
    PUSH    {r0-r4, lr}
    LDR     r1, vApplicationIRQHandlerConst
    BLX     r1
    POP     {r0-r4, lr}
    ADD     sp, sp, r2
    /* R2 can be used again */


    CPSID   i
    DSB
    ISB

    /* Write the value read from ICCIAR to ICCEOIR. */
    LDR     R2, ulICCEOIRConst
    LDR     R2, [R2]
    STR     R0, [R2]
```

Listing B.7: Section of `FreeRTOS_IRQ_Handler` that could be optimized to handle external yield requests

1. Both cores fail on the first attempt to acquire the locks even if the monitor is previously cleared with CLREX.
2. Executing an LDREX on CPU0 and issuing the LDREX-STREX instructions on CPU1 leads to complete the exclusive access on CPU1. As expected, a successive STREX on CPU0 fails. The outcome is different if the same scenario is repeated switching the cores' roles. In this case, the sequence of LDREX and STREX on CPU0 fails, while the STREX on CORE1 succeeds.
3. An LDREX on CPU1, followed by an LDREX on CPU0, leads to a successful STREX on CPU1 and a failed access on CPU0. However, if the execution order is reversed, the access on CPU1 succeeds while the one on CPU0 is unsuccessful.

- Two cores access two locks with different tags:

  1. Both cores fail on the first attempt to acquire the locks even if the monitor is previously cleared with CLREX.
  2. Executing an LDREX on CPU0 and issuing the LDREX-STREX instructions on CPU1 leads to complete the exclusive access on CPU1. As intended, a successive STREX on CPU0 does not fail because the state of the monitor used by CPU0 is not invalidated. The result is equivalent if the roles of the cores are reversed.

- When two cores try to claim the same lock, the observed behavior is identical to performing an exclusive access on addresses that produce the same tag. Therefore, during a contention, CPU1 wins over CPU0.

The tests' results do not align with the global monitor's description provided by ARM [8, ch. A3, 116–119]. More specifically, executing an LDREX instruction on a core should not interfere with exclusive accesses to the same tag of another core.

To assess the consistency of such results, the same set of tests has been executed with the spinlock API provided in the standalone BSP:

- Two cores access the same lock

  1. Issuing an LDREX on CPU0 and then on CPU1 leads to a successive STREX on CPU0 to succeed. The outcome is equivalent if the roles are switched.
  2. Performing an LDREX on CPU1, followed by a LDREX-STREX sequence on CPU0, results in a successful exclusive access on CPU0. The same happens on CPU1 when the roles are reversed.

Unlike the spinlock implemented in the port layer, the one included in the standalone BSP exhibits the expected behavior. One possible explanation is that the Zynq-7000 SoC favors exclusive accesses from CPU1 in case of contention, in accordance with ARM's guidelines [8, ch. A3, p.121]. Nevertheless, this behavior should have also been observed during the tests on Xilinx's spinlock.

Further experiments on the code have been conducted to unravel the causes of the reported behavior; however, no definitive conclusion has been reached. For example, an unsuccessful attempt to solve the failure on the first exclusive access to a lock led to disabling the L2 cache, since it does not have any monitor.

Even though the cause of these observations is still unknown, it has been noticed that debugging the code that performs the exclusive access in instruction-step mode leads STREX to fail more often and in less predictable ways. This might be consistent with possible interferences to the monitor's state due to store instructions [8, ch. A3, p. 117] and cache maintenance operations [8, ch. A3, pp. 120-121].

B.8 Floating Point Operations within Interrupt Handlers

The `IntQueueTimer.c` file included in the ZC702 demo of the FreeRTOS repository [17] contains sections of code dedicated to testing FP operations within interrupt handlers. However, as explained in Section 3.6, the Zynq-7000 SoC demo created for the SMP port has not been equipped with such a feature. The reason is that changing the application interrupt handler `vApplicationIRQHandler()` to its FP-safe version `vApplicationFPUSafeIRQHandler()`, as mentioned in Subsection 2.4.2, completely bloats the demo due to the high volume of serviced interrupts. Therefore, `prvCheckTask()` cannot print any message to inform the user about the status of the tests.

Unfortunately, since the application-level handler is the same for every interrupt, it is not even possible to selectively preserve the FP context without modifying the implementation of `FreeRTOS_IRQ_Handler()`. Furthermore, manually saving the FP registers inside an interrupt handler is not feasible, as the compiler may use them to optimize some memory operations before the registers can be pushed onto the stack.

Finally, as explained in Subsection 2.4.2, it is essential to understand that, even though the FP registers are not explicitly modified, if the FreeRTOS APIs is used inside an interrupt service routine, the FP context might still need to be saved. For instance, considering the test demo, if `prvTimerHandler()` enabled interrupts, defining `vApplicationFPUSafeIRQHandler()` could be necessary to correctly execute the application. Both `xFirstTimerHandler()` and `xSecond-TimerHandler()` make use of the queue's functions, so if any of the two interrupted the other while it is executing a `memcpy()`, the FP registers would probably be corrupted. Listing B.8 shows the `memcpy()`'s usage in the queue functions.

```
0010b3d4 <xQueueReceiveFromISR>:
  10b3d4:   e92d47f0   push  {r4, r5, r6, r7, r8, r9, sl, lr}
  10b3d8:   e2504000   subs  r4, r0, #0
 ...
  10b44c:   eb0072cb   bl  127f80 <memcpy>
  10b450:   e3790001   cmn  r9, #1
  10b454:   e2455001   sub  r5, r5, #1
 ...

001178ec <xFirstTimerHandler>:
  1178ec:   e92d4070   push  {r4, r5, r6, lr}
 ...
  117958:   e1a0200d   mov  r2, sp
  11795c:   e28d1004   add  r1, sp, #4
  117960:   ebffce9b   bl  10b3d4 <xQueueReceiveFromISR>
  117964:   e3500001   cmp  r0, #1
 ...

00127f80 <memcpy>:
  127f80:   e1a0c000   mov  ip, r0
  127f84:   e3520040   cmp  r2, #64  @ 0x40
  127f88:   aa000028   bge  128030 <memcpy+0xb0>
 ...
  128080:   aa000032   bge  128150 <memcpy+0x1d0>
  128084:   ed910b00   vldr  d0, [r1]
  128088:   e25aa040   subs  sl, sl, #64  @ 0x40
  12808c:   ed911b02   vldr  d1, [r1, #8]
  128090:   ed8c0b00   vstr  d0, [ip]
  128094:   ed910b04   vldr  d0, [r1, #16]
  128098:   ed8c1b02   vstr  d1, [ip, #8]
  12809c:   ed911b06   vldr  d1, [r1, #24]
```

Listing B.8: Usage of `memcpy()` within an interrupt handler

Appendix C

Benchmark Program

Chapter 4 analyzes the results gathered with a simple benchmark program implemented on the Zynq-7000 SoC in the SMP and AMP configurations. This chapter offers additional details on the setup of such an application in AMP, along with its code. However, the FreeRTOS configuration is omitted, since `FreeRTOSConfig.h` can be directly retrieved from the SMP port [18].

Although the SMP program is not included, it can be easily obtained by combining the code provided for AMP.

C.1 AMP Configuration

Several online guides describe the preliminary steps required to create an application running in AMP on the Zynq-7000 SoC [35] [36], therefore this process is not presented in details. The discussion focuses on the changes needed by APs based on the standalone BSP to correctly run a FreeRTOS instance on both cores of the CA9 processor.

Configuring an application in AMP on the Zynq-7000 SoC requires creating two APs in a single system project, each targeting a different domain, and then loading them into the PS. This operation can be performed by programming the flash memory embedded in the Pynq-Z2 with a bootable image of the system project. Furthermore, the AP running on CPU1 must define the compilation flag `USE_AMP=1` to prevent the BSP's boot code from configuring shared resources (e.g., L2 cache) that have already been initialized by CPU0.

As explained in Chapter 4, the developed application implements a producer-consumer problem using two tasks, a queue, and, optionally, a higher-priority task. In terms of memory, each FreeRTOS instance requires a private address space, but the queue must be accessible by both cores. Therefore, the default memory layout is modified in the `lscript.ld` files of both APs, as displayed by

Listing C.1. In particular, each core privately allocates 255MB of DDR memory (`ps7_ddr_0`), as well as a shared 1MB region (`shared_ddr`) to host common data. Listing C.2 shows that `shared_ddr` is populated at runtime with the memory section `.shareable_mem`, which defines the labels to reference the queue.

Listings C.3 and C.4 report the application code, respectively, of CPU0 and CPU1. It is possible to notice that `__queue` is used to share the handle returned after the creation of the queue in the `main()` function of CPU0. Furthermore, since in AMP this operation is performed using statically allocated memory, the additional data structures `__queue_storage` and `__queue_buffer` are defined, according to the description of `xQueueCreateStatic()` in `queue.h`. The support for static allocation must be enabled in `FreeRTOSConfig.h` through the `configSUPPORT_-STATIC_ALLOCATION` macro.

Regarding the dual-core setup, as described for SMP mode in Subsection 3.3.1, CPU0 must wake up CPU1 following the procedure provided in the Zynq-7000 SoC TMR [4, p. 160]. However, in AMP, the memory attributes need to be manually assigned, since the memory layout is not standard. For this reason, both cores use the function `Xil_SetTlbAttributes()` to properly set up the shared region, as it can be observed at lines 50 and 26. The used attributes configure the memory as shareable and non-cacheable.

Finally, placing the shared region in DDR and using the previously described memory attributes should also allow to target locations within `.shareable_mem` with exclusive access. Therefore, the spinlock presented in Subsection 2.5.2 can be exploited when mutual exclusion is not implicit.

```
/*** CPU0 ***/
MEMORY
{
   ps7_ddr_0 : ORIGIN = 0x100000, LENGTH = 0xFF00000
   ps7_qspi_linear_0 : ORIGIN = 0xFC000000, LENGTH = 0x1000000
   ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
   ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
   shared_ddr : ORIGIN = 0x1FF00000, LENGTH = 0x100000
}

/*** CPU1 ***/
MEMORY
{
   ps7_ddr_0 : ORIGIN = 0x10000000, LENGTH = 0xFF00000
   ps7_qspi_linear_0 : ORIGIN = 0xFC000000, LENGTH = 0x1000000
   ps7_ram_0 : ORIGIN = 0x0, LENGTH = 0x30000
   ps7_ram_1 : ORIGIN = 0xFFFF0000, LENGTH = 0xFE00
   shared_ddr : ORIGIN = 0x1FF00000, LENGTH = 0x100000
}
```

Listing C.1: Memory layout of the benchmark program in AMP configuration

```
.shareable_mem (NOLOAD) : {
   __shareable_start = .;
   __queue = .; /* Queue handle */
   . += 0x4;
   __queue_storage = .;
   . += 0x4; /* Queue of 1 element */
   __queue_buffer = .; /* Queue handler */
   . += 0x50;
   __shareable_end = .;
} > shared_ddr
```

Listing C.2: Shared memory section definition in AMP configuration

```c
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
/* Xilinx includes. */
#include "xil_printf.h"
#include "xil_mmu.h"
#include "xil_io.h"
#include <sleep.h>
#include "gtimer.h"

#define sev()                       __asm__("sev")
#define ARM1_STARTADR               0xFFFFFFF0
#define ARM1_BASEADDR               0x10000000
#define cleanToPoU()                __asm__ volatile(                   \
                                    "dsb                            \n" \
                                    "mcr p15, 0, %0, c7, c10, 1     \n" \
                                    "dsb                            \n" \
                                    ::"r" (ARM1_STARTADR): "memory");

/*-----------------------------------------------------------*/

static void prvRxTask( void *pvParameters );

/*-----------------------------------------------------------*/

/* FreeRTOS variables. */
TaskHandle_t xRxTask;
QueueHandle_t xQueue;
/* Shared variables. */
extern uint32_t __shareable_start;
extern uint32_t __queue;
extern uint32_t __queue_storage;
extern uint32_t __queue_buffer;

int main( void )
{
    /* Leave time to connect to the Serial interface */
    sleep(5);

    /* Setup global timer. */
    GlobalTimer_Stop();
    GlobalTimer_SetCounter(0);
    GlobalTimer_SetPrescaler(0);
    GlobalTimer_Start();

    /* Sets 1MB of DDR as Shareable, Non-cacheable, Full Access
    since each L1 Page Table entry is a section covering 1MB. */
    // S=b1 TEX=b100 AP=b11, Domain=b1111, C=b0, B=b0
    Xil_SetTlbAttributes((uint32_t) &__shareable_start,0x14de2);

    /* Create FreeRTOS objects. */
    xTaskCreate( prvRxTask,
                 ( const char * ) "Rx",
                 configMINIMAL_STACK_SIZE,
                 NULL,
                 tskIDLE_PRIORITY + 1,
                 &xRxTask );

    configASSERT( xRxTask );

    xQueue = xQueueCreateStatic(1,
                                sizeof( uint32_t ),
                                ((uint8_t*)  &__queue_storage),
                                ((StaticQueue_t*) &__queue_buffer) );

    configASSERT( xQueue );
    /* Pass the queue handle to CPU1. */
    *((uint32_t*)&__queue) = (uint32_t) xQueue;

    /* Start CPU1 before continuing. */
    Xil_Out32(ARM1_STARTADR, ARM1_BASEADDR);
    cleanToPoU();   /* Clean address to PoU */
    sev();          /* Wake up CPU1 */

    /* Start the scheduler. */
    vTaskStartScheduler();

    for( ;; );
}
/*-----------------------------------------------------------*/
static void prvRxTask( void *pvParameters )
{
    uint32_t ulRxData = 0;
    for( ;; )
    {
        xQueueReceive(   xQueue, &ulRxData, 0 );

        if( ulRxData == 10000 )
        {
            GlobalTimer_Stop();
            xil_printf("Higher 32 bits: %u \r\n", (unsigned int) (GlobalTimer_GetCounter_High32()));
            xil_printf("Lower 32 bits: %u \r\n", (unsigned int) (GlobalTimer_GetCounter_Low32()));
            break;
        }
    }

    vTaskDelete(NULL);
}
```

Listing C.3: Benchmark program for CPU0 in AMP

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
/* Xilinx includes. */
#include "xil_mmu.h"
#include "gtimer.h"

/*-----------------------------------------------------------*/

static void prvTxTask( void *pvParameters );
static void prvBusyTask( void *pvParameters );

/*-----------------------------------------------------------*/

/* FreeRTOS variables. */
static TaskHandle_t xTxTask;
static TaskHandle_t xBusyTask;
QueueHandle_t xQueue;
/* Shared variables. */
extern uint32_t __shareable_start;
extern uint32_t __queue;

int main()
{

    Xil_SetTlbAttributes((UINTPTR) &__shareable_start,0x14de2);

    /* Set Queue handle. */
    xQueue = (QueueHandle_t) __queue;

    xTaskCreate(prvTxTask,
                ( const char * ) "Tx",
                configMINIMAL_STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY + 1,
                &xTxTask );

    xTaskCreate(prvBusyTask,
                ( const char * ) "Busy",
                configMINIMAL_STACK_SIZE,
                NULL,
                tskIDLE_PRIORITY + 2,
                &xBusyTask );

    configASSERT( xTxTask );
    configASSERT( xBusyTask );

    vTaskStartScheduler();
    for( ;; );

    return 0;
}

/*-----------------------------------------------------------*/
static void prvTxTask( void *pvParameters )
{
    UBaseType_t uxResult;
    for(uint32_t ulTxData = 0 ; ulTxData <= 10000 ; )
    {
        uxResult =  xQueueSend( xQueue, &ulTxData, 0 );
        ulTxData += (uxResult == pdTRUE)? 1:0;
    }

    vTaskDelete(NULL);
}

/*-----------------------------------------------------------*/

```

```
static void prvBusyTask( void *pvParameters )
{
  volatile uint32_t* pulTrash;
  for(uint32_t ulIndex=0; ulIndex<1000000 ; )
  {
    pulTrash = (volatile uint32_t*) &ulIndex;
    (*pulTrash) += 1;
  }

  vTaskDelete(NULL);
}
```

Listing C.4: Benchmark program for CPU1 in AMP

Bibliography

[1] Bryon Moyer. *Real World Multicore Embedded Systems*. Newnes, 2013.

[2] FreeRTOS. *FreeRTOS Documentation*. Website. Accessed: 2025-06-22. URL: `https://www.freertos.org/Documentation/00-Overview`.

[3] Nico De Witte, Robbie Vincke, Sille Van Landschoot, Eric Steegmans, and Jeroen Boydens. «Comparing dual-core SMP/AMP performance on a telecom architecture». In: *Annual Journal of electronics* (2013).

[4] AMD. *Zynq 7000 SoC Technical Reference Manual (UG585)*. v1.14. Advanced Micro Devices (AMD). June 2023. URL: `https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/`.

[5] AMD. *Zynq 7000 SoC Software Developers Guide (UG821)*. v13.0. Advanced Micro Devices (AMD). Sept. 2023. URL: `https://docs.amd.com/r/en-US/ug821-zynq-7000-swdev/`.

[6] ARM. *ARM Cortex-A Series Programmer's Guide (ARM DEN 0013)*. Version 4.0. Advanced RISC Machine (ARM). Jan. 2014. URL: `https://developer.arm.com/documentation/den0013/latest/`.

[7] ARM. *PrimeCell Generic Interrupt Controller (PL390) Technical Reference Manual (ARM DDI 0416)*. Revision B. Advanced RISC Machine (ARM). Nov. 2009. URL: `https://developer.arm.com/documentation/ddi0416/latest/`.

[8] ARM. *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition (ARM DDI 0406)*. Revision C. Advanced RISC Machine (ARM). Nov. 2011. URL: `https://developer.arm.com/documentation/ddi0406/c/`.

[9] ARM. *Cortex-A9 Technical Reference Manual (ARM DDI 0338)*. Revision G. Advanced RISC Machine (ARM). July 2011. URL: `https://developer.arm.com/documentation/ddi0388/g/`.

[10] ARM. *Cortex-A9 MPCore Technical Reference Manual (ARM DDI 0407)*. Revision G. Advanced RISC Machine (ARM). July 2011. URL: `https://developer.arm.com/documentation/ddi0407/g/`.

[11] FreeRTOS. *FreeRTOS*. Accessed: 2025-07-08. URL: `https://www.freertos.org/`.

[12] FreeRTOS. *FreeRTOS-Kernel*. Github repository. Accessed: 2025-06-22. URL: `https://github.com/FreeRTOS/FreeRTOS-Kernel/`.

[13] XMOS. *FreeRTOS SMP Change Description*. Accessed: 2025-06-22. XMOS, Jan. 2020. URL: `https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/smp/design`.

[14] *SMP Porting Checklist - A53 *4 as reference*. Accessed: 2025-06-22. URL: `https://forums.freertos.org/t/smp-porting-checklist-a53-4-as-reference/14499`.

[15] FreeRTOS. *Using FreeRTOS on ARM Cortex-A9 Embedded Processors that incorporate a Generic Interrupt Controller (GIC)*. Website. Accessed: 2025-06-26. URL: `https://www.freertos.org/Using-FreeRTOS-on-Cortex-A-Embedded-Processors`.

[16] Xilinx. *embeddedsw*. Github repository. Accessed: 2025-06-22. URL: `https://github.com/Xilinx/embeddedsw`.

[17] FreeRTOS. *FreeRTOS*. Github repository. Accessed: 2025-06-22. URL: `https://github.com/FreeRTOS/FreeRTOS`.

[18] Matteo Fragassi (Matth9814). *FreeRTOS 11.2.0 for Zynq7000 platforms*. Github repository. Accessed: 2025-06-22. URL: `https://github.com/FreeRTOS/FreeRTOS-Kernel-Community-Supported-Ports/tree/main/GCC/CORTEX_A9_Zynq7000`.

[19] Matteo Fragassi (Matth9814). *Zynq7000 Test Demo*. Github repository. Accessed: 2025-07-08. URL: `https://github.com/FreeRTOS/FreeRTOS-Community-Supported-Demos/tree/main/CORTEX_A9_Zynq7000_GCC`.

[20] Cathal McCabe. *Tutorial: Creating a new hardware design for PYNQ*. AMD PYNQ Support. Accessed: 2025-06-22. URL: `https://discuss.pynq.io/t/tutorial-creating-a-hardware-design-for-pynq/145`.

[21] Xilinx. *Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)*. v2021.2. Accessed: 2025-06-22. Xilinx. Dec. 2021. URL: `https://docs.amd.com/r/2021.2-English/ug1400-vitis-embedded/`.

[22] ARM. *ARM GNU Toolchain*. Accessed: 2025-07-09. URL: `https://developer.arm.com/downloads/-/gnu-a`.

[23] *DMB for data cache maintenance*. ARM Community. Accessed: 2025-07-02. URL: `https://community.arm.com/support-forums/f/architectures-and-processors-forum/56346/dmb-for-data-cache-maintenance`.

[24] ARM. *Cortex-A9 MPCore Technical Reference Manual (ARM DDI 0407).* Revision I. Advanced RISC Machine (ARM). June 2012. URL: `https://developer.arm.com/documentation/ddi0407/i/`.

[25] ARM. *Arm Generic Interrupt Controller v1.0 Architecture Specification (ARM IHI 0048).* Revision A. Advanced RISC Machine (ARM). Sept. 2008. URL: `https://developer.arm.com/documentation/ihi0048/a/`.

[26] Percepio. *Getting Started with Percepio View on FreeRTOS.* Website. Accessed: 2025-07-08. URL: `https://traceviewer.io/getting-started-freertos-view/`.

[27] Percepio. *Percepio Tracealyzer: FreeRTOS.* Website. Accessed: 2025-07-08. URL: `https://percepio.com/getstarted/latest/html/freertos.html`.

[28] Matteo Fragassi. *Cortex-A9 port: disable interrupts before writing to ICC_PMR*. Accessed: 2025-07-08. URL: `https://forums.freertos.org/t/cortex-a9-port-disable-interrupts-before-writing-to-icc-pmr/22952`.

[29] Wikipedia. *uart).* Accessed: 2025-07-09. URL: `https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter`.

[30] Percepio. *Multicore Tracing on FreeRTOS 11 and TI AM62x.* Website. Accessed: 2025-07-08. URL: `https://percepio.com/multicore-tracing-on-freertos-11-and-ti-am62x/`.

[31] Percepio. *Adapting Tracealyzer to the Nios II soft-core CPU.* Website. Accessed: 2025-07-08. URL: `https://percepio.com/devblog-adapting-tracealyzer-nios-ii/`.

[32] Matteo Fragassi. *FreeRTOS SMP: port testing.* Accessed: 2025-07-09. URL: `https://forums.freertos.org/t/freertos-smp-port-testing/23148`.

[33] Mouser Electronics. *PYNQ-Z2 Reference Manual.* v1.0. Mouser Electronics. May 2018. URL: `https://www.mouser.com/datasheet/2/744/pynqz2_user_manual_v1_0-1525725.pdf`.

[34] *Use of UART on PYNQ-Z2.* AMD PYNQ Support. Accessed: 2025-06-22. URL: `https://discuss.pynq.io/t/use-of-uart-on-pynq-z2/2523/`.

[35] Whitney Knitter. *Dual ARM Hello World on Zynq Using Vitis.* Accessed: 2025-07-09. URL: `https://www.hackster.io/whitney-knitter/dual-arm-hello-world-on-zynq-using-vitis-9fc8b7`.

[36] John McDougall. *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors.* Accessed: 2025-07-09. URL: `https://docs.amd.com/v/u/en-US/xapp1079-amp-bare-metal-cortex-a9`.