TÉCNICO LISBOA

Paravirtualization of a Real Time Operating System

Development of the AIR hypervisor with RTEMS for ARM

Carolina Pinto dos Santos Serra

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors: Prof. João Nuno de Oliveira e Silva
Eng. Daniel Silveira

Examination Committee

Chairperson: Prof. Paulo Jorge Coelho Ramalho Oliveira
Supervisor: Prof. João Nuno de Oliveira e Silva
Member of the Committee: Prof. Luís Manuel Antunes Veiga

October 2020

Acknowledgments

Firstly, I would like to thank GMV for welcoming me into the company. I thank Daniel Silveira for the constant guidance and the AVOS team for the patience and unceasing support.

To João Silva, I thank for guiding me in the conception of this document.

To my family and friends, I am immensely grateful that I have a group of people that I can trust to support me in everything I do. To my parents, for helping me unconditionally. To my brother, João, who has inspired and helped me throughout all my education, and gave me valuable tips and feedback without which this dissertation would not be the same. To Tiago Alexandre, for his daily support and for being at my side throughout both fun and hard moments. To my grandfather, who did not live to see me finish my Master's degree, but whose endless curiosity for space, physics and mathematics will continue to inspire me.

Finally, to all the amazing women that surround me. Mom, grandmothers, Mónica, Ana, Mariana, Marta Duarte, Sofia, Joana, Marta Vilela, Sara, Yolanda, Mel, every single one of you inspires me beyond words and I am grateful that I had you by my side while I was working on this project. Thank you.

Resumo

Com o crescente interesse em transpor o conceito de sistemas aviónicos modulares integrados da aviação para a indústria espacial, a GMV desenvolveu o AIR, um hipervisor que permite um único computador executar múltiplas aplicações e sistemas operativos, mantendo rígida segregação temporal e espacial através de virtualização.

Tendo sido originalmente desenvolvido para a arquitetura SPARC, as recentes propostas de utilização de ARM em missões espaciais levaram ao início da migração de AIR para esta arquitetura. Foi anteriormente desenvolvido o suporte de AIR para a placa Arty Z7 baseada nos sistemas integrados Zynq-7000 da Xilinx, capaz de executar aplicações com um sistema operativo barebones, destinado apenas a testar as funcionalidades básicas do hipervisor.

Esta dissertação dá continuidade ao trabalho de migração de AIR para ARM através da virtualização de RTEMS, o sistema operativo em tempo real presentemente adotado pela ESA e pela NASA e elegido nas suas futuras missões. A virtualização deste sistema operativo para AIR permite que o hipervisor suporte todas as aplicações desenvolvidas para RTEMS, introduzindo o conceito de Time and Space Partitioning em presentes e futuras missões espaciais.

Foi alcançada com sucesso uma versão melhorada de AIR para ARM capaz de suportar RTEMS, permitindo executar testes mais complexos do que anteriormente possível. Foram executados 21 testes de validação, assim como estudos comparativos a nivel funcional e temporal entre RTEMS original e virtualizado, e entre AIR para ARM e para SPARC, que demonstraram a correta operação das funcionalidades do hipervisor.

Palavras-chave: IMA, AIR, RTEMS, ARM, virtualização, hipervisor, RTOS

Abstract

With the increasing interest in transposing the concept of integrated modular avionics from aviation to the space industry, GMV developed AIR, a hypervisor that allows a single computer to run multiple applications and operating systems, maintaining strict temporal and spatial segregation through virtualization.

Having originally been developed for the SPARC architecture, recent proposals for the use of ARM in space missions have led to the beginning of the migration of AIR to this architecture. It has been previously developed the support of AIR for the Arty Z7 board based on Zynq-7000 SoC by Xilinx, capable of running applications with a barebones operating system, designed to test the basic functionalities of the hypervisor.

This dissertation continues the migration of AIR to ARM through the virtualization of RTEMS, the real time operating system presently adopted by ESA and NASA and elected for their future missions. The virtualization of this operating system for AIR allows the hypervisor to be used for every application developed for RTEMS, introducing Time and Space Partitioning to current and future space missions.

It was successfully accomplished an improved version of AIR for ARM capable of supporting RTEMS, enabling the execution of tests of higher complexity than previously possible. 21 validation tests were executed, as well as comparative studies at functional and temporal levels between the original and the virtualized RTEMS, and between AIR for ARM and AIR for SPARC, which demonstrated the correct operation of the hypervisor functionalities.

Keywords: IMA, AIR, RTEMS, ARM, virtualization, hypervisor, RTOS

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since it was first implemented in the cockpit functions of the Boeing 777 in 1995, the concept of Integrated Modular Avionics (IMA) has developed to be a vastly important and successful achievement in the aviation field [1]. This architecture provides fault containment and system portability without compromising the weight, volume and cost of the avionic system by centralizing the computing capacity into a single processing unit. In an effort to standardize the access to shared resources, the Airlines Electronic Engineering Committee (AEEC) has published a series of specifications, among which the ARINC 653 specification was defined [2]. It outlines the properties of the operating system (OS) that manages all the other software functions.

The success of IMA in aviation caught the attention of the space community, and under the efforts to transpose this concept into space avionics, GMV[1] developed AIR [3], a Real Time Operating System (RTOS) that implements the ARINC 653 standard. AIR allows several safety-critical applications to run on the same processor while maintaining strict temporal and memory segregation through Time and Space Partitioning (TSP). AIR was originally developed for SPARC, the architecture adopted by the European Space Agency (ESA) for a large majority of space missions since the early 90's. Using SPARC, AIR has been integrated in various space projects, including several activities commissioned by ESA, a Global Navigation Satellite System (GNSS) receiver [4] and ESROCOS, a Robot Control Operating Software for space robotics applications [5].

While SPARC is still the dominant processor architecture in the European space industry today, the traction that ARM processors gained in the Sistem on a Chip (SoC) market over the last decades and the higher performances achievable through this architecture led to an increasing number of proposals for the use of ARM based SoCs in space. It is expected that the relevance of ARM in the space sector will increase over the following years, and this prediction prompted GMV to migrate AIR to this architecture. In 2019, GMV started the development of an ARM processor's Board Support Package (BSP) for AIR, the software layer that acts as an interface between an embedded system's hardware and the OS. A simplified depiction of the structure of AIR running a single application can be seen in Figure 1.1. The newly implemented BSP is targeted at an Arty Z7 board based on Zynq-7000 SoC by Xilinx, featuring a

[1] `https://www.gmv.com/`

dual-core ARM Cortex-A9 processor [6].

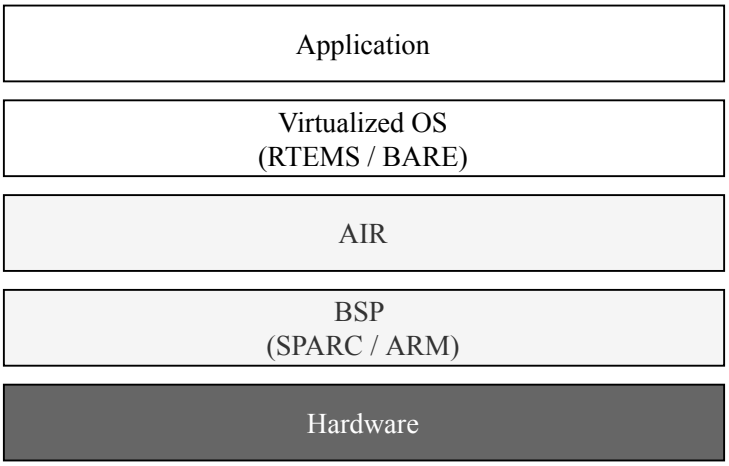

Figure 1.1: Simplified structure of AIR running a single application.

The migration of AIR to the ARM architecture broadens the scope of missions in which AIR can be applied. AIR is planned to be deployed in INFANTE, the first satellite to be developed exclusively by the Portuguese Industry [7]. The INFANTE project, along with the emergence of the Portugal Space Agency in 2019 [8], marks a new age of opportunities for the space sector in Portugal.

1.1 Motivation and Objectives

AIR is a hypervisor, meaning a software layer that allows several OSs to run on the same hardware through virtualization. In order to be able to run on top of AIR, as represented in Figure 1.1, an OS must be virtualized to perform all accesses to the hardware through the hypervisor instead of directly accessing the processor registers. Currently, AIR for ARM has only been tested with a simple virtualized barebones OS, created specifically to test the basic functionalities of the hypervisor and with limited capabilities. Providing a virtualized RTOS will not only allow features of higher complexity to be implemented, but also enable further testing to be performed on AIR for ARM, raising the opportunity to fix any issues that might have gone undetected so far and ensure the correct functioning of the hypervisor.

There are two possible approaches to this goal: a) Create a virtualized OS from scratch, implementing the typical RTOS services, or b) Virtualize an existing RTOS, such as RTEMS. The second approach was the one chosen, as it offers several advantages:

- it is simpler, allowing the reuse of code of an RTOS which has already been developed and tested;
- since RTEMS has already been virtualized for AIR for SPARC, the virtualization for ARM facilitates the portability of applications between the two architectures, allowing the same program to run on both ARM and SPARC by simply configuring a different target BSP;
- RTEMS is a recognized tool with valuable features, used in several space projects, and supporting this RTOS will be a great benefit for AIR.

RTEMS stands for Real Time Executive for Multiprocessor Systems. It is an open source RTOS with an active global community available for online support [9] that is currently in the process of qualification for space applications [10]. RTEMS supports multiple processor architectures, including all the processors currently used by ESA and The National Aeronautics and Space Administration (NASA), and provides many relevant features such as multitasking capabilities, intertask communication and synchronization protocols, priority-based preemptive scheduling and interrupt management [11]. The most recent upgrades also implement Symmetric Multiprocessing (SMP) support. While the presence of multiple cores with SMP brings significant performance improvements, some challenges may arise from the increased complexity of the system. The true concurrency achievable by multiple cores executing in parallel makes race conditions more likely to occur. RTEMS offers task management services that can provide support for using SMP systems to their maximum capability. These features and the extensive testing that it has been subjected to led to RTEMS being a reliable and robust operating system, and the most prevalent RTOS in space for decades [12], having been deployed in numerous NASA and ESA missions.

The virtualization of RTEMS will elevate the potential of AIR for ARM as it allows it to support all of the space applications that have been developed for RTEMS, making it suitable for integration in a variety of high profile space missions. In this prospect, the goal of this dissertation is to virtualize RTEMS, allowing it to run on AIR for ARM, and use the newly virtualized RTOS to further test and validate the AIR hypervisor.

1.2 Approach and Results

This work accomplished to further develop the value of AIR for ARM by enabling it to run RTEMS applications. To achieve this milestone, modifications to both RTEMS and AIR for ARM were needed, and were performed through an iterative work flow.

To allow RTEMS to run on the AIR hypervisor, it is necessary to perform virtualization methods to prevent it from accessing the real processor registers. Of the existing virtualization methods, the one elected was paravirtualization, in which the guest OS is modified to replace these critical accesses by system calls to the hypervisor. This way, the hypervisor performs the desired action instead of the guest OS, and then returns to the normal execution. A correctly paravirtualized OS should behave as would the same OS when running natively on the hardware. To ensure this, as well as to mantain the reliability that RTEMS has achieved through years of testing, it is important to limit the changes to the RTEMS code to a bare minimum.

The execution of RTEMS applications on AIR for ARM highlighted flaws of the original migration of the hypervisor from the SPARC architecture that had previously been undetected from the testing with a barebones OS. These flaws are mostly due to the architectural differences in the exception handling, and require modifications to the hypervisor code.

The target processor of this dissertation is the ARM Cortex-A9 processor integrated in XILINX's Zynq-7000 series SoCs, and the software was developed in the Diligent's Arty Z7 board provided by GMV. At

the end of this dissertation, AIR for ARM is capable of running single core RTEMS applications. The virtualized OS was tested through running the RTEMS Testsuites and comparing the RTEMS behaviour to the RTOS running in a non virtualized environment. By incorporating these tests in the development process it was possible to check that every single modification applied to RTEMS achieved the expected result. Then, AIR examples and validation tests were used to further test the hypervisor. AIR examples allow the verification of basic hypervisor functionalities, such as TSP, exception handling and recovery from faults. The AIR validation tests are a set of tests designed to achieve a nearly total code coverage of AIR for SPARC. Of 38 validation tests currently running on AIR for SPARC, 21 have been successfully employed on AIR for ARM, presenting the same results in both supported architectures.

Once the virtualization of RTEMS is extended to multicore and AIR for ARM is subjected to more extensive testing, GMV will have the opportunity to compete in a larger number of proposals for space projects and missions, ultimately leading AIR to an increased market value. The MIURA project is an example of a system that will benefit from the advantages of AIR, and the first space mission in which the work resulting from this dissertation will be implemented. The avionics for MIURA are being developed by GMV, and the implementation of AIR could significantly decrease the weight, volume and cost of the launcher while providing reliable isolation and fault containment.

1.3 Thesis Outline

This thesis is divided into 6 chapters. Chapter 1 starts by introducing the setting under which this work takes place, the motivation that led to it and the main goals it will attempt to achieve. Chapter 2 presents the avionics concepts that are relevant for this dissertation, providing the theoretical background to understand the work ahead. Chapter 3 details the most important features of each of the technologies that were the platform for this thesis. Chapter 4 covers the implementation, both the modifications to the RTEMS code in order for it to run on top of AIR, and the changes that had to be applied to AIR itself. Chapter 5 presents the testing environment and the results obtained from the tests performed to validate the migration to ARM and the virtualization of RTEMS. Finally, the conclusions and future work are presented in Chapter 6.

Chapter 2

Avionics Concepts

This chapter goes through avionics concepts that are fundamental to understand this thesis. Section 2.1 starts by introducing the federated architecture for avionic systems, and the problems that led to the demand of an alternative architecture. Section 2.2 offers virtualization as a possible solution, giving a brief historical overview of the concept as well as the most common virtualization procedures. Section 2.3 discusses Integrated Modular Avionics, and the context that led to its deployment in the space sector. Section 2.4 approaches Real Time Operating Systems, and their relevance in space avionics. Finally, Section 2.5 presents the challenges of sending processors to space, and the CPUs that overcome those challenges.

2.1 Federated Architecture

Since radios for communication and navigation became the first avionic devices to be applied to military aircrafts in the decade of 1910, the increasing demand for safer, more dependable and higher quality systems limited by economic constraints led to more functionalities being performed by electronic controllers rather than mechanical equipment. Moore's Law states that processor speeds and capabilities double approximately every two years, and this phenomenon can be observed in the evolution of the performance of aviation electronics over the last five decades [13].

In the avionics system's federated architecture, each function is assigned to one computer, isolating the hardware into black boxes named line-replaceable units (LRUs) [14]. This organization is represented in Figure 2.1. This modular design offers isolation between avionic functions as the fault management is performed at the LRU level, avoiding fault propagation and easing recovery procedures. However, the increasing complexity of avionics systems demands a progressively higher number LRUs to be employed, and each hardware module adds weight and volume to an aircraft, eventually meeting envelope restrictions. Another disadvantage is higher power consumption and increased cost [15]. In the early 90's, these drawbacks led to a demand for an alternative architecture that could centralize the computing resources without compromising the fault containment. The research of virtualization by the computing community would provide a solution.

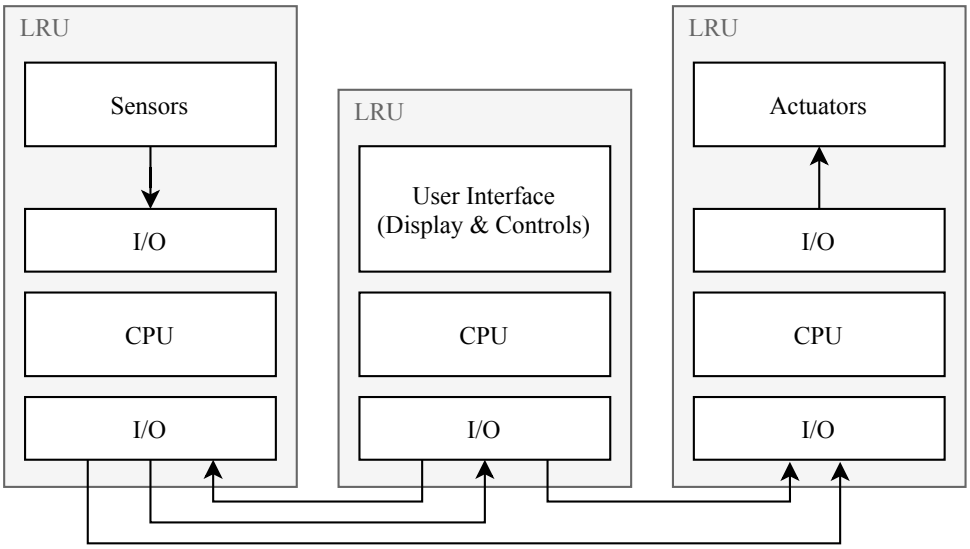

Figure 2.1: Possible federated architecture organization.

2.2 Virtualization

The research for virtual machines (VMs) started in the late 1960s with the purpose of allowing the sharing of computer resources among several users through time-sharing solutions [16].

Virtualization refers to the separation of a service request from the underlying hardware that delivers that service. Through virtualization, multiple VMs can operate in a single computer, managed by the Virtual Machine Monitor (VMM). Each virtual machine is an efficient and isolated duplicate of a real machine, capable of running its own OS [17]. This architecture made it possible to overcome the limitations of conventional processors, that are characterized by a dual state hardware organization, which allows for two modes of operation:

- **Supervisor**, a privileged mode with access to all instructions;
- **User**, a non-privileged mode with access to a subset of the instructions.

When a non-privileged application attempts to execute a privileged instruction, the program is interrupted by an exception, being redirected to a table of exception handlers. To be able to perform privileged actions, an application running on user mode resorts to system calls to the privileged software, which performs the desired action for them if the application has the required permissions of the functionality [18]. The subset of user instructions plus the supervisor calls form the extended machine. This organization can be seen in Figure 2.2.

This architecture is successful in plenty computer systems. However, it has limitations: since there is only one bare machine interface, only one OS can run at a time, and software that requires direct access to privileged instructions is unable to be transposed into this architecture [19].

With the introduction of virtualization, the VMM runs at a higher privilege level than the OS, which would usually run at supervisor mode. For this reason, the VMM can also be referred to as the Hypervisor. Some literature distinguishes the two terms, using VMM to refer to the software responsible for

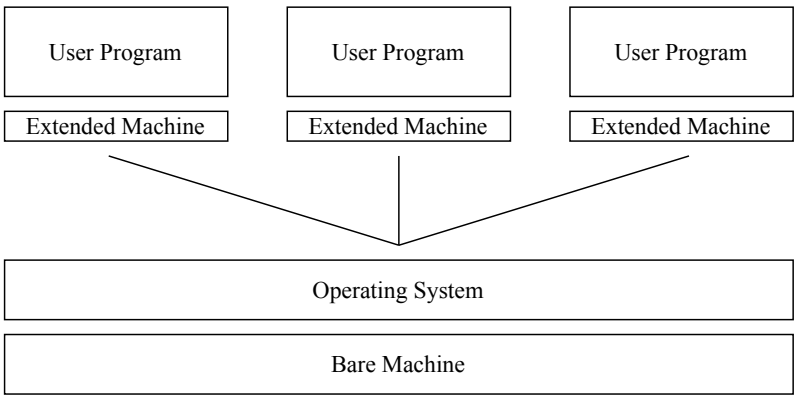

Figure 2.2: Conventional extended machine organization.

virtualizing an architecture and hypervisor as the junction of a VMM with an operating system. For this dissertation, the isolation of a VMM from its OS is not meaningful, and therefore both the terms will be used as synonyms.

The virtual machine organization is represented in Figure 2.3.

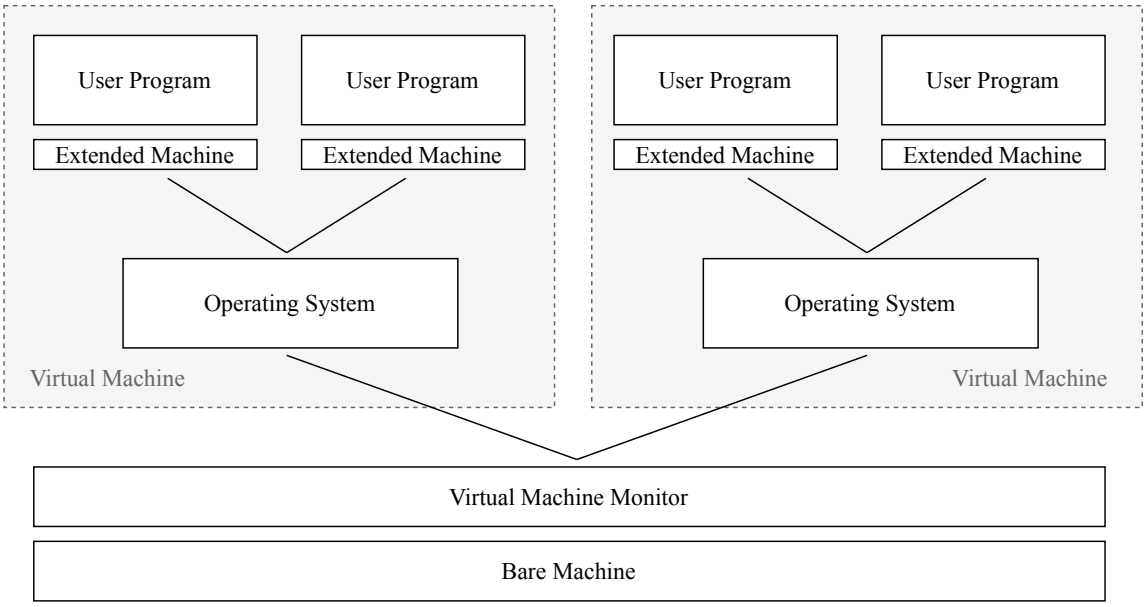

Figure 2.3: Virtual Machine organization.

In both conventional and virtualized architectures, the instructions can be categorized into sensitive or innocuous. **Sensitive instructions** are both the instructions that attempt to change the processor state (control sensitive instructions) and the instructions whose execution is dependent on that state (behavior sensitive instructions). If an instruction is neither control sensitive nor behavior sensitive, then it is considered **innocuous**.

In 1974, Gerald J. Popek and Robert P. Goldberg introduced a set of three conditions to determine whether a computer architecture supports efficient virtualization [20]:

- **Efficiency**: The vast majority of the instructions should run with no intervention by the VMM. This means that all innocuous instructions should be executed directly by the hardware.
- **Resource Control**: The VMM should hold complete control over the system's virtualized resources.
- **Equivalence**: The behavior of an application running on top of a VMM should be identical to the behavior of that same application running with no VMM. The only exceptions to this condition are the timings of execution and the system resources availability

In order to meet the first two conditions, all the sensitive instructions should be executed by the hypervisor, and therefore should be privileged instructions. However, most architectures have non-privileged, sensitive instructions, referred to as **critical instructions**. In the presence of critical instructions the Popek and Goldberg virtualization requirements cannot be met, so a different virtualization solution is needed.

2.2.1 Full Virtualization

In this method, the VMM presents each VM with a complete duplicate of the bare machine. The guest OS is not aware of the virtualization, and requires no modifications, which allows for direct portability. However, the hardware must be modified in order to support virtualization. This is usually done through hardware extensions.

The ARM V7-A Architecture implements the ARM virtualization Extensions [21], which introduce a new privilege level, higher than supervisor, for the hypervisor to operate at. The guest OS can therefore run at the privilege levels it was originally designed for. The hypervisor either traps sensitive instructions or offers a duplicate of the system information in memory to the guest OS trying to perform a critical sequence. A new level of translation is added to the Memory Management Unit (MMU) in order to allow the guest OS to perform the memory translation as it normally would and only afterwards have the hypervisor perform the final translation into physical memory. The interrupt handling is managed by the hypervisor.

When virtualization extensions are not available, full virtualization can be achieved through binary translation [22]. In this method, the hypervisor replaces the sensitive instructions by code sequences with equivalent results.

Although Full Virtualization offers isolation and security for VMs, as well as OS portability, it also represents more complex VMMs and loss of performance, especially in the case of binary translation, due to the significant initial overhead.

2.2.2 Paravirtualization

In paravirtualization, the guest OS is modified by replacing the critical instructions with hypervisor calls (HVCs) that communicate directly with the hypervisor. The HVCs interrupt the VM with a jump to

the hypervisor, which performs the service requested on their behalf, allowing information to be passed to and from the hypervisor.

This method has the advantage of a simple VMM, making it possible to achieve performance similar to the non-virtualized system [16]. However, since it can only support guest OS specifically modified for the hypervisor, the compatibility and portability is poor. It also requires higher support and maintenance costs, and is limited by the availability of the target guest OS for customization.

The goal of this dissertation is to allow the AIR hypervisor to run RTEMS as the guest OS on the ARM Cortex-A9, which does not implement the virtualization extensions that would enable hardware virtualization. As RTEMS is open-source, paravirtualization is the simplest and most favorable solution.

2.3 Integrated Modular Avionics

The concept of Integrated Modular Avionics (IMA) was presented by Honeywell in 1995, applied to the cockpit functions of the Boeing 777 [1]. It consists on centralizing the computing capacity into a single processing unit, using virtualization to separate the software functions from each other as well as from the hardware [14]. An example of a possible IMA hardware configuration is presented in Figure 2.4. Honeywell proved that IMA systems could achieve a level of reliability comparable to the federated architecture while significantly reducing the weight and volume of the avionics.

Airbus then further developed this idea by applying it to the entire design of the A380, with a set of added properties that introduced Open-IMA. With this approach, the avionic components could be manufactured by third party suppliers according to the Application Programming Interface (API) published by Airbus. Since then, the AEEC has published a series of specifications in an effort to standardize the access to shared resources in this architecture. Among them is the ARINC 653 – Avionics Application Software Standard Interface, which describes the required and recommended properties of an IMA OS [2, 23].

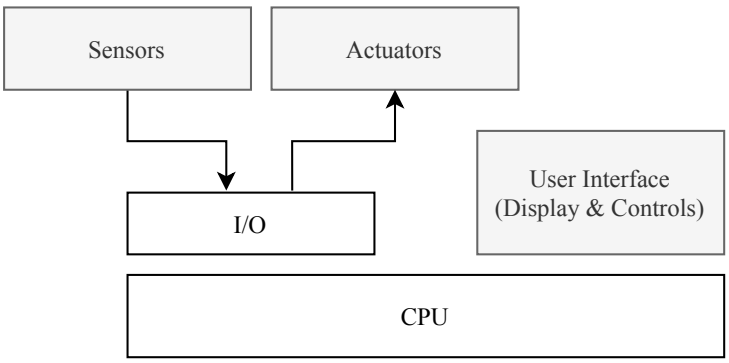

Figure 2.4: Possible IMA hardware configuration.

IMA systems are typically characterized by:

- a **high fault tolerance**, allowing a defective module to be replaced without harming the rest of the system;

- **technology transparency** due to the standardization efforts, which in turn supports **application portability**;
- **scalability**, enabling its application to systems of various sizes and supporting future growth;
- **system reconfigurability**.

These characteristics are also highly desired in space avionics, and the issues that led to the development of the IMA concept in aeronautics have been addressed by The Consultative Committee for Space Data Systems (CCSDS), through the definition of the Spacecraft Onboard Interface Services (SOIS) standard [24]. SOIS provides an abstraction of each spacecraft unit through a layered architecture, separating the mission-specific application blocks from the generic execution blocks. However, the space community could not ignore the success of IMA in the aeronautics field, and in 2010 the IMA for Space (IMA-SP) program was started [25]. In a first phase, the goal was to investigate the feasibility of an IMA-SP system. Under this initiative, GMV developed AIR, competing with pikeOS by Sysgo [26] and XtratuM by the Universitat Politècnica de València [27].

In parallel to IMA-SP, in November 2008 the Space Avionics Open Interface Architecture (SAVOIR) initiative joined together the European space community with the goal of promoting the standardization of the main avionic functions and the interfaces between them [28], and in 2012 a working subgroup focused on transposing the IMA concept to space avionics was created – SAVOIR-IMA [29]. This new initiative took over the responsibility of integrating IMA-SP into the already implemented space avionics components, and AIR was upgrated to be compliant with the IMA-SP architecture that resulted from this development.

2.4 RTOS

The development of Real Time Operating Systems started in the eighties in order to support real time applications. Real time is an extremely important concept in the computing field, referring to systems capable of assuring responses within precise timings.

A real time application is composed by a set of tasks that cooperate with each other [30]. These tasks can be **periodic**, if invoked consistently over regular periods of time, or **aperiodic**, if only activated when a certain event occurs. Most periodic tasks are also **time critical**, meaning that the task has a deadline that must be met in order for the system to function. If the task does not have a deadline, it is not time critical but it should be completed as fast as possible without compromising the deadlines of other tasks. The deadlines can be categorized into **hard**, if the consequences of missing it are catastrophic, such as life danger, equipment damage or environmental harm; **firm**, if after the deadline the results of the task are no longer useful; or **soft**, if after the deadline the results are still applicable but their value decreases over time. These deadlines are set by the application and introduce the timing constraints that are one of the main aspects that differentiate RTOS from regular OS.

A key concept in RTOS is **predictability**, meaning the degree to which the times of execution are guaranteed within minimal variations [31]. Rather than being fast, a predictable system is one whose

response times are known a priori. Predictability does not imply determinism, which refers to a fully predictable system guaranteed to achieve the same results for the same initial conditions, with no margin of uncertainty. For a system to be deterministic, it is required that the exact characteristics of all tasks and the environment conditions are known at the time of the system design. For complex embedded system such requirements are implausible, but an adequate predictability is usually enough for a robust RTOS.

The predictability of RTOS is crucial for safety-critical, real time space applications, in which a failure or delay may have catastrophic consequences. Some of these applications include the software that monitors and controls the spacecraft's health, position and response to harsh environment conditions. The selection of an RTOS for space applications should take into consideration the unique requirements of the space domain, as well as the particular constraints of each mission. One of the most important criteria in choosing the adequate RTOS is its availability for the target processor, as the exposure to radiation, vibrations and extreme temperatures demand specific hardware that can sustain those conditions [32]. The RTOS should also provide support for the necessary device drivers, and be capable of operating in an environment with severe memory and power constraints. At last, since often space projects are planned to operate during a long lifetime, an RTOS may need to offer perenniality, meaning the ability to endure the required period of time.

2.5 Space-grade Processors

The harsh environment to which hardware is subjected to during space missions demands extensive testing to ensure that it is capable of sustaining conditions such as extreme temperatures, vibrations, vacuum and radiation. While radiation was not a limiting factor in the first computing units to be sent to space, the technological advances in processor design push a trend to decrease the size of the integrated electronic circuits, making them more susceptible to the effects radiation [33]. The reason for this is that, as transistors get smaller, so is the power supply voltage reduced. A single ionizing particle can therefore change the state of the transistor and interfere with the system's behavior. This change of state is referred to as Single Event Upset (SEU) [34] and can be caused by galactic cosmic rays, solar coronal mass ejections and the radiation belt of protons and electrons that surrounds each planet [35].

To prevent the harmful effects of radiation, space-grade processors must be subject to methods of radiation hardening (rad-hard) [36]. There are multiple rad-hard techniques, both physical, consisting on manufacturing chips on isolating substrates, and logical, using redundancy to detect possibly corrupted data. Most of the modern rad-hard processors use logical methods, or Radiation Hardening By Design (RHBD), since it allows processors to be produced through standard fabrication processes, significantly reducing the costs [37].

The first European rad-hard processors were MAS281 and MA3170, both based on the MIL-STD-1750A 16-bit Instruction Set Architecture (ISA) [38]. As new space projects demanded increasingly higher performances, ESA began the investigation for 32-bit architectures that could replace the previous processors. Of the set of architectures considered, the Scalable Processor Architecture (SPARC) was

chosen for its open architecture, with no patents or license fees, and the existing software support [39]. The development of ERC32 started in 1992, successfully producing a rad-hard SPARC processor deployed in the International Space Station (ISS), the Automated Transfer Vehicle (ATV) supply truck and PROBA-1 Earth Observation microsatellite. Despite its success, the fact that ERC32 used proprietary processing cores hindered the modifications necessary to the rad-hard processes [40]. This led ESA to start the development of LEON, a new series of SPARC processors designed from scratch to incorporate fault tolerance through redundancy techniques [41]. To this day, the LEON series are still the most prevalent processors in European space missions. For this reason, AIR by GMV was developed for this architecture.

In an effort to provide better performance to future missions, in 2016 NASA launched the High Performance Spaceflight Computing (HPSC) Processor Chiplet program solicitation [42, 43]. This four year project, won by Boeing, has the goal of designing a radiation-hardened multi-core ARM processor for deep space exploration [44]. In the European Union the same goal is being pursued by DAHLIA, a project born from the collaboration of Airbus Defense and Space, ISD, NanoXplore, STMicroelectronics and Thales Alenia Space [45], in response to the Horizon 2020 topic "COMPET-1-2016: Critical Space Technologies for European Strategic Non-Dependence" [46]. The developed ARM-based SoC is expected to achieve performances 20 to 40 times higher than the existing SoC for space, and will integrate dedicated peripherals for GNSS, Telemetry and Telecontrol support. The possibility of running key space applications in a single chip will significantly reduce the cost and the mass of the system. In light of these proposals, it is expected that ARM will gain relevance in the space industry in the following years, and this prediction raised the interest of GMV to migrate AIR to the ARM architecture.

Chapter 3

Technologies

This chapter introduces the technologies that are relevant to this dissertation, starting by, in Section 3.1, describing the OS to be paravirtualized, RTEMS. Then, Section 3.2 goes through the ARINC 653 specification, followed by Section 3.3 which introduces the AIR hypervisor by GMV. Finally, Section 3.4 approaches ARM, the target architecture of this thesis.

3.1 RTEMS

The predictability of RTOS along with the possibility of coordinating a large number of concurrent tasks led the United States Army Missile Command to begin the research for a real time executive for embedded systems in 1988 [12]. The result was the Real Time Executive for Military Systems [47], later renamed Real Time Executive for Multiprocessor Systems (RTEMS).

From then onwards, RTEMS has gone through years of development, sustaining a philosophy of cooperation between users and maintainers which further elevated the open source project. In 2006 the Portuguese company EDISOFT started the development of an improved version of RTEMS that would be qualified for space deployment, RTEMS 4.8i or RTEMS by EDISOFT [48]. Meanwhile, the RTEMS community has continued to evolve the standard RTEMS, and as the qualified version is not maintained in the main RTEMS repository, space project developers face the decision of using a qualified, older version of RTEMS, or a newer version that is in the process of qualification [49]. AIR for SPARC currently supports both RTEMS 4.8i and RTEMS 5, and at the end of this dissertation, AIR for ARM supports RTEMS 5.

RTEMS supports over 160 BSPs, including the most relevant CPU architectures in European and American space missions, such as SPARC, with BSPs for ERC32 and LEON, PowerPC, MIPS, x86 and ARM [50]. It has been the chosen RTOS for several space missions, examples of which are the NASA Solar Dynamic Observatory, launched in 2010 with five radiation-hardened Coldfire CPUs running RTEMS [51], and the Electra UHF antenna of the Mars Reconnaissance Orbiter [52].

3.1.1 RTEMS Architecture

RTEMS is structured to encourage the development of modular components, as well as to isolate hardware dependent code so as to allow most of the source code to be shared across all architectures. The RTEMS conceptual architecture, presented in Figure 3.1, consists of three main layers: the hardware support layer, encompassing all the hardware dependent code, the kernel layer and the application programming interface (API) layer.

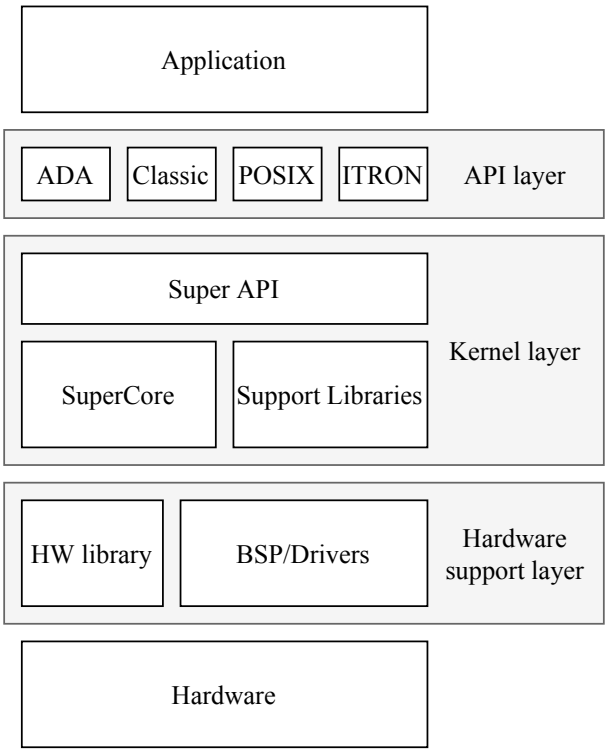

Figure 3.1: RTEMS conceptual architecture.

The kernel layer is further divided into the RTEMS SuperCore (score), the super API (sapi) and the portable support libraries. The SuperCore provides a common foundation for the upper layers, with services such as scheduling, dispatching and object management. The Super API contains code for services such as API initialization and extensions support.

The API layer connects the kernel to the application through services provided by resource managers, which are executive interfaces formed by grouping directives into logical sets. RTEMS allows the application programmer to use APIs such as Ada, POSIX, μITRON and the Classic API developed by RTEMS. The Classic API's resource managers are represented in Figure 3.2.

3.1.2 Tasks

As defined by RTEMS, a task is the smallest thread of execution which can compete on its own for system resources. RTEMS Tasks are created by the Task Manager by calling the *TASK_CREATE* service. When this service is called, RTEMS allocates Task Control Blocks (TCB), which are data structures that contain all the necessary information related to executing a task, such as the task's

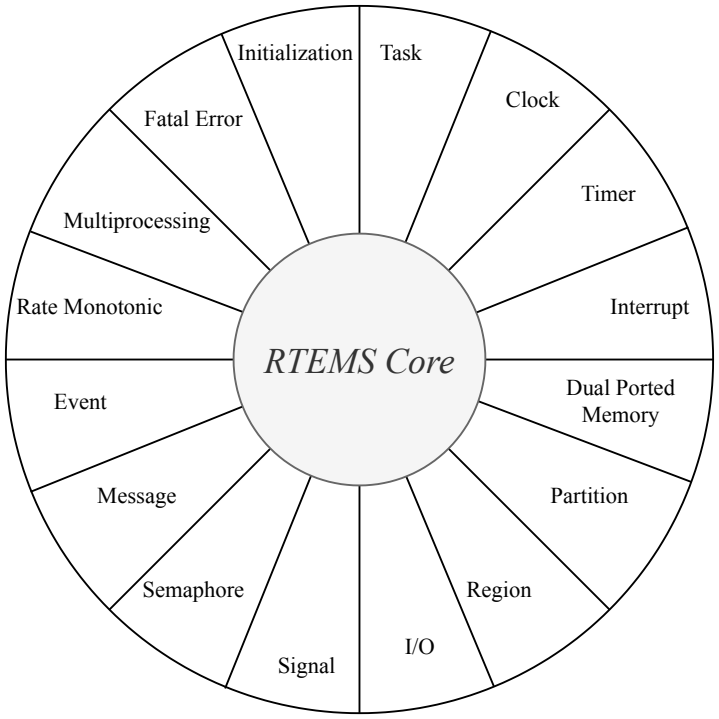

Figure 3.2: RTEMS resource managers.

name, priority and ID. The task is set at the dormant state until it is activated through the *TASK_START* routine. The operational states of the tasks and transitions between them are represented in Figure 3.3, and comprise the following:

- **Non-existent:** State before a task is created or after a task is deleted. All the TCBs are preallocated at initialization time, but are only associated with a user task at the creation service.
- **Dormant:** State after a task is created and before a task is started. In this state, a task is not able to compete for resources.
- **Ready:** State after a task is started, after a blocked task is readied or after the processor is yielded. The tasks can be allocated to the processor and the scheduler considers them for execution.
- **Executing:** State of a task that is allocated to the processor.
- **Blocked:** State after a blocking action. A blocked task is unable to be scheduled to the CPU until a readying action is taken.

RTEMS supports both periodic and sporadic tasks. When a task is created, it is sporadic by default. In order to be repeated periodically, a rate monotonic period should be created using the rate monotonic manager.

Each task has a priority level, ranging from 1 to 255, which is used by the scheduler to determine which ready task will be executed. The scheduling process can be changed through the task's mode, which can also be used to alter the execution environment. The task's mode is defined by four components:

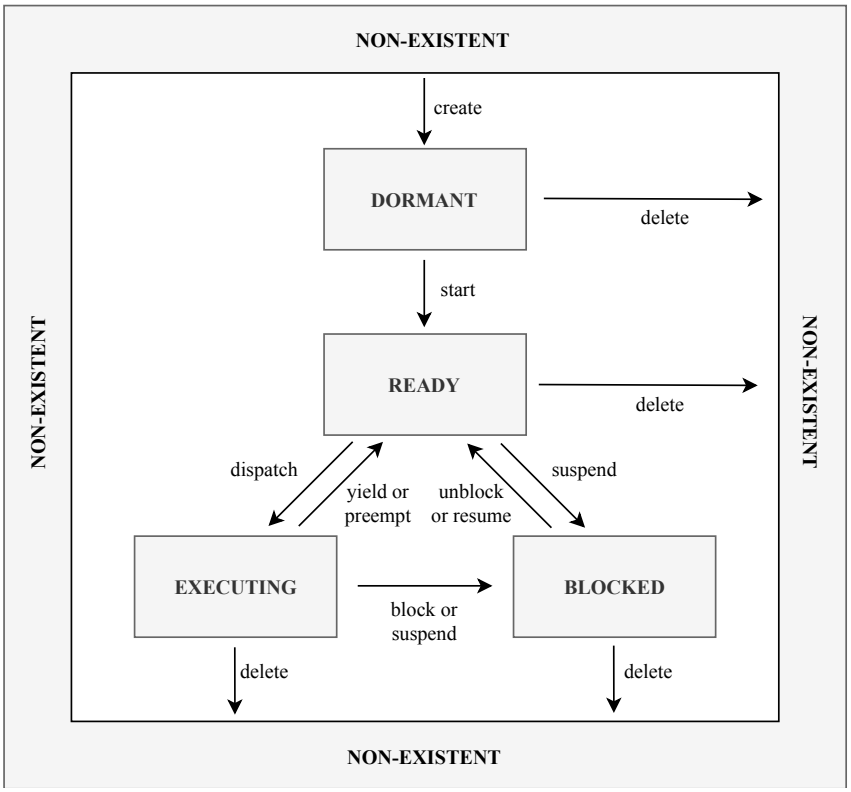

Figure 3.3: RTEMS Task State Diagram.

- **Preemption** – if enabled, the executing task will be interrupted when a higher priority task is made ready, and the control of the processor will be passed to the higher priority task; if disabled, the task will retain control of the processor for as long as it is in the executing state, even if a higher priority task is ready.

- **Asynchronous signal processing** (ASR) – This component only affects tasks that have a routine for processing asynchronous signals established. If enabled, the signals sent to the task will be processed the next time the task is executed. If disabled, the signals will only be processed when the signal processing is enabled.

- **Timeslicing** – This component is only considered by the scheduler if preemption is enabled. If timeslicing is enabled, when multiple tasks of the same priority are ready, the time that each task can hold the control of the processor is limited to the period defined in the configuration table. If disabled, the task will continue executing until a higher priority task is ready.

- **Interrupt Level** – It is used to determine which interrupts are enabled when the task is executed.

3.1.3 Scheduling

The scheduler is the component responsible to allocate the processor time to the various competing tasks. The implementation of scheduling algorithms is done through a plugin framework, which allows users to select from multiple scheduling algorithms provided by RTEMS or to implement their own.

The most common scheduling algorithm is Priority Scheduling, which allocates the processor to the ready task with the highest priority. A task that is readied is placed in the ready chain behind all of the other tasks with the same priority, being executed in a First in, First Out (FIFO) order.

There are plenty of priority-based schedulers. The default scheduler for single-core systems is the Deterministic Priority Scheduler. This is the classic RTEMS scheduling algorithm, and the only one available in RTEMS 4.10 and earlier. It is implemented through an array of FIFOs and achieves fixed and predictable execution time, hence the deterministic designation. An alternative with a lower memory footprint is the Simple Priority Scheduler, which uses a single linked list to manage all tasks. Its disadvantage is that the number of ready tasks is limited, but in systems with few tasks this does not raise performance issues. Both of these schedulers are only aware of a single core. There are alternative schedulers such as the Earliest Deadline First scheduler (EDF), which considers the deadlines declared by the Rate Monotonic Manager as priorities, and the Constant Bandwidth Server (CBS) scheduling, which assigns each task to a server characterized by a deadline (period) and a computation time (budget). The fraction of the CPU to be reserved by the scheduler for each period is calculated by the ratio budget/period (bandwidth).

For symmetric multiprocessor (SMP) configurations, the default scheduler is EDF SMP scheduler, and some alternatives include the Deterministic Priority SMP scheduler, the Simple Priority SMP scheduler and the Arbitrary Processor Affinity Priority SMP scheduler, which places the ready tasks in a table of chains with one chain for each priority level.

As mentioned in Section 3.1.2, the scheduling process can be modified by the Task mode, namely by the Preemption and Timeslicing controls. The scheduler can also be customized by the user through assigning the desired priority levels to each individual task, either when it is being created or during run-time, and through a mechanism called manual round-robin, invoked by the rtems_task_wake_after directive. When this function is called, the task is returned to the end of the ready-chain of its priority group and the control of the processor is taken to the next task of the same priority. When no other task of the same priority is ready to run, the task keeps the control of the processor.

The allocation of the processor for each task is done by the dispatcher, which retrieves the control of the processor from the current task and passes it to the next by performing a context switch. The context switch consists on saving the context of the current task and restoring the context of the task that has been allocated to the processor. The context includes all of the necessary information to ensure that the task is capable of returning to execution without suffering any change by the interruption. This information is stored either in the TCB or the task's stacks.

3.1.4 System Configuration and Build

The build system in RTEMS relies on makefiles that can be split into two types: directory and leaf. Each directory containing source code for libraries or programs is compiled through a leaf makefile, and the build is propagated to the source directory through a directory makefile.

RTEMS provides tools to assist the build process by automatically generating all the necessary

makefiles. The first step of this procedure is to bootstrap the build system. The bootstrap uses a file called configure.ac to generate the files needed to configure an application. RTEMS provides several BSPs, but not the Arty Z7 board which is the target of this thesis. As such, to run RTEMS on the target hardware, it was necessary to add the Arty Z7 board to the configure.ac file. This file defines the memory regions of the BSP. RTEMS provides a BSP for the Zedboard, a board that is also based on the Zynq-7000 SoC [53], and therefore, the memory regions defined for the target Arty Z7 board in configure.ac were the same as the ones RTEMS defines for the Zedboard. It was also necessary to provide a build configuration file with information such as the compiler flags and libraries included, so the xilinx_zynq_artyz7.cfg was added, containing the default flags for the Cortex-A9 processor.

The following step is to configure the system. RTEMS is configured for each application through a set of macros to automate the generation of the data structures used at the system initialization. These macros encompass information such as the length of each clock tick, the application initialization tasks, the task scheduling algorithm and the device drivers needed by the application. An example of the possible configuration macros is presented in Figure 3.4. The configure program generates a variety of files based on these macros, including the necessary Makefiles to build and install RTEMS.

```
1  #define CONFIGURE_INIT_TASK_ATTRIBUTES RTEMS_FLOATING_POINT
2
3  #define CONFIGURE_MICROSECONDS_PER_TICK 1000
4
5  #define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
6
7  #define CONFIGURE_MAXIMUM_TASKS 20
8
9  #define CONFIGURE_MAXIMUM_TIMERS 4
10
11 #define CONFIGURE_RTEMS_INIT_TASKS_TABLE
12
13 #define CONFIGURE_MAXIMUM_PROCESSORS 1
14
15 #define CONFIGURE_INIT
16
17
18 #include <rtems/confdefs.h>
```

Figure 3.4: RTEMS configuration example.

3.2 ARINC 653

In an effort to standardize the accesses to shared resources in IMA systems, in October 1996 the AEEC published the first version of *ARINC 653 - Avionics Application Software Standard Interface* [2]. While originally a single document, it has since been divided into 5 parts, which describe the required and recommended services of an IMA OS, the subset services, the core software recommended capabilities and the conformity tests specification. *Part 1 - Required Services* of ARINC 653 defines the application executive (APEX), an added software layer that separates the application from the operating system. Figure 3.5 shows the ARINC 653 architecture overview.

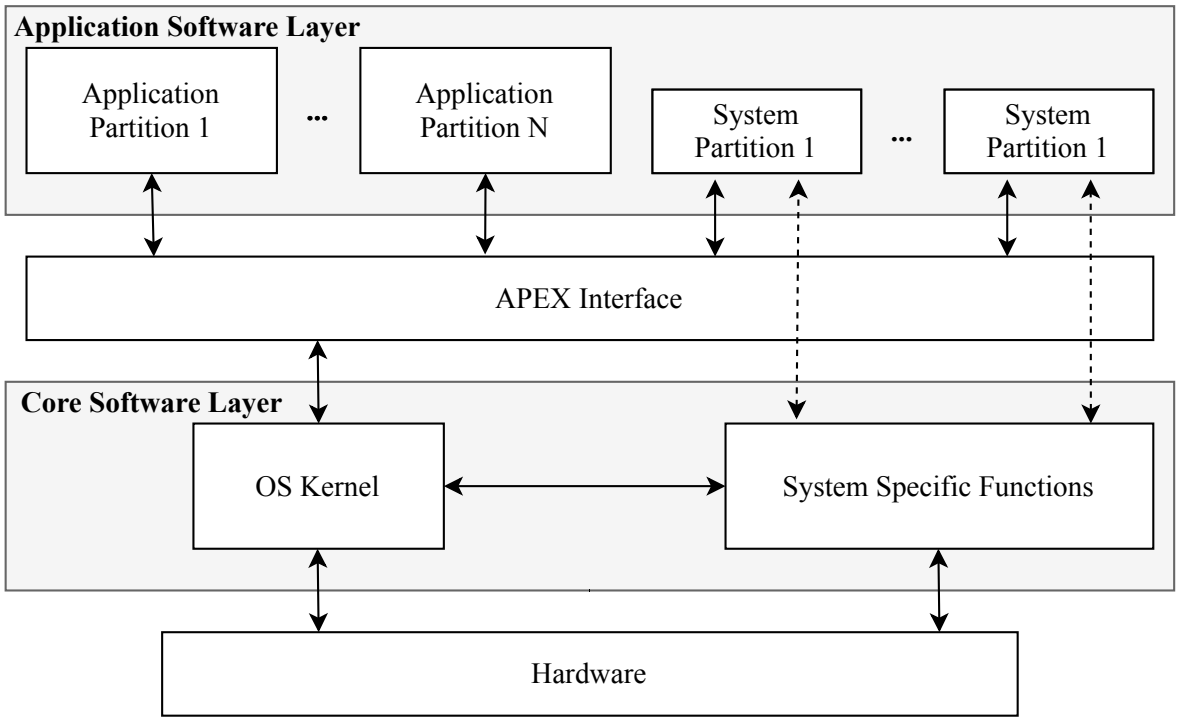

Figure 3.5: ARINC 653 architecture.

This specification offers several benefits, such as:

- Modularity, achieved by removing hardware and software dependencies;
- Portability of software applications;
- Reusability of components;
- Integration of software with multiple criticalities.

3.2.1 Partitioning

One of the main purposes of the hypervisor in an IMA system is to support multiple avionic applications, independent from each other. The separation of the applications is crucial in order to provide fault containment and ease of verification, validation and certification. To ensure this separation, both in time and in space, the ARINC 653 specification describes the concept of partitioning. Each partition is a program in a single application environment, which encompasses their individual data, context, configuration attributes and other partition specific information. Robust partitioning ensures that partitions with different criticality levels can be executed without affecting one another. With robust partitioning, partitions should not be able to interfere with other partition's code, I/O and data storage areas, nor to consume shared processor and I/O resources out of their period of execution. Any failure occurring inside a partition should not be propagated to the rest of the system.

Application partitions comprise the software specific to avionic applications, and are restricted to only using the system calls defined in the APEX. Optionally, there can be System Partitions that are also

subject to the robust TSP but are allowed to use services other than those defined in the APEX. System partitions can be used, for example, for managing faults or communication with hardware devices.

The partitioning is enforced by the hypervisor, which not only ensures that the partitions are completely contained from each other, but also handles any error that is raised in one application without interfering with the other partitions.

Temporal Partitioning

To ensure that each partition is temporally contained, the hypervisor implements a fixed schedule, defined beforehand through configuration tables. The schedule is implemented in a major time frame of fixed duration which is periodically repeated until the end of the runtime operation. The major time frame is divided into minor time frames that can be allocated for a partition.

During the execution, at the end of each minor time frame, the hypervisor checks if the next minor time frame is assigned to the same partition as the previous. If so, the partition is resumed. Otherwise the partition context is saved and the succeding partition's context is restored, returning the execution to the application. Through this method a deterministic behavior is achieved, and the partitions are provided uninterrupted access to common resources during their assigned time periods. An example of a possible schedule with three partitions in a dual core configuration is presented in Figure 3.6.

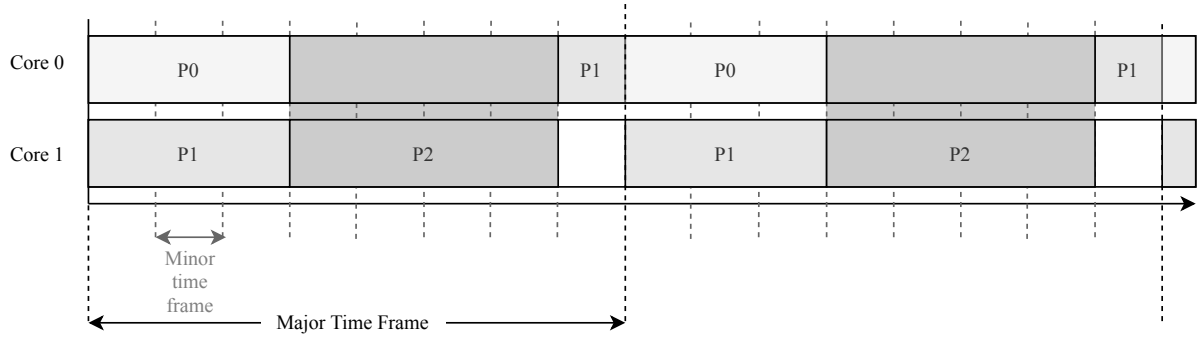

Figure 3.6: Example of a possible schedule with three partitions and two cores. Two major time frames are represented, each consisting of 8 minor frames.

Memory Partitioning

Spacial containment is achieved by allocating predetermined areas of memory for each partition. The partition is prohibited to access memory outside its designated areas.

In targeted hardware that contains an MMU, such as the Cortex-A9 processor, the MMU can be used to translate the partition's virtual memory to physical memory, ensuring that the memory accesses of the partition are within the permitted boundaries. A possible alternative would be to virtualize every instruction that accesses the memory, but the performance would significantly deteriorate.

The communication between partitions is accomplished through messages, which are defined as continuous blocks of data of limited length. These messages are sent and received through channels to

which the partitions access via source and destination ports. The hypervisor conducts the ARINC 653 services that establish this inter-partition communication.

3.2.2 Fault Detection and Response

Although partitioning can provide fault containment, by itself it is not enough to ensure that the occurring failures are properly handled without interfering with the remaining system. It is important that there is a mechanism to monitor and report faults in the hardware, application and OS, providing isolation and adequate recovery.

Following the ARINC 653 specification, this is achieved through the implementation of the Health Monitor (HM). The Health Monitor uses configuration tables to decide the adequate response to a certain fault according to the error level, which is established in light of the operational state in which the fault occurs as well as the error ID. The three error levels defined by the ARINC 653 are presented in Table 3.1. These levels do not include the errors that might occur in the hypervisor, leaving them and their reporting mechanisms to the responsibility of the system integrator.

Error Level	Impact	Examples
Process	One or more processes in the partition, or entire partition; The HM will not violate partitioning.	Errors raised by an application process; Illegal OS requests; Process execution errors such as overflows or memory violations.
Partition	One partition; The HM will not violate partitioning.	Errors during partition initialization, process management or error handling;
Module	All the partitions within the module.	Errors during module initialization, system specific functions or partition switching; Power fail.

Table 3.1: ARINC 653 Error Levels.

In case of a partition or module fault, the Health Monitor uses the error level and the error ID to determine the appropriate HM Callback to respond to that particular fault. The HM Callback is independent of the implementation, and should not violate the partitioning: If the fault is raised at the partition level, the HM Callback should be performed within the partition's time slice. If raised at the module level, all the partitions within the module are affected, but the HM Callback should not interfere with the other modules. In both of these cases, there are three possible recovery actions:

- Ignore (the error should be resolved by the HM Callback);
- Stop the partition/module;
- Stop and restart the partition/module.

As for errors at process levels, the adequate response should be determined by the application using the error handler process, a higher priority task of the partition. Several process level errors are assigned

error codes that can be returned to the faulty application to be used in the recovery. These error codes are implementation independent to allow portability. Aside from the possible recovery actions of the partition level errors, in the process level the HM might also choose to stop the faulty process and either re-initialize it, start another process or assume that the partition is capable of recovering unassisted.

The HM recovery actions might instigate mode changes to the partition. There are four possible modes of operation:

- *IDLE* – The partition is not initialized and no processes are executing
- *NORMAL* – The process scheduler is active, the processes have been created and those in the ready state are able to run.
- *WARM_START* and *COLD_START* – The partition is initializing. The distinction between the two modes depends on the implementation: The information of which mode is running can be useful for the application.

According to the action that is performed to recover from partition or process errors, the adequate partition mode will be set. When operational, the partition mode is NORMAL. When the partition is stopped, the mode will be set to *IDLE*. When the partition is restarted, its mode will be either *WARM_START* or *COLD_START*. Apart from the changes through the Health Monitor, the mode of the partition can also be changed through the *SET_PARTITION_MODE* request.

3.3 AIR

AIR stands for ARINC 653 Interface for RTEMS, and, as the name implies, it was developed as an adaptation of RTEMS to implement the ARINC 653 standard. It is a real time TSP hypervisor, allowing multiple VMs to run on a single processor while mantaining temporal and spacial isolation between applications.

GMV started developing AIR under the IMA-SP [25] and SAVOIR-IMA [29] initiatives, and it has been upgraded to be compliant to the core requirements set by the IMA Separation Kernel Qualification Project [54]. The AIR project is managed in a Gitlab repository [55] and is currently kept under the open source GPLv2 license.

3.3.1 AIR Architecture

AIR consists of four development independent modules, presented in Figure 3.5:

- The **Partition Management Kernel (PMK)** holds the main functionalities of the hypervisor. It implements temporal and spacial segregation, the scheduling initialization, the interrupt handling and the context switch between partitions. It can be further split into three submodules:
  - *Core* holds the core functionalities of AIR such as partition managment, TSP, multicore handling, implementation of system calls and the HM;

  - *Arch* implements the functionalities required for AIR to run on the target processor architecture, such as exception handling and memory management;
  - *BSP* provides functionalities required for AIR to run a BSP. This includes the board initialization and shutdown, and the access to peripherals.

- Each partition has its own **Partition Operating System (POS)**, a paravirtualized RTOS capable of features such as scheduling tasks and managing virtual interrupts. Currently AIR for SPARC supports RTEMS 5, RTEMS 4.8i and a bare executive without an RTOS. The goal of this dissertation is to have AIR for ARM also support RTEMS 5, in addition to the barebones OS that it previously supported.

- The **LIBS** module consists of a set of libraries that implement functions to assist the connection between the modules. It currently includes five libraries:
  - *IMASPEX* implements the APEX interface which defines services that conform to the ARINC 653 standard;
  - *LIBAIR* assists the POS virtualization by implementing the adequate system calls;
  - *LIBIOP* configures a system partition to manage peripheral I/O devices;
  - *LIBPRINTF* supplies the printf functionality;
  - *LIBTEST* provides functions to test and validate the software.

- The **AIR toolchain (TOOLS)** is a parallel module coded in python to aid the build process. It is split into two submodules:
  - The *Configurator* generates the necessary makefiles to build AIR. It is called through the configure script, that can be used at two levels: first at the root level, to configure AIR for the target processor and BSP; then, at the application level, to extract the application configuration options from the config.xml file. This file contains the partition options, the schedules, module information and Health Monitor Tables, and an example can be seen in Figure B.1.
  - The *Partition Assembler* aggregates the built partitions into a single executable to allow PMK to manage the partitions.

3.3.2 Health Monitor

As defined by the ARINC 653 standard, a fault in the execution should be handled by a Health Monitor (HM). In the ARM architecture, the HM handles abort and undefined exceptions. Figure 3.8 depicts the HM procedure. The HM handler starts by searching the error ID and the operational state in the system HM table to check whether the fault occurred at the module or partition level. It then searches for the error ID in the table corresponding to that error level to obtain the action that should be performed. These tables are defined by the user in the application configuration XML file (config.xml). An example of these tables is presented in Figure B.2

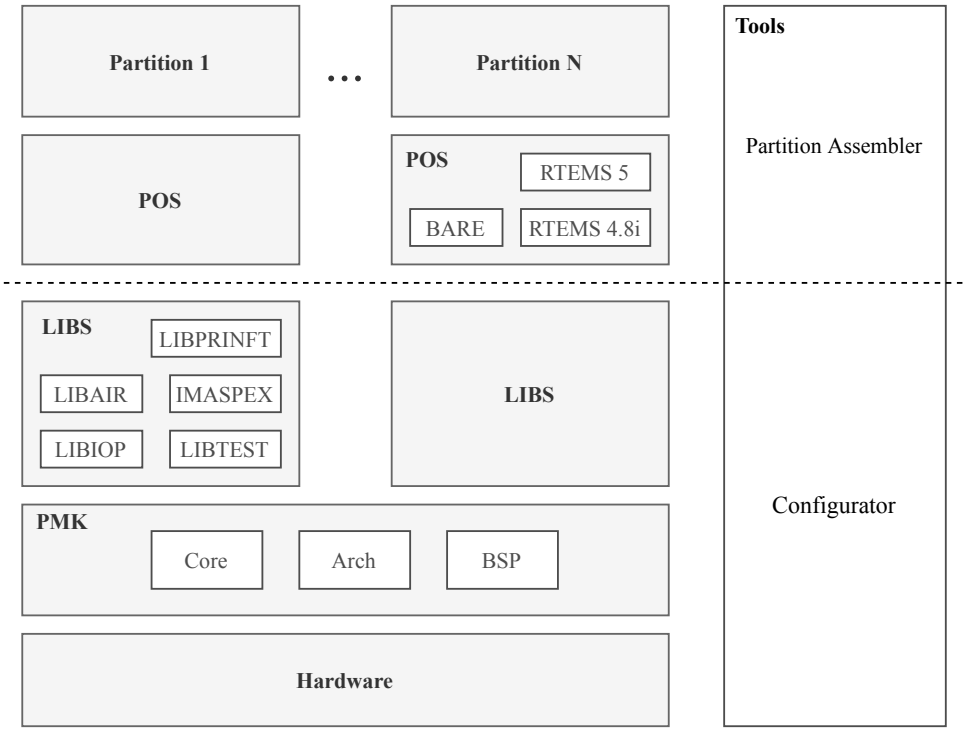

Figure 3.7: AIR architecture.

In the module level, the available actions are IGNORE, SHUTDOWN and RESTART. In the partition level, the actions are IGNORE, SHUTDOWN, COLD START AND WARM START. The COLD START and WARM START both perform a partition reset, but the information of which occurred can be useful for the application.

IGNORE is used to leave the handling of the exception to the application programmer's criteria. This action returns to the partition HM handler, which the application programmer can change to resolve the fault however they see fit.

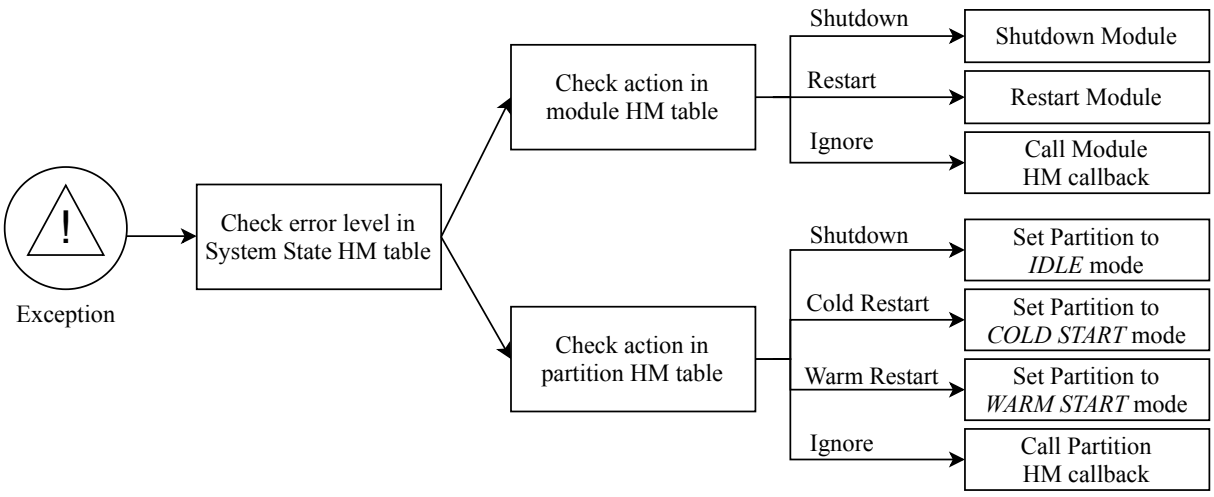

Figure 3.8: AIR HM flow chart.

3.3.3 Communication with I/O devices

To ensure TSP in the communication with hardware devices, AIR defines a system partition responsible for managing the transmission and reception of data to and from I/O device drivers. The I/O partition (IOP) operates in a loop which verifies the messages to be sent to or received from the driver, performing a write and/or read for each of the configured drivers. The configuration of the device drivers and communication routes between them and the IOP is done in the iop.xml file.

The application partitions communicate with the IOP through the inter-partition communication services defined by the ARINC 653. Figure 3.9 presents a possible system comprised of two application partitions and the IOP.

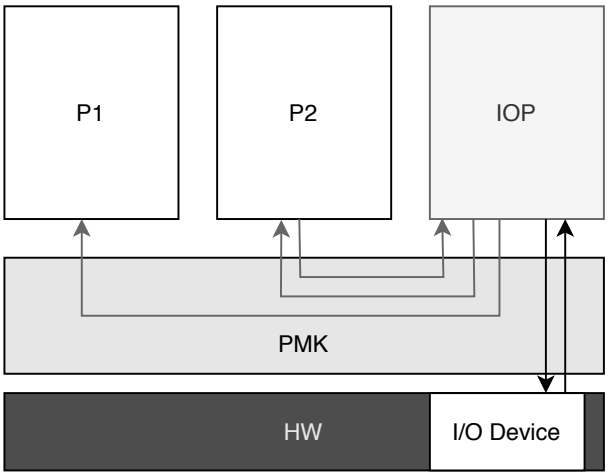

Figure 3.9: Communication with I/O devices.

AIR for SPARC currently supports Ethernet, Spacewire, CANBUS and MIL1153 device drivers. As for the ARM architecture, the drivers of UART and CAN are available while the time-sentitive network (TSN) driver is currently in development.

3.3.4 Migration to the ARM architecture

In the recent migration of AIR from SPARC to ARM [6], a comparative study between the two architectures was performed in order to maintain the same logic of the hypervisor for both architectures and to minimize the hardware dependencies. This migration comprised all of the required functionalities for AIR to run on the target Zynq-7000 SoC, such as initialization, shutdown and access to peripherals, as well as most of the hypervisor features.

The most influential dissimilarity between ARM and SPARC is the exception handling. While SPARC processes the exceptions through a single table of handlers (trap table), ARM resorts to multiple redirection tables. Despite the significant changes on the logic of handling exceptions that this difference entails, the implementation of the AIR exception handlers achieved the same results in both architectures.

The goal of a hypervisor is to add a virtualization layer to allow several VMs to run on the same machine while providing strict segregation of each application's resources. To enforce isolation, the

partitions are assigned a virtual memory region that is then translated into real memory by the MMU. Although the implementation of spacial segregation through the MMU is hardware dependent, SPARC and ARM MMUs share a similar structure and therefore the logic of the implementation was kept mostly the same.

The temporal segregation is achieved through the scheduler, which is shared between the two architectures. The scheduler is activated at each timer interrupt, and checks if the schedule demands a partition switch. The context switch between the partitions is specific to each architecture, and resorts to virtualization data structures to store the current VM state.

The first of the two main data structures that aid virtualization is the core context, used to restore and store data whenever there is a partition switch. Prior to this dissertation, the core context for ARM comprised:

- The virtual CPU structure, including:
  - the CPU ID,
  - the Program Status Register (PSR), clarified in Section 3.4.2 ,
  - the virtual trap table address,
  - the MMU control registers;
- A structure containing the virtual Generic Interrupt Control (GIC) registers (further explained in Section 3.4.5);
- The core entry point;
- A pointer to the current Interrupt Stack Frame;
- The interrupt nesting level;
- The information on the current AIR state;
- The Floating Point Unit (FPU) context, if enabled;
- The HM event currently being serviced, if there is one.

The other main virtualization data structure is the Interrupt Stack Frame (ISF), stored and restored at each exception, and it included:

- The ARM general registers, r0 to r12;
- The current exception name;
- The Stack Pointer (SP) and Link Register (LR) prior to the current exception;
- The return address;
- The return PSR.

Both of these structures required modifications throughout this dissertation, which will be approached in Chapter 4.

The communication between a VM and the hypervisor is done through system calls. While a few system calls were added or completed in this dissertation, most of them were already provided. Table 3.2 presents some of the available system calls to be used in the paravirtualization of an RTOS.

Syscall	Description
AIR_SYSCALL_GET_NB_CORES	Get the number of virtual cores in the partition.
AIR_SYSCALL_GET_CORE_ID	Get the current virtual core ID.
AIR_SYSCALL_GET_US_PER_TICK	Get the number of microseconds per tick.
AIR_SYSCALL_GET_ELAPSED_TICKS	Get the number elapsed ticks since the partition start.
AIR_SYSCALL_ENABLE_FPU AIR_SYSCALL_DISABLE_FPU	Enable or disable the FPU unit.
AIR_SYSCALL_ENABLE_TRAPS AIR_SYSCALL_DISABLE_TRAPS	Enable or disable the virtual traps.
AIR_SYSCALL_ENABLE_INTERRUPTS AIR_SYSCALL_DISABLE_INTERRUPTS	Enable or disable the virtual interrupts.
AIR_SYSCALL_GET_IRQ_MASK_REGISTER AIR_SYSCALL_SET_IRQ_MASK_REGISTER	Get or set the virtual interrupt mask.
AIR_SYSCALL_GET_TBR AIR_SYSCALL_SET_TBR	Get or set the virtual trap table.
AIR_SYSCALL_GET_PSR AIR_SYSCALL_SET_PSR	Get or set the virtual processor status register.
AIR_SYSCALL_RETT	Return from the virtual exception.

Table 3.2: AIR virtualization system calls.

The migration of AIR to the ARM architecture resulted in the successful deployment of the AIR hypervisor in an ARM based SoC, achieving the same behavior as the SPARC's BSP. The broadening of the AIR supported architectures allows the hypervisor to be a candidate for various space projects, having already won proposals for the INFANTE [56], PERIOD[1] and EROSSplus[2] projects and being planned to be integrated in MIURA 5 [57], a suborbital micro launcher being developed by PLD Space for research and technology development purposes, as a successor of MIURA 1.

Prior to this dissertation, AIR for ARM had only been tested with a basic barebones OS, and some important functionalities were not yet implemented or lacked testing, such as the virtual exception handling, the error processing inside the application and multitasking capabilities. Through the virtualization of RTEMS, this dissertation accomplished an improved version of AIR for ARM capable of running this

[1] PERASPERA In-Orbit Demonstration is a project to develop European space robotics technologies using ESROCOS, a RTOS that incorporates AIR to provide TSP. The project is led by Airbus and the target hardware is the Zynq UltraScale SoC.

[2] EROSSplus is pursuing the same goal as PERIOD. It is led by Thales-F and targeted at the DAHLIA SoC

RTOS, allowing more complex tests to be executed and consequently further the validation of the hypervisor.

3.4 ARM

ARM is a family of Reduced Instruction Set Computing (RISC) processor architectures whose first chip had been in development since the 80's by Acorn Computers. In 1990, this company partnered with Apple Computer and VLSI Technology and founded Advanced RISC Machines Ltd which in 1994 created a 16-bit instruction set that lowered memory demands. This design was licensed by Texas Instruments and deployed on the Nokia 6110, which was greatly successful [58].

ARM introduced an innovative and profitable business model in which, rather than building and selling a product themselves, ARM develops intellectual property (IP) blocks that are then licensed to semiconductor companies. This led to ARM being designed into more and more SoCs, becoming the standard for the growing market of cell phones [59]. By 2015 ARM had a share of 96% of the mobile market, and at the end of 2018 ARM held a share of 33% of the SoC market [60].

In 2005, ARM introduced the three Cortex families in order to diversify their offers to better cover the demands of the industry. The Cortex-A, or the Application Processor family, focuses on higher performance. The Cortex-M provides extremely low-power, low-cost microcontrollers. The Cortex-R provides high performance, real time processors. In 2008, in order to provide increased performance while maintaining a long battery life, ARM introduced the Cortex-A9 MPCore, which implements the ARMv7-A architecture. This is the processor equipped in the Arty Z7 provided by GMV, and will be the target processor of the work ahead.

3.4.1 Instruction Sets

The ARMv7 architecture, implemented by the Cortex-A9 MPCore, has two main instruction sets:

- The **ARM** instruction set is a set of 32-bits instructions, aligned on a four-byte boundary, supporting performance critical functions such as interrupt handling.
- The **Thumb** instruction set was initially developed as a 16-bit instruction set to offer higher code density in detriment of performance. An extension of this instruction set was then released as Thumb-2, which includes the original 16-bit instructions as well as added 32-bit instructions, allowing this instruction set to perform similarly to the ARM instruction set with the benefit of improved code density.

Each of these instruction sets has a corresponding processor state which can be interchanged freely from Thumb to ARM and vice versa. The information on the processor state is stored in the program status register (PSR) [61].

ARMv7 also provides two additional instruction sets to support execution environments:

- The **ThumbEE** is a variant of Thumb designed for dynamically generated code, providing support for Just-in-Time (JIT), Dynamic Adaptive Compilation (DAC), and Ahead-Of-Time (AOT) compilers. However, it can't interwork freely with ARM and Thumb.

- The **Jazelle** instruction set allows the execution of Java Bytecode directly on hardware.

In AIR, most code is written in C, leaving only the start-up code and the exception handlers in Assembly. In space applications, memory constraints make code density an advantageous trait. For this reason, in AIR, only the exception handlers are compiled in ARM and the remaining code in Thumb-2.

As for RTEMS, in addition to the start-up code and the exception handlers, the Context Switch is also coded in Assembly. The exception handlers contain both ARM and Thumb-2 segments, while the rest of the assembly code is ARM.

3.4.2 Program Status Register

The Program Status Register is a 32-bit processor register that holds information on the processor status and control. Figure 3.10 shows the allocation of the bits of PSR. Bits 0-4 store the processor mode, further explained in the Section 3.4.4. The bits 5 and 24 are the Instruction Set Flags, T for thumb and J for Jazelle. Bits 6-8 are the interrupt flags, disabling IRQ (I), FIQ (F) and imprecise aborts (A) if set. Bits 28-31 are the condition flags: Negative (N), zero (Z), carry (C) and overflow (V).

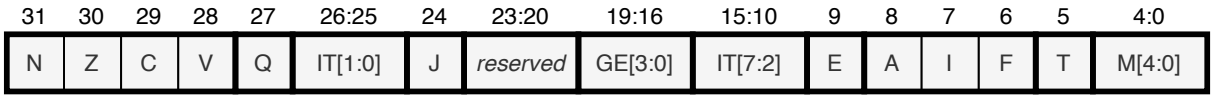

Figure 3.10: Bit allocation of Program Status Register.

The current state of the processor is stored in the Current Processor State Register (CPSR), and each mode (except for User and System modes) has its own Saved Processor State Register (SPSR) which holds the previous CPSR from before changing to the current mode, so that the previous context can be restored.

3.4.3 Coprocessors

ARM processors can be complemented by coprocessors, which expand the processing features of a core by either extending the instruction set or by providing configuration registers. The coprocessors are accessed through dedicated ARM instructions, which support the connection up to 16 coprocessors.

The Vector Floating-Point (VFP) coprocessor is mapped as CP10 and CP11, and it provides single and double precision floating point arithmetic operations through an added instruction set.

CP14 and CP15 are reserved for configuration and control related to the architecture. CP14 is the debug coprocessor, implementing debug features, and CP15 is the system control coprocessor, providing control over the standard memory and system facilities. Most of the CP14 and CP15 registers require privileged access.

3.4.4 Operating Modes

The ARMv7 basic model operates in two privilege levels: PL0, the lowest privilege level, in which an exception is raised if there is an attempt to access privileged resources, and PL1, in which an operating system is expected to run. If the Virtualization Extensions are implemented, another privilege level (PL2) is added for the hypervisor to execute at.

There are seven basic operating modes, each with its own stack pointer (SP) and link register (LR):

- **USER**: The mode most applications are expected to run at, being the only one executing at PL0. This means that it can only access unprivileged system resources and its assigned memory, and that in order to switch to a mode that runs at a higher privilege level, an exception must be raised.
- **SYSTEM**: While this mode shares the USER registers, it runs at PL1, making it possible to access the USER's SP and LR without sacrificing the access to privileged resources. This mode can't be entered through an exception.
- **ABORT**: This mode runs at PL1 and is entered through the Data Abort and Prefetch Abort exceptions. Prefetch Aborts occur when the CPU attempts to fetch an instruction from an invalid memory region. Data Aborts are raised when there is an attempt to write or to read data from an illegal memory location.
- **UNDEFINED**: Running at PL1, it is entered through the Undefined Exception, which occurs when the processor does not recognize the instruction it is attempting to execute, or when the processor recognizes it as a coprocessor instruction but no coprocessor recognizes it - for instance, if there is an VFP instruction but the floating point unit (FPU) is disabled.
- **SUPERVISOR**: Without the Virtualization Extensions, this is the standard entry mode of the processor after a reset. It can also be entered through a Supervisor Call (SVC). This should be the default mode in which the OS runs.
- **IRQ**: The Interrupt Request mode, running at PL1, is entered through IRQ exceptions.
- **FIQ**: The Fast Interrupt Request mode, running at PL1, is entered through FIQ exceptions. These are higher privilege interrupts that are able to interrupt IRQ.

If CPU extensions are implemented, two additional operating modes might be available:

- **HYPERVISOR**: Introduced by the Virtualization Extensions, this mode runs at the higher privilege level PL2. It is entered through Hypervisor Calls or Hypervisor Traps.
- **MONITOR**: Introduced by the Security Extensions, this mode runs at PL1 and manages the switches between the Secure and Non-secure states. If this extension is available, all the basic operating modes have copies in the Secure and Non-secure states, and the son-secure applications are not permitted to interfere with the secure. The Monitor mode is always in the Secure and the Hypervisor mode is always in Non-Secure. The Monitor mode is entered through a Secure Monitor Call.

3.4.5 Exception Handling

Exceptions, also referred to as traps, are anomalous or exceptional events that interrupt the normal execution and require special processing. When an exception is raised, the execution jumps to a predefined table of handlers.

While in other architectures exceptions are handled through a single trap table (such as SPARC, with a 256 entries trap table), ARM handles exceptions through multiple redirection levels: a first main trap table contains the 7 main exceptions that can be raised, presented in Table 3.3, and redirects the execution to a **low level handler**, coded in assembly. Within the 7 main exceptions, there are several possible interrupt sources, such as timers, device drivers and different faults. The low level handler calls a C subroutine, or a **high level handler**, to identify the interrupt source and perform the adequate actions.

Exception	Corresponding Mode
Reset	Supervisor
Undefined Instruction	Undefined
Supervisor Call (SVC)	Supervisor
Prefetch Abort	Abort
Data Abort	Abort
Interrupt Request (IRQ)	IRQ
Fast Interrupt Request (FIQ)	FIQ

Table 3.3: ARM exceptions.

When an exception occurs, the CPSR is stored in the SPSR of the mode that corresponds to the exception raised, and the return address is stored in the LR. The execution jumps to the trap table, assuming the operating mode corresponding to the exception, and performs the low level handler assigned, which should store the processor context, link to the high level handler and restore the context before returning to the normal execution.

The information needed to manage the interrupts is mapped in the General Interrupt Controller (GIC) registers [62, 63].

The GIC architecture is logically divided into:

- A Distributor block that centralizes all interrupt sources, determines the priority of each interrupt and forwards the interrupt with the highest priority to each CPU interface;
- CPU interface blocks, which perform priority masking and preemption handling for a processor connected to the GIC.

The most relevant GIC registers for the work ahead are presented in Table 3.4

The high level handler, or the Interrupt Service Routine (ISR), should verify if the interrupts are enabled through the CPU Interface Control Register (ICCICR) and then check the interrupt priority through the Interrupt Priority Register (ICDIPR). If the current priority is higher than the Interrupt Priority Mask (ICCPMR), then the interrupt is handled. If not, the interrupt is set as pending until the ICCPMR priority is lower than the current interrupt priority.

Register	Description
ICDISER	Interrupt Set-Enable
ICDICER	Interrupt Clear-Enable
ICDPR	Interrupt Set-Pending
ICDIPR	Interrupt Priority
ICDSGIR	Software Generated Interrupt
ICCICR	CPU Interface Control
ICCPMR	Interrupt Priority Mask
ICCIAR	Interrupt Acknowledge
ICCEOIR	End of Interrupt
ICCHPPIR	Highest Priority Pending Interrupt

Table 3.4: ARM GIC Registers.

When an interrupt is handled, the CPU should access the interrupt ID stored in the Interrupt Acknowledge Register (ICCIAR) and perform the handler installed for that interrupt ID. For instance, the Global Timer interrupt ID is 27, and this prompts the ISR to perform the clock handler.

In a virtualized environment, the virtualized OS running in a non privileged level should not access the real GIC blocks, so the necessary registers must be copied into the VM. The data structure that holds the virtual GIC registers in AIR for ARM is presented in Figure 3.11

```
1  typedef struct {
2      air_u32_t vm_ctrl;              /**< VM interrupt control       */
3      air_u32_t ilist[16];            /**< 32 virtual interrupts      */
4      air_u32_t apr;                  /**< active priority            */
5      air_u32_t pmr;                  /**< priority mask              */
6      air_u32_t bpr;                  /**< binary point               */
7      air_u32_t iar;                  /**< interrupt acknowledge      */
8      air_u32_t eoir;                 /**< end of interrupt           */
9      air_u32_t rpr;                  /**< running priority           */
10     air_u32_t hppir;                /**< highest priority pending   */
11 } arm_virtual_gic_t;
```

Figure 3.11: Virtual GIC registers data structure.

When returning from the exception, the SPSR is put in the CPSR, returning to the previous mode, and the program jumps back to address stored in the LR.

Chapter 4

Implementation

This chapter presents the work performed in order to accomplish an improved version of AIR for ARM capable of running RTEMS 5 as the Partition Operating System (POS). It starts by, in section 4.1, describing the work procedure used throughout this dissertation, and then goes through the logical sequence of running RTEMS on AIR while detailing the modifications applied to RTEMS and to the AIR hypervisor.

4.1 Work Plan

Prior to this dissertation, the migration of AIR to ARM had already accomplished some Partition Management Kernel (PMK) functionalities on AIR, such as the memory translation between virtual and real memory, the hypervisor scheduler, partition switching, AIR exception handling and the implementation of several system calls. It was tested with a simple barebones OS, created specifically to test the basic hypervisor functionalities. However, due to the simplicity of the barebones OS, some important functionalities were not implemented or lacked testing. For instance, AIR did not have a mechanism to run the guest OS's virtualized exception handlers, and consequently the guest OS's scheduler and multitasking were not supported. This means that, in order to run RTEMS on AIR for ARM, it was not enough to apply the required modifications to paravirtualize RTEMS. It was also necessary to perform modifications to the PMK to ensure that it correctly interoperates with the POS in order to maintain the expected behavior of RTEMS.

RTEMS is a reliable RTOS that has been subjected to extensive testing with a nearly total code coverage [64]. To keep this reliability, it is important to minimize the amount of changes applied to the RTEMS code.

The easiest way to perform the necessary modifications to both the POS and the PMK is through a sequential and iterative workflow, by executing the RTEMS code on top of AIR in a logical order, detecting the flaws, correcting them and repeating the process until the program is capable of reaching the end of the test as expected. The expected behavior of RTEMS is known from the execution of the tests in the original RTEMS, without the hypervisor. To facilitate this comparison, the first tests used for

this iterative process were from the RTEMS test suites [65], starting with a simple example such as the hello test and evolving to more complex examples and intricate features as the basic functionalities are ensured. This workflow is depicted in Figure 4.1.

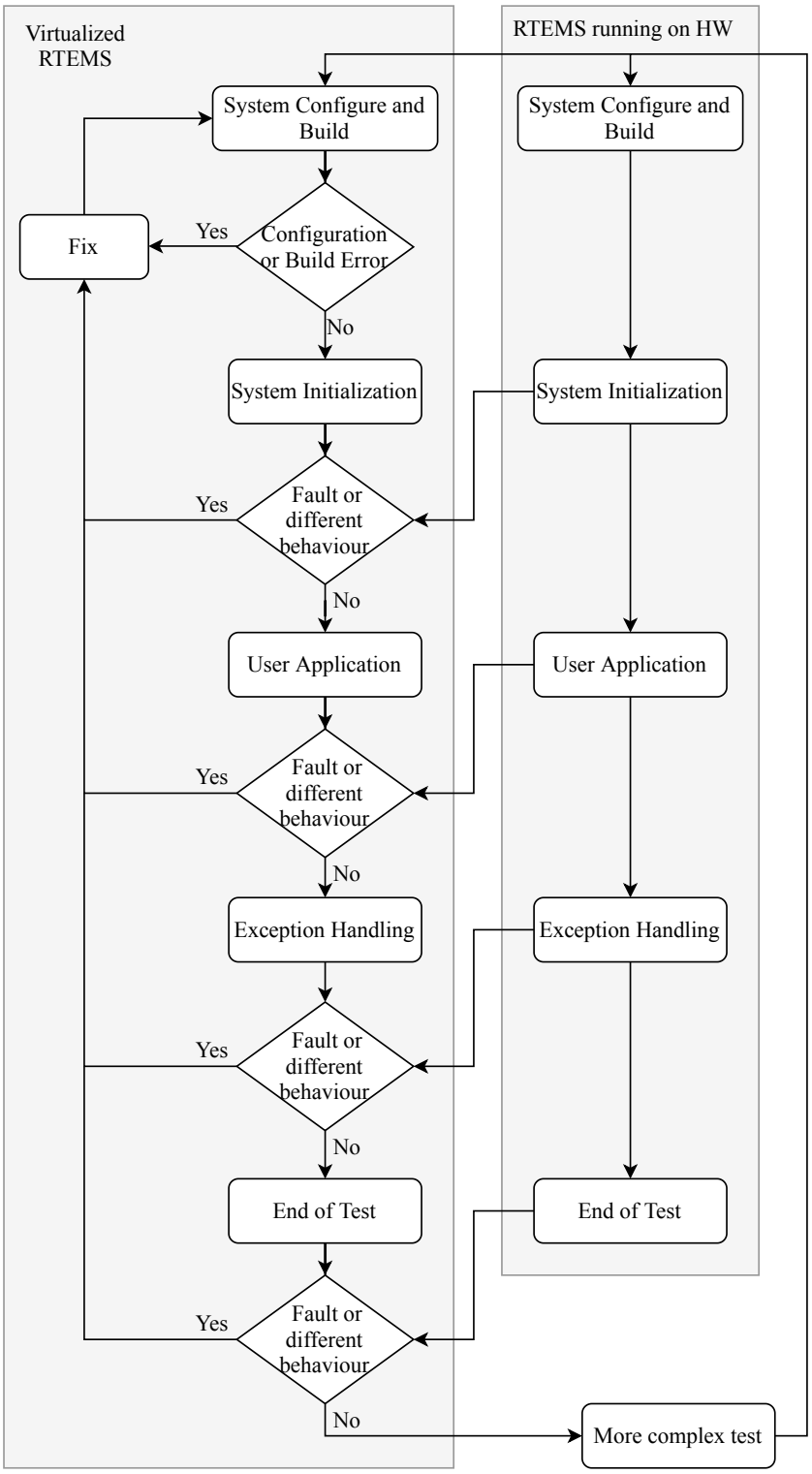

Figure 4.1: Adopted iterative workflow.

RTEMS was designed so that most of the code is hardware independent. The required modifications can therefore be split into five main areas of focus:

1. System configuration and build;

2. System initialization;

3. User Application;

4. Exception Handling;

5. System shutdown (End of Test).

This logical sequence will be followed through the method depicted in Figure 4.1.

In RTEMS, the illegal accesses must be detected and prevented. These accesses generally raise faults, and must evaluated as to whether it is necessary to replace them by system calls or if they can simply be removed. System calls should only be used if no other solution allows the desired results, as they interrupt the OS code through an exception, forfeiting efficiency. There are many instances in the code in which RTEMS attempts to perform an action that, in a virtualized environment, should be left to the hypervisor. These segments can be removed, reducing and simplifying the POS code.

In summary, the changes applied to RTEMS comprised the following:

- Remove code that performs actions that should be left to the hypervisor;

- Prevent accesses to privileged resources, either by removing them or by replacing them with system calls;

- Fix issues raised from running in user mode a function designed for more than one operating mode.

The adopted iterative workflow also allows to spot flaws in the hypervisor code. These flaws include lacking the implementation of a required system call, corrupting data or restoring the wrong contexts, which affect the behavior of the POS and can therefore be detected through the comparison of the obtained and expected results. A flaw can also be due to an incomplete migration of the code that was intended for SPARC. The modifications to AIR included:

- Implementing the missing system calls;

- Completing the exception handling to support virtualized exception handlers and multitasking;

- Completing the virtual data structures with needed information;

- Adapting code intended for SPARC in order to run on ARM as well, minimizing hardware dependencies when possible.

The next sections present an overview of the changes performed in RTEMS and AIR, following the logical sequence presented in Figure 4.1.

4.2 System Configuration and Build

The first step to run RTEMS on AIR for ARM is to add it as a supported POS in the AIR toolchain, so that the AIR configure script can generate the necessary files to compile the system and run the application while incorporating the RTEMS configuration and build process described in Section 3.1.4. The python toolchain resorts to Mako templates to generate these files, which include makefiles, the init.c that initializes the user application and, in the case of RTEMS partitions, the rtems_config.h, which contains the RTEMS configuration macros mentioned in Section 3.1.4.

Since RTEMS is already a supported POS for SPARC, at this point the implementation consists only on migrating the RTEMS configuration and makefile template from SPARC and applying it to ARM by changing the name of the target BSP.

Before building the system it is also necessary to provide each partition a script that configures the memory regions in which the code and data of the application will be written to. This data is configured in the linkcmds file, which is used by the partition assembler to generate an executable encompassing the partitions that will be decompressed by the PMK at the partition initialization. The memory assigned to a partition in the linkcmds was contained between the addresses 0x10000000 and 0x10200000, providing each partition 2MB. The memory regions attributed to the partitions correspond to virtual memory, which will be translated into physical memory by the MMU.

4.3 System Initialization

The initialization of the AIR hypervisor is responsible for starting the BSP and initializing the hypervisor features such as the scheduler, spacial segregation, HM and, if available, multicore. Once the PMK is ready to run the applications, the execution enters the entry point of the partition assigned to the first minor time frame by the scheduler. If the OS assigned to that partition is RTEMS, then this entry point corresponds to the start of the RTEMS initialization.

The RTEMS initialization is performed at two levels. First, at a lower level, by the start code present in the start file. This file is typically written in assembly and is specific for each architecture, performing the minimum actions possible that enable the processor to run C code correctly. As in a virtualized environment the board is already initialized when entering a partition, most of the low level initialization required to run RTEMS directly on the HW is redundant and unnecessary when running on top of AIR, and can be removed.

The start code is concluded by calling the boot_card() function to perform the high level initialization. This function is common to every BSP, and calls the rtems_initialize_executive directive provided by the Initialization Manager, which is responsible for passing the control of the processor from the OS to the user application.

4.3.1 Low Level Initialization

RTEMS starts by initializing the registers if the system configuration deems it necessary. If not, the bootloader parameters are saved in registers that will not be changed throughout the function. When running on top of AIR, none of these cases is applicable: the hypervisor should manage both. Therefore, these instructions can be removed.

RTEMS then accesses the coprocessor CP15 in order to get the current processor ID. On AIR, to get the core ID without accessing the CP15, which would require a privileged instruction and is therefore illegal in user mode, the AIR_SYSCALL_GET_CORE_ID could be used instead. However, the core ID is only used to perform actions that are not applicable when running on the hypervisor: storing information in real processor registers, to which the partition should not have access, and setting the SP for each operating mode. Inside the partition, the only SP available is the user mode SP, already set by the hypervisor. Thus, getting the core ID is unnecessary at this stage, and there is no need to run or replace the instruction.

The initialization of the BSP, including setting up the cache and the MMU, starting the global timer, and copying the standard sections from load to run time, should be performed exclusively by the hypervisor, and were removed from the POS. The function to clean the .bss was kept, to ensure that the statically allocated variables are initialized with the value 0. This function only affects memory within the partition and therefore requires no virtualization.

The RTEMS start code for ARM also performs the initialization of the exception vectors, that is, assigning an address to the trap table. While in a virtualized environment the exceptions should be handled by the hypervisor, it is crucial to provide a virtual trap table so that the RTEMS handlers are performed after the AIR handlers. The setting of the virtual trap table will be delayed to the high level initialization in order to keep the virtualization of RTEMS for ARM consistent with the existing virtualization of RTEMS for SPARC. This implementation as well as the necessity of performing the virtualized RTEMS handlers will be further explained in Section 4.3.2.

As expected, most of the low level initialization could be removed, greatly simplifying the code. After the low level initialization, the system is capable of running C code safely, so the boot_card() function is called to complete the initialization at a high level. In the original RTEMS, all this function does is prepare to pass the control to the Initialization Manager and call the initialization directive rtems_initialize_executive(). However, as previously mentioned, the virtualized RTEMS for AIR still needs to set the virtual trap table.

4.3.2 Virtual Trap Table

When running RTEMS on top of AIR, the exceptions should be handled by the hypervisor. When an exception is raised, it will be caught by the AIR trap table and handled by the AIR handler. If AIR is not informed of a virtual trap table address, then after handling the exception, it will resume the partition from the same point where the exception was raised, as represented in Figure 4.2. The RTEMS handlers will not be executed, which means that the POS will not be aware that an exception occurred. This affects

several functionalities. In the case of timer interrupts, not performing the RTEMS clock handler means that its scheduler will not be activated at regular intervals, and the task management will be severely limited; regarding faults that are handled by the HM, the RTEMS exception handlers also need to be executed to allow the application programmer to assign a HM Callback to resolve the fault.

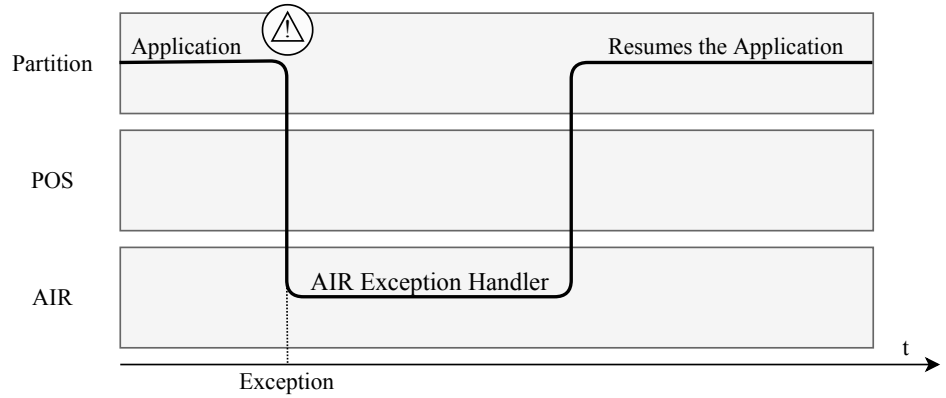

Figure 4.2: AIR exception handling procedure without setting a virtual trap table.

It is therefore imperative to set a virtual trap table using the AIR_SYSCALL_SET_TBR, to inform AIR that after performing the AIR exception handler and before resuming the user application, the execution should be redirected to the RTEMS handler assigned in the virtual trap table. Figure 4.3 depicts this exception handling procedure, which will be detailed in Section 4.5. The current section is focused in the initialization of the virtual trap table to allow future exceptions to be handled through this method.

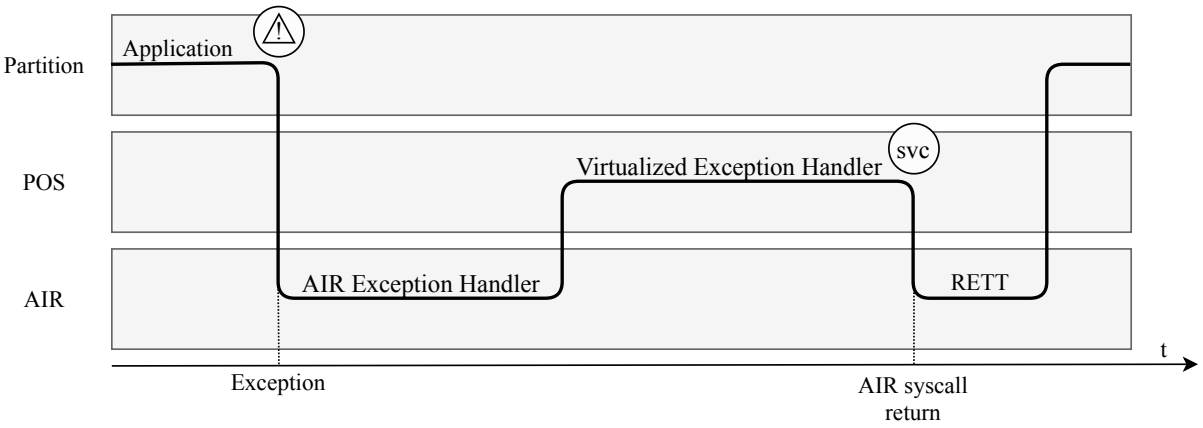

Figure 4.3: AIR exception handling procedure with a set virtual trap table.

This initialization will be applied to the boot_card() function to keep consistency between the virtualization of RTEMS for SPARC and ARM. This function is shared between all BSPs, so it it is crucial to ensure that the virtualization is valid for both architectures supported by AIR.

In SPARC, the CPU_INTERRUPT_NUMBER_OF_VECTORS macro holds the total number of entries to the trap table, and is used to declare an address to a vector that will be set as the virtual trap table base address, in which all the entries will be initialized with empty handlers to later replace them with the correct virtualized handlers. However, this macro is only defined for architectures with simple vectored

interrupts, which is not the case for ARM due to resorting to multiple redirection levels, as explained in Section 3.4.5.

A possible solution would be to define the CPU_INTERRUPT_NUMBER_OF_VECTORS macro as 8, the number of entries in the main ARM trap table, corresponding to the 7 main exceptions and an additional empty field to ensure memory alignment. Doing this would allow the virtualized boot_card() function to be the same for ARM and SPARC, initializing the trap table with empty handlers and afterwards installing each of the exception handlers one by one, but this solution is inefficient and would demand more modifications to the RTEMS code later on. Instead, the fact that the macro is not defined for architectures with multiple redirection tables can be taken advantage of, allowing a simpler solution to be implemented for the ARM architecture, shown in Figure 4.4.

```
1  #ifdef AIR_HYPERVISOR
2  /**
3   * @brief trap table of the system redefined by AIR
4   */
5  #ifdef CPU_INTERRUPT_NUMBER_OF_VECTORS
6      void *trap_table[CPU_INTERRUPT_NUMBER_OF_VECTORS];
7  #else
8      #include <bsp/start.h>
9      void *trap_table = bsp_start_vector_table_begin;
10 #endif
11 #endif /* AIR_HYPERVISOR */
12
13 void boot_card( const char *cmdline )
14 {
15   rtems_interrupt_level bsp_isr_level;
16
17 #ifdef AIR_HYPERVISOR
18   /*
19    * AIR redefines take control of the trap table here
20    */
21
22 #ifdef CPU_INTERRUPT_NUMBER_OF_VECTORS
23   /* clear trap table */
24   uint32_t i;
25
26   for (i = 0; i < CPU_INTERRUPT_NUMBER_OF_VECTORS; ++i) {
27       trap_table[i] = NULL;
28   }
29 #endif
30   /* set TBR */
31   air_syscall_set_tbr((air_u32_t )&trap_table[0]);
32
33   /* enable traps*/
34   air_syscall_enable_traps();
35
36 #endif /* AIR_HYPERVISOR */
37 }
```

Figure 4.4: Setting the virtual trap table in bootcard.c.

The original RTEMS for ARM defines the main trap table at the start.S file, starting at the address bsp_start_vector_table_begin (see Figure A.1). In the boot_card() function, for architectures that handle exceptions through multiple redirection levels, it is merely necessary to define the trap_table as a pointer

to this address, as done in line 9 of Figure 4.4. This way the trap_table already points to a table with the default exception handlers, and it is not necessary to install empty handlers (as done for architectures with simple vectored interrupts in lines 22 to 29) only to replace them with the desired handlers afterwards.

Having defined an address in which the handlers are installed, the AIR_SYSCALL_SET_TBR is used in line 31 to set that memory address as the virtual trap table, so that the hypervisor knows the address that it must return to after the AIR exception handler.

4.3.3 Initialization Manager

The rtems_initialize_executive() directive is responsible for preparing the system to pass control to the user application, and will initialize the RTEMS features that are configured for that particular program through a system initialization linker set. Most of these features do not require virtualization. The two services that do require modifications are the routine to start the BSP, which initializes the interrupts, and the routine to initialize all the device drivers.

The interrupts are initialized in bsp_interrupt_facility_initialize(), a function that is specific for each BSP. In ARM, this function accesses the GIC registers to enable and set the priority of the interrupts. The GIC should not be accessed in user mode, and since the interrupts should be managed by the hypervisor, these accesses can simply be removed.

The BSP start also uses arm_cp15_set_exception_handler() to install the IRQ exception handler. As explained in the previous section, ARM's main trap table is already initialized with the default low level exception handlers. These default handlers are simple routines coded in assembly that assume that the exception was an error and therefore terminate the program. However, it might be desired that the exception handler calls a high level handler, adequate for the particular interrupt that occurred. For instance, the IRQ handler should distinguish a timer interrupt and redirect the execution to the clock handler. In those cases, RTEMS provides alternative low level exception handlers, prepared to store the context, redirect to a high level handler and finally restore the context to return to the normal execution. The function that installs the low level handlers is arm_cp15_set_exception_handler(). This function performs several accesses to the processor that should not be performed with AIR, so a simplified version was implemented and can be seen in Figure 4.5.

This function receives two arguments: the exception for which the handler should be replaced, and the routine that should replace the current handler. It declares, in line 10, a pointer to the memory region containing the addresses of the current handlers, and, if the current and the new handlers are different functions, replaces, in line 16, the address corresponding to the current exception handler by the address of the function received as an argument.

As for the initialization of device drivers, the only driver that should be configured is the clock. All the remaining I/O drivers should be managed by AIR through the IOP, and the header file containing the RTEMS configuration macros that was generated by the AIR configurator does not include any other device.

```
1  void arm_cp15_set_exception_handler(
2    Arm_symbolic_exception_name exception,
3    void (*handler)(void)
4  )
5  {
6    if ((unsigned) exception < MAX_EXCEPTIONS) {
7
8  #ifdef AIR_HYPERVISOR
9       uint32_t *tbr = (uint32_t *)bsp_vector_table_begin;
10      uint32_t *mirror_table = tbr + MAX_EXCEPTIONS;
11      uint32_t current_handler = mirror_table[exception];
12
13      if (current_handler != (uint32_t) handler) {
14          rtems_interrupt_level level;
15          rtems_interrupt_local_disable(level);
16          mirror_table[exception]=(uint32_t) handler;
17          rtems_interrupt_local_enable(level);
18      }
19
20 #else
21            (...)
```

Figure 4.5: Routine to install the exception handlers.

The clock initialization consists on, first, installing a clock handler to the corresponding interrupt ID. In the Cortex-A9 this is done through the a9mpcore_clock_handler_install() function, which does not require virtualization. This function is used to install the Clock_isr() routine as the exception handler for the Global Timer (GT) interrupt. The virtualization of Clock_isr() will be approached in Section 4.5.3.

Then, in a9mpcore_clock_initialize(), the GT registers are accessed in order to initialize the RTEMS timecounter. With AIR, this function is replaced by air_a9mpcore_clock_initialize, its virtualized version, presented in Figure 4.6.

```
1  static uint32_t air_clock_get_timecount(struct timecounter *tc) {
2      (void) tc;
3      return (uint32_t)air_syscall_get_elapsed_ticks()*
           rtems_configuration_get_microseconds_per_tick();
4  }
5
6  static void air_a9mpcore_clock_initialize(void) {
7      uint32_t now;
8      now = (uint32_t) air_syscall_get_elapsed_ticks();
9      air_clock_last = (uint32_t) now;
10
11     air_tc.tc_get_timecount = air_clock_get_timecount;
12     air_tc.tc_counter_mask = 0xffffffff;
13     air_tc.tc_frequency = 1000000/rtems_configuration_get_microseconds_per_tick();
14     air_tc.tc_quality = RTEMS_TIMECOUNTER_QUALITY_CLOCK_DRIVER;
15     rtems_timecounter_install(&air_tc);
16 }
```

Figure 4.6: Virtualization of the clock initialization and the timecounter tick.

This function, instead of accessing the real processor registers, uses the AIR system call in line 8 to get the time elapsed since the beginning of the partition, and set the AIR timecounter with the

appropriate data in lines 11 to 15.

This concludes the features that require virtualization in the system initialization. After going through all features configured for an application, the RTEMS initialization is complete, and the OS is ready to pass the control to the user application.

4.4 User Application

The application's init.c is generated by the configurator through the init.c.mako template. This file is part of the user application, designed to run in an unprivileged mode, so it does not require virtualization. However, since it was intended for SPARC, some modifications need to be applied so that it can run on ARM as well.

The Init() function, before redirecting the execution to the entry_point defined in the configuration, installs a HM handler. The reason why AIR installs a HM handler is so that the application programmer can change the HM Callback as desired. When AIR only supported the SPARC architecture, the HM handler was installed through the rtems_interrupt_catch() function. This function is only defined for CPU architectures with simply vectored interrupts, which, as previously discussed, is not the case for ARM. In order to keep Init() a common function between all architectures, the rtems_interrupt_catch() function was replaced by a new function, hm_handler_install(), which is defined in the same file. Figure 4.7 shows the implementation of install_hm_handler() in the init.c.mako template.

```
1  /**
2   * @brief Install the HM handler
3   */
4  %if os_configuration.arch == "arm":
5  void hm_handler_install(void *handler){
6      arm_cp15_set_exception_handler(ARM_EXCEPTION_DATA_ABORT, &_ARMV4_Exception_data_abort);
7      _ARMV4_Exception_data_abort_set_handler(handler);
8      arm_cp15_set_exception_handler(ARM_EXCEPTION_PREF_ABORT, &_ARMV4_Exception_prefetch_abort);
9      _ARMV4_Exception_prefetch_abort_set_handler(handler);
10     arm_cp15_set_exception_handler(ARM_EXCEPTION_UNDEF, &_ARMV4_Exception_prefetch_abort);
11 }
12 %else:
13 void hm_handler_install(void *handler){
14
15     rtems_isr_entry isr_ignored;
16     rtems_interrupt_catch(
17             (rtems_isr_entry)handler,
18             AIR_IRQ_HM_EVENT,
19             &isr_ignored);
20 }
21 %endif
```

Figure 4.7: Implementation of hm_handler_install in init.c.mako.

The Mako template allows for conditionals, and so, in case of configuring a target architecture other than ARM, the hm_handler_install() will consist of the same code that previously installed the handlers, as shown in lines 12 to 21 of Figure 4.7. If the target architecture is ARM, then the HM handler needs to be installed for each of the abort and undefined exceptions. This was achieved through the

arm_cp15_install_handler() function, used in lines 6, 8 and 10 to replace the RTEMS default exception handlers by the alternative low level handlers provided by RTEMS, as discussed in the previous section; followed by the routines to set the high level handler for each abort exception, in lines 7 and 9, which set the desired C handler that the low level exception handlers will redirect to.

RTEMS does not provide a handler for the undefined instructions other than the default handler. For this reason, and since this exception's handler should perform the same actions as the abort handlers, the low level handler installed for the undefined exception was the prefetch abort handler (line 10), as a simpler solution than implementing a new function with the same goal for the undefined exception.

4.5 Exception Handling

Having successfully initialized both RTEMS and the application, the next step is to guarantee that the exception handling is working correctly. Throughout the application run-time, the execution will be interrupted by exceptions. These exceptions can be periodic, such as the clock interrupts which will be raised as IRQ exceptions, or occur in particular events, such as an error that raises an abort or an undefined exception.

4.5.1 AIR Exception Handler

When an exception is raised, the execution jumps to the trap table in AIR, which branches to the AIR Exception Handler. This handler starts by storing the Interrupt Stack Frame (ISF) in the IRQ stack, as depicted in Figure 4.8, making the IRQ SP go from position SP_1 to SP_2. Then, AIR performs the adequate actions to handle the exception. IRQs will be handled by the ISR, while the HM handles aborts and undefined instructions, as well as FIQ exceptions as they are not currently being used by AIR. SVCs should only be handled by the hypervisor, therefore AIR will not redirect the execution to the RTEMS SVC handler, instead returning it to the application, following the procedure described in Figure 4.2 and not the one detailed in this section, corresponding to Figure 4.3.

Inside the ISR and the HM, if a) the exception was raised inside a partition, b) the virtual trap table is initialized, and c) the virtual exception is enabled both by the interrupt flags and by the exception's priority, then the handler defines the return address as the address of the corresponding entry in the virtual trap table configured in Section 4.3.2, rather than the instruction in which the exception was raised. This means that AIR recognizes that a virtual exception must be raised, and after handling the real exception, it will redirect the execution to the virtual trap table so that RTEMS can also perform its exception handling.

If AIR does not recognize a virtual exception, then the context will be restored from the stack, moving the IRQ SP back to the top of the ISF (position SP_1 in Figure 4.8) so that when the next exception occurs, its context will replace the one that has already been restored. The execution returns to the application, as in Figure 4.2.

If AIR recognizes a virtual exception, then the processor registers take the values that are stored in

the IRQ, but the SP remains at the bottom of the ISF (position SP_2) until the end of the virtual exception handling. This way, if another exception occurs inside the virtual handler (for instance, one of the several SVCs that are raised inside the virtualized RTEMS IRQ handler), it will be stored below the previous exception, not corrupting the context that has yet to be restored.

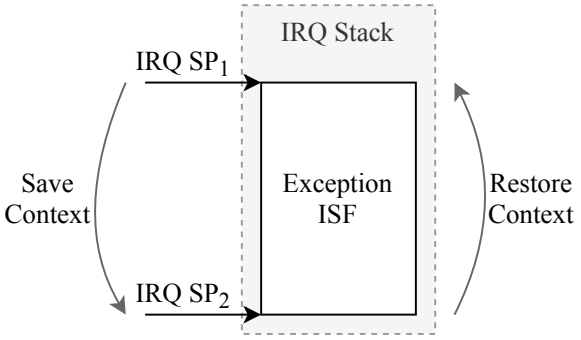

Figure 4.8: Single context storage in AIR exception handling.

4.5.2 RTEMS Exception Handlers

After the AIR exception handler sends the execution to the virtual trap table, the RTEMS exception handler will be performed. Currently, this applies to IRQs, aborts and undefined instructions, and the handlers for these exceptions required virtualization. The virtualization of the handler functions is similar, and mostly consists on preventing the processor to access the PSR registers, either by replacing these accesses with supervisor calls or, if these accesses are unnecessary when running on top of AIR, by removing them entirely.

Both the data abort and prefetch abort exception handlers save the previous context before calling the high level handler. Without hypervisor, the SPSR is saved and the CPSR is accessed to change to system mode in order to save the user SP and LR. With AIR, the AIR exception handler already has the previous context stored in the IRQ stack, so these instructions can be removed. It is then verified whether there is a high level handler installed for that exception. If not, the low level handler will call the _ARM_Exception_default function, which terminates the program. Otherwise, the installed high level handler will be executed. As described in Section 4.4, in the application initialization file generated by the AIR tools, the hm_handler was installed as the abort and undefined exceptions' high level handler, and will therefore be executed at this point. Afterwards, the context is restored and the exception is returned to normal execution. The virtualization of this section of the code consist on removing the write to SPSR and replacing the return from exception instruction for AIR_SYSCALL_RETT, further explained in Section 4.5.4.

Although the principle of virtualizing the IRQ handler is the same as for the abort handlers – that is, preventing accesses to the PSR and using AIR_SYSCALL_RETT to return to normal execution - the IRQ handler presents more challenges and will therefore be further detailed.

The main challenge in virtualizing the IRQ handler is that, without a hypervisor, both the SVC stack

and the IRQ stack are used to store data. When running on AIR, the whole function will run on user mode, therefore requiring adaptations to ensure that the data is not corrupted. Two possible solutions were tested:

- Adjusting the stack pointer so that the storage of data does not replace information that was previously stored;
- Virtualizing both the SVC stack and the IRQ stack, assigning each virtual stack a part of the user stack memory.

While the second option is safer in case of an unexpected access to the stack, it requires more modifications to AIR and more system calls throughout the IRQ handler function, as all the accesses to the CPSR need to be replaced by supervisor calls to access the virtual PSR, in order to change the virtual mode and point to the corresponding stack. The first solution is simpler, and, since the operating mode does not affect the execution, the accesses to the CPSR directed at changing the mode can be removed from the virtual IRQ handler, not needing to be replaced by system calls, which makes it more efficient. Since both methods seem to be achieving the same results, the first option is the currently implemented solution.

After the context is stored, bsp_interrupt_dispatch() is called. This function accesses the GIC registers to get the interrupt ID from the Interrupt Acknowledge Register (IAR). While the non-virtualized RTEMS accesses the real processor register, the AIR_SYSCALL_ACKNOWLEDGE_INT was implemented in AIR for ARM to get the interrupt ID from the virtual IAR. The bsp_interrupt_dispatch() function then executes the handler corresponding to that ID, which has previously been installed during the initialization. Currently, the only IRQ that is being passed from AIR to the RTEMS handler is the Global Timer (ID=27). The clock handler will be discussed further in Section 4.5.3.

Having completed the interrupt dispatch, the IRQ handler then verifies if a context switch is necessary by checking the thread dispatch state. If so, the control is passed to the dispatcher to perform the context switch. Although the dispatcher itself does not require virtualization, the context switch presents another challenge for the implementation of AIR_SYSCALL_RETT, which will be tackled in Section 4.5.4.

Finally, as with the abort exception handlers, the IRQ handler is concluded by restoring the previous context and returning from the exception, which is done through AIR by calling AIR_SYSCALL_RETT.

4.5.3 Clock

The precise response times that characterize RTOS are achieved both in AIR and RTEMS through periodic timer interrupts. When running RTEMS on top of AIR, since the interrupts are managed by the hypervisor, the time counting in RTEMS must be done through an AIR service rather than by accessing the real processor timer registers. AIR provides the AIR_SYSCALL_GET_ELAPSED_TICKS routine, which gets the number of ticks elapsed since the partition's initialization. This routine is used both in the clock initialization (approached in Section 4.3.3) and in the clock interrupt handler.

The RTEMS clock handler is Clock_isr(), installed as the handler for the Global Timer interrupt ID during the OS initialization. This routine is global to all architectures, and had already been virtualized

for AIR for SPARC. Although it uses functions that are specific for each architecture, it calls them using general macros that RTEMS defines for each BSP, making Clock_isr() itself hardware independent. Therefore, the virtualization of this function for SPARC is applicable for ARM as well, and is presented in Figure 4.9. The BSP specific functions that it calls are the ones who require modifications.

```
1  rtems_isr Clock_isr( rtems_vector_number vector )
2  {
3    /*
4     *  Accurate count of ISRs
5     */
6  #ifdef AIR_HYPERVISOR
7    /*
8     *  AIR control Control_driver_ticks
9     */
10
11   /* AIR sets accurate count of clock ISRs */
12   Clock_driver_ticks= (uint32_t) air_syscall_get_elapsed_ticks();
13
14   /*
15    *  Do the hardware specific per-tick action.
16    *
17    *  The counter/timer may or may not be set to automatically reload.
18    */
19   Clock_driver_support_at_tick();
20
21   /*
22    *  The driver is one ISR per clock tick.
23    */
24   Clock_driver_timecounter_tick();
25
26 #else /* AIR_HYPERVISOR */
27             (...)
```

Figure 4.9: Virtualization of the clock handler (Clock_isr).

The first RTEMS function is called through the macro Clock_driver_support_at_tick() in line 19 of Figure 4.9, which for the Cortex-A9 is defined as a9mpcore_clock_at_tick(). This function updates the global timer flag by accessing the real processor timer registers, and since this is prohibited in user mode, the virtualized function was left empty, as the corresponding function in SPARC.

Then, the Clock_driver_timecounter_tick() macro is used in line 24 to call a9mpcore_tc_tick(). This function was replaced by its virtualized version, air_a9mpcore_tc_tick(), shown in Figure 4.10.

The air_a9mpcore_tc_tick() routine gets the current time using the AIR system call in line 6 of Figure 4.10, compares it with the last time measurement stored in the global variable air_clock_last in line 8, and updates the timecounter accordingly in line 9 through the function rtems_timecounter_tick(), which is crucial to ensure that RTEMS scheduler correctly allocates the execution time to the right task.

4.5.4 Return from Exception

The virtualized exception handlers end by calling AIR_SYSCALL_RETT in order to perform a return to normal execution. The implementation of this SVC is different from the other exceptions. While other exceptions need to store the ISF in order to return to the previous context after the exception is handled,

```
1  static void air_a9mpcore_tc_tick(void)
2  {
3      uint32_t now;
4      uint32_t last;
5
6      now = (uint32_t)air_syscall_get_elapsed_ticks();
7      last = air_clock_last;
8      while (now != last) {
9          rtems_timecounter_tick();
10         last += 1;
11     }
12     air_clock_last = last;
13 }
```

Figure 4.10: Virtualization of the clock initialization and the timecounter tick.

AIR_SYSCALL_RETT does not need to return to the context before the SVC was called, but rather the context stored before that, corresponding to the exception from which the return is intended.

When an exception occurs, AIR stores the context in the IRQ stack, as explained in Section 4.5.1. If another exception is raised before the previous is returned, then the context of the second exception is stored below the context of the first, as represented in Figure 4.11. For example, when there is a timer interrupt, an IRQ exception is raised, and its context is stored, taking the place of ISF 1 in the figure, and moving the SP from SP_1 to SP_2. Inside the virtualized RTEMS IRQ handler, several SVCs are called, and their context is stored below the context of the IRQ exception in the IRQ stack, taking the place of ISF 2 and moving the SP to SP_3. The easiest way to implement the AIR_SYSCALL_RETT is simply skipping the routine that saves the context. This way, the ISF 2 will not be stored, and the SP stays in position SP_2. Unless RTEMS performed a context switch in the virtual exception handler, the SP is already pointing to the context that should be recovered, and it is only necessary to update the exception priority before getting to the routine that restores the context ISF 1.

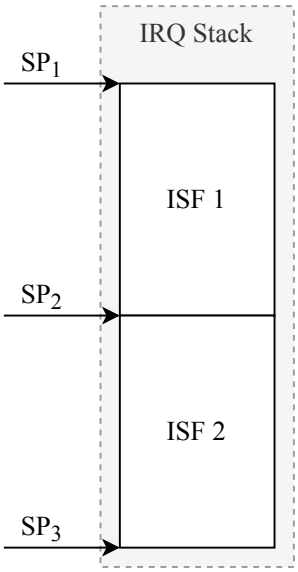

Figure 4.11: Storage of the context of an exception raised inside an exception handler.

Context Switch

In an application with multitasking, RTEMS performs context switches in the cases in which the scheduler deems it necessary to change the task that is allocated to the processor. This context switch is performed inside the RTEMS IRQ handler, and does not require virtualization. However, without accounting for this feature, AIR is not aware that a context switch occurred, and in the return system call, it will always restore the last context regardless of the context that RTEMS is trying to recover. This will lead to a different behaviour than the desired.

The solution to this problem relies on the difference between the user SP before and after the RTEMS exception handling. Recall from Section 3.4.4 that the user SP is separate from the IRQ SP, the former corresponding to the POS and the latter pointing to the current ISF in AIR. Since RTEMS is running exclusively in user mode, the contexts inside the OS are stored in the user stack. In context switches inside the OS, RTEMS changes the user SP to point to the context of the next task. This means that it is possible to verify if RTEMS performed a context switch in AIR_SYSCALL_RETT by comparing the current user SP with the user SP stored in the previous context. If the SP before the exception (the user SP stored in the ISF) differs from the SP at the end of the exception handler (the current user SP, which can be obtained by accessing system mode), then AIR knows that the OS performed a context switch and is trying to recover a different context. AIR can then go through all the stored contexts to find the one with the user SP that RTEMS means to restore. By changing the IRQ SP, AIR can restore the right context.

Figure 4.12 exemplifies this situation. In it, an IRQ was raised, its ISF stored (IRQ 1) and a context switch occurred inside the RTEMS IRQ handler. Since RTEMS changed the task allocated to the processor, the handler that was running did not reach AIR_SYSCALL_RETT, so the IRQ SP stays in position SP_1. When the next timer interrupt occurs (IRQ 2), the context will therefore be stored below IRQ 1, and the IRQ SP will take the position SP_2. Inside IRQ 2, RTEMS performed another context switch to change back to the context of IRQ 1, and RTEMS changed the user SP to restore this context. The execution will continue from where the first context switch interrupted, which is the IRQ handler, and will reach AIR_SYSCALL_RETT. The last context AIR stored was IRQ 2, and therefore the IRQ SP is in position SP_2. Without the solution presented in this section, AIR would restore IRQ 2, unaware that RTEMS is attempting to restore IRQ 1. With the presented solution, AIR obtains the user SP by accessing the system mode, and compares this value to the user SP stored in IRQ 2. As it is different, AIR is informed that a context switch occurred, and will search the IRQ stack for a context with the same user SP. Once it finds that IRQ 1 is the context to be restored, AIR moves the IRQ SP to position SP_1 to restore the right ISF.

However, this solution brings a problem of its own as the IRQ SP will no longer be pointing to the bottom of the stack. When another exception is raised, its context will take the place of the context that was last restored (IRQ 1), but when the exception handler calls SVCs, the SVC context will replace the context stored below (IRQ 2), corrupting data that might have yet to be recovered. To solve this issue, the way IRQs are stored was changed to leave enough space for another frame between two consecutive IRQs, as depicted in Figure 4.13. This way, when there is an SVC inside the IRQ handler, its context will

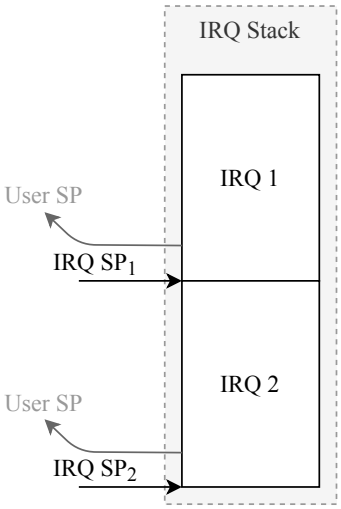

Figure 4.12: Context switch example.

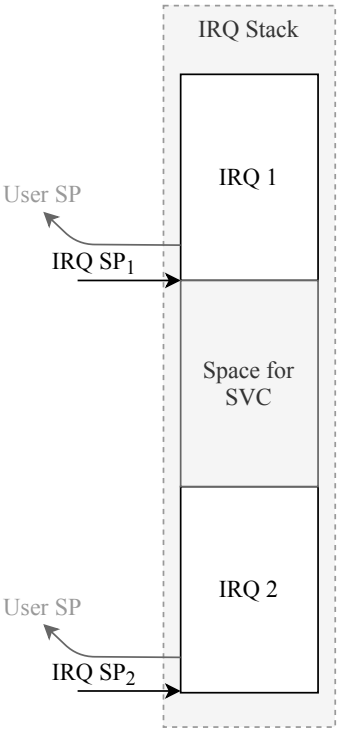

Figure 4.13: Current context switch solution.

be stored in this space and will not corrupt the contexts of the IRQs below that have yet to be restored.

Floating Unit Registers

As mentioned in Section 3.3.4, when AIR was first migrated to the ARM architecture, the FPU context was stored in the core context rather than the ISF, meaning that only one FPU context was stored per core, per partition. This did not take into account the multitask capabilities of the POS, which might be running different tasks with different FPU registers, and must be able to restore the FPU context as well as the general registers. For this reason, the FPU context was moved into the ISF, and the stack size increased in order to contain enough space for multiple FPU and general contexts.

4.6 End of Test

When the test ends, RTEMS shuts down the hardware. When running on top of AIR, the POS should not be allowed to shutdown the board, and that responsibility falls onto the hypervisor. AIR provides the AIR_SYSCALL_SHUTDOWN_MODULE, an SVC that should be called at the end of the user applications to terminate AIR and shutdown the machine.

RTEMS may also attempt to shut down the machine when a process level error occurs. In this case, the shutdown system call should not be used, as a process level error should not affect the other partitions or terminate AIR. Instead, the error should be caught by the HM. AIR provides the AIR_SYSCALL_RAISE_HM_EVENT, which receives as an input the ID of the error that should be raised. The AIR_UNIMPLEMENTED_ERROR was chosen as the error ID for this case. By using this system

call, the user can decide the action that should be taken through the HM tables in the configuration XML file, and install a HM Callback to resolve the fault.

Chapter 5

Evaluation

This chapter describes the tests that were performed to evaluate and validate the improved hypervisor. Section 5.1 introduces both platforms used to test the software, QEMU and Arty Z7. Then, the tests that were executed were categorized into three main groups:

The first group of tests, approached in Section 5.2, corresponds to the RTEMS Test Suites, a set of tests that verify several RTEMS features. Through these tests, the behaviour of the virtualized OS can be compared to the behaviour of RTEMS when running without a hypervisor.

The second group, discussed in Section 5.3, consists of AIR examples, both those designed for the ARM BSP and those migrated from SPARC. The former are the tests that were already executed on ARM with the barebones OS, and running them on the current software ensures not only that the basic OS functionalities are met by the virtualized RTEMS, but also that the modifications to the AIR hypervisor did not negatively impact the behaviour of the ARM BSP. The latter use RTEMS routines, so by running them on AIR for ARM the virtualized OS can be further tested, and the behaviour of AIR for both architectures can be compared.

The third group, presented in Section 5.4, includes the AIR validation tests, a set of tests developed in the scope of the AIR validation and qualification project to test the required functionalities of a TSP hypervisor.

5.1 Testing Environment

Throughout this dissertation, the modifications applied to RTEMS and AIR for ARM were tested through two platforms: QEMU and the Arty Z7 Board.

QEMU is an open source machine emulator [66], that using dynamic translation virtualization methods, allows software to be executed in machines other than the one it was designed for. It is capable of emulating a variety of different architectures, including almost fifty ARM machines, one of which is the Xilinx Zynq SoC equipped with the Cortex-A9, the target architecture of this work. Although QEMU is a useful tool to test software when the target machine is not available, it is not as accurate as the real hardware. The limitations of QEMU were mostly felt in the timings of execution, as QEMU does

not seem to have the precise notion of time that the board takes in performing certain actions. For this reason, the testing on the hardware was crucial to detect issues in the program which are not perceived by the emulator.

The Arty Z7 board [67] by Digilent is based on the Zynq-7000 All Programmable SoC from Xilinx. This architecture integrates a dual-core, 650 MHz ARM Cortex-A9 processor with Xilinx 7-series Field Programmable Gate Array (FPGA) logic, which allows several peripherals and controllers to be defined through design environments such as the Xilinx Vivado Design Suite. The block design conceived in Vivado and used in this dissertation is depicted in Figure 5.1. This design allows data to be transmitted through the UART 1 peripheral module by setting it as an external I/O. Through an intuitive design flow, Vivado can be used to generate an HDL wrapper for the designed configuration and compile it into a bitstream that can be programmed to the SoC's FPGA.

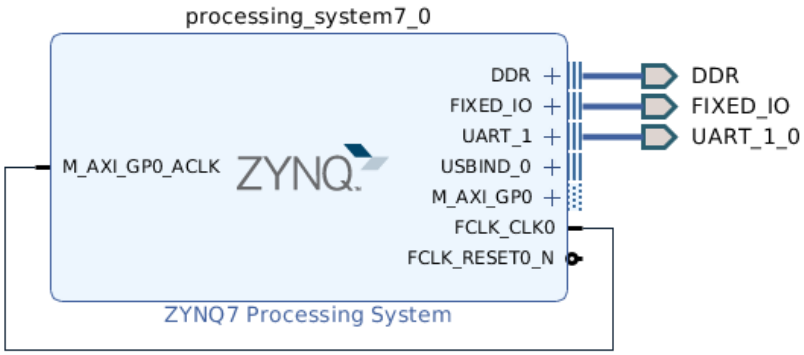

Figure 5.1: Vivado block design.

In order to run AIR on the Cortex-A9 and program the FPGA with the bitstream generated from the Vivado Design, this bitstream was exported to Xilinx's Software Development Kit (SDK), an Integrated Design Environment for the development of embedded applications, which provides tools to design, integrate, launch, and debug a software application in the target Zynq SoC.

5.2 RTEMS Test Suites

There are over 600 tests comprised in the RTEMS test suites, and without an automatized facility to migrate these programs to AIR and execute the tests, performing all of them would be a time consuming and exhaustive job. Therefore, only a few of these tests were selected.

First, some of the sample tests were used to start the basic virtualization. These include the hello, ticker, nsecs, and paranoia tests. Through them it was possible to virtualize and test the RTEMS initialization, exception handling without context switching, and the end of test. Despite their simplicity, the execution of these tests was critical as most of the modifications performed to RTEMS were decisions made based on the issues detected from them. In all of them, the functional behavior of the virtualized RTEMS corresponded to the behavior found in the original RTEMS.

5.2.1 Ticker

The ticker test was used to perform a basic timing analysis, comparing the results from the virtualized RTEMS running on AIR with those of the original RTEMS, both on the QEMU emulator and on the Arty Z7 board. This test is comprised in the RTEMS Sample Test Suite and consists on 3 tasks that periodically print the time in seconds:

- **TA1**, printing the time every 5 seconds;
- **TA2**, printing the time every 10 seconds;
- **TA3**, printing the time every 15 seconds.

To compare the original RTEMS with the virtualized RTOS, as well as to study the impact of partitioning in the response timings, the application was migrated into two partitions, splitting a major time frame of 2 seconds into two applications running for 1 second each. The example was configured at a frequency of 100 Ticks per second. As observable in Table 5.2.1, the results of both partitions correspond exactly to the expected, demonstrating that the partitioning did not affect the timings of execution.

	Original RTEMS	Virtualized RTEMS			
Task	Arty Z7	QEMU		Arty Z7	
		P1	P2	P1	P2
TA1	0,00	0,00	0,00	0,00	0,00
TA2	0,00	0,00	0,00	0,00	0,00
TA3	0,00	0,00	0,00	0,00	0,00
TA1	5,00	5,00	5,00	5,00	5,00
TA2	10,00	10,00	10,00	10,00	10,00
TA1	10,00	10,00	10,00	10,00	10,00
TA1	15,00	15,00	15,00	15,00	15,00
TA3	15,00	15,00	15,00	15,00	15,00
TA2	20,00	20,00	20,00	20,00	20,00
TA1	20,00	20,00	20,00	20,00	20,00
TA1	25,00	25,00	25,00	25,00	25,00
TA2	30,00	30,00	30,00	30,00	30,00
TA1	30,00	30,00	30,00	30,00	30,00
TA3	30,00	30,00	30,00	30,00	30,00

Table 5.1: Times printed (in seconds) in the ticker test, with a frequency of 100 ticks per second and two partitions.

To further analyze the influence of the frequency in the timings of the system, the test was repeated in AIR with different values of ticks per second. Table 5.2.1 presents the results. It is possible to observe that QEMU presents a consistent margin of error corresponding to the value of seconds per tick in one or more tasks. The Arty Z7 board seems to be more affected by the different frequencies, presenting errors of up to 18ms for the highest frequency tested, 9 times the value of seconds per tick.

The higher the frequency, the more times the AIR scheduling routine will be executed, introducing delays that need to be taken into consideration when integrating AIR in real time space applications, hence requiring these errors to be further investigated. However, it should be noted that these results

Task	50 Ticks/s	100 Ticks/s	200 Ticks/s		500 Ticks/s	
	QEMU & Arty Z7	QEMU & Arty Z7	QEMU	Arty Z7	QEMU	Arty Z7
TA1	0,00	0,00	0,000	0,000	0,000	0,000
TA2	0,00	0,00	0,000	0,000	0,000	0,000
TA3	0,00	0,00	0,000	0,000	0,000	0,000
TA1	5,02	5,00	4,995	5,005	5,002	5,002
TA2	10,02	10,00	9,995	10,000	10,000	10,002
TA1	10,02	10,00	9,995	10,005	10,002	10,006
TA1	15,02	15,00	15,000	15,005	15,002	15,010
TA3	15,02	15,00	14,990	15,000	15,000	15,004
TA2	20,02	20,00	19,995	20,000	20,000	20,004
TA1	20,02	20,00	20,000	20,005	20,002	20,014
TA1	25,02	25,00	25,000	25,005	25,002	25,016
TA2	30,02	30,00	29,995	30,000	30,000	30,006
TA1	30,02	30,00	30,000	30,005	30,002	30,018
TA3	30,02	30,00	29,990	30,000	30,000	30,006

Table 5.2: Times printed (in seconds) in the ticker test running on AIR for ARM, with varying frequencies and a single partition.

were obtained from console prints, which is not a reliable method for time analysis as the prints themselves may significantly affect the timings. It is therefore imperative to acquire tools to perform a more accurate time analysis before implementing this software in safety critical real time systems.

5.2.2 Pthread and spcontext

Once the sample tests achieved a behaviour similar to the one presented in the original RTEMS, two more complex tests were selected based on the recommendations of the RTEMS organization: pthread and spcontext.

The **pthread test** is part of the POSIX API Test Suite, designed to test the thread creation in conformity with the POSIX standard. This test performed as expected, presenting the same behavior on the original and the virtualized RTEMS, requiring no modifications to the software.

The **spcontext test** is comprised in the RTEMS Single Processor Test Suite, which are tests designed to provide a code coverage of more than 98% of the single processor code in RTEMS. The spcontext test was selected for its complexity by suggestion of the RTEMS organization, but in the future, more of these tests should be performed to ensure the correct virtualization of the single processor code. This test creates three tasks of different priorities and different FPU contexts, and activates a timer that switches the task priorities in periodic time intervals. The scheduler will therefore perform context switches, and the application verifies whether the context was correctly restored. This test was used to develop and validate the context switch solution in the AIR exception return, described in Section 4.5.4. Through this test it was possible to verify that this solution performs correctly, saving and restoring the correct contexts without corrupting data.

5.3 AIR Examples

AIR provides a set of examples to test the basic functionalities of the hypervisor. These examples do not have hardware dependencies, but can still be split into the tests that were designed to test the ARM BSP and those that were originally developed for SPARC.

5.3.1 ARM unit tests

The ARM unit tests of the AIR example set were first designed to test the initial migration of AIR to the ARM architecture, when only a barebones OS was available. By running these applications with RTEMS, it is possible to compare the behaviour of the two operating systems, as well as to verify that the modifications applied to AIR did not negatively impact the hypervisor. All of the single processor ARM unit tests were successfully deployed on the current software, being the two most relevant the **time** and the **HM** examples.

Time example

RTEMS offers mechanisms to create periodic tasks and to sleep for a defined period of time. The barebones OS does not provide such functionalities, and since the **time example** was developed for the barebones OS, it does not use functions with precise timings, but rather relies on loops to print the time at unpredictable intervals, as presented in the code of Figure 5.2. Therefore, this test will not allow an accurate timing analysis, but will present a basic means of comparison of the speeds of the barebones OS and the virtualized RTEMS.

```
1     /*Measurement m*/
2  for(m = 0; m < 8; m++) {
3     for(i = 0; i < wait_loops[m]; ++i) {
4      /*Do nothing*/
5     }
6     tick = air_syscall_get_elapsed_ticks();
7     t = tick * ms_per_tick;
8     printf("time = %d\n", t);
9  }
```

Figure 5.2: Code of the measurements in the time example of the ARM unit tests.

The times printed by this test in a single partition are presented in Table 5.3.1 in milliseconds, t referring to the time stamp at the print and Δt to the time passed since the last stamp.

From these results it is possible to observe that RTEMS is consistently slower than the barebones OS. These results are expected, considering that both the initialization and the exception handling in RTEMS are significantly more complex than the barebones OS's. The RTEMS initialization presents an overhead of 9ms, 3 times the overhead of the barebones OS. After the initialization, RTEMS consistently takes added 0,11ms for each ms of the execution of the barebones OS. This consistency is a good indicator of the predictability of RTEMS.

Measurement		Arty Z7				
		BARE		RTEMS		$\frac{\Delta t_{RTEMS} - \Delta t_{BARE}}{\Delta t_{BARE}}$
m	wait_loops	$t(ms)$	$\Delta t(ms)$	$t(ms)$	$\Delta t(ms)$	
0	0	3	0	9	0	2,0000
1	16777215	5136	5133	5709	5700	0,1105
2	33554431	15403	1026	17103	11394	0,1098
3	83886079	41059	25656	45585	28482	0,1101
4	117440511	76976	35917	85451	39866	0,1099
5	1048575	77303	327	85814	363	0,1101
6	50331647	92700	15387	102902	17088	0,1098
7	50331647	108096	15396	119994	17092	0,1102

Table 5.3: Times printed (in milliseconds) in the time example of the ARM unit tests.

HM example

The other relevant ARM unit test is the **HM example**, in which different errors can be induced to ensure that the HM is capable of adequately containing and handling them through the actions defined in the HM tables of the configuration XML. Both partition and module level errors were tested and AIR performed as expected, as presented in Table 5.3.1.

Level	Action	Results
Partition	IDLE	Stops the partition and enters IDLE mode.
	WARM START COLD START	Restarts the partition.
	IGNORE	Performs the HM Callback and returns to execution.
Module	SHUTDOWN	Shuts down the module.
	RESET	Restarts the module.
	IGNORE	Returns to the instruction that raised the error.

Table 5.4: Results obtained from the HM ARM unit test.

The test was performed with 1 and 2 partitions, obtaining the same results. The partition level errors were successfully contained inside the partition in which they were raised, while the module level errors affect the whole module. Errors inside the partition HM Callback were also tested, performing as expected, achieving the same results as the partition level errors in Table 5.3.1.

5.3.2 Tests migrated from SPARC

The tests that were developed for SPARC use RTEMS routines, hence why they had not previously been deployed in AIR for ARM. After achieving a virtualized RTEMS for the ARM architecture, these examples can be effortlessly deployed in the target hardware, making it possible to perform a basic comparison of the behavior of AIR for both architectures. The SPARC based hardware used to perform this comparison was GR740, a rad-hard SoC featuring a LEON4 SPARC processor that is provided by GMV.

The tests executed include **hello_world**, **periodic**, **ports** and **shm** (shared memory).

The virtualization of RTEMS should not affect the inter-partition communication nor the AIR shared memory, and the **ports** and **shm** tests ensured that these functionalities remained unchanged.

The **periodic** example tests the creation of periodic tasks through RTEMS functions, ensuring that these features provided by the Task and the Rate Monotonic managers are correctly virtualized. These tests all performed as expected, exhibiting same behavior as AIR for SPARC.

The **hello_world** example provided by AIR prints not only the classical hello message but also the time, and it was used to compare the timings of execution of the two ARM platforms with the SPARC board. It consists on 3 partitions (P1, P2 and P3), each performing a loop of 10 iterations in which they print the hello message and the time at intervals of 0,1 seconds. The configured schedule uses a single core (CPU 0) and defines a major time frame of 1 second, assigning 0,3 seconds to each partition, at a rate of 500 ticks per second. Figure 5.3 represents the first major time frame of this example's schedule. As depicted, the time counters in each partition are configured to count the global time, meaning the total time elapsed since the partition start (including the time in which the partition is not active). The diamond shapes in the diagram represent each print of the time.

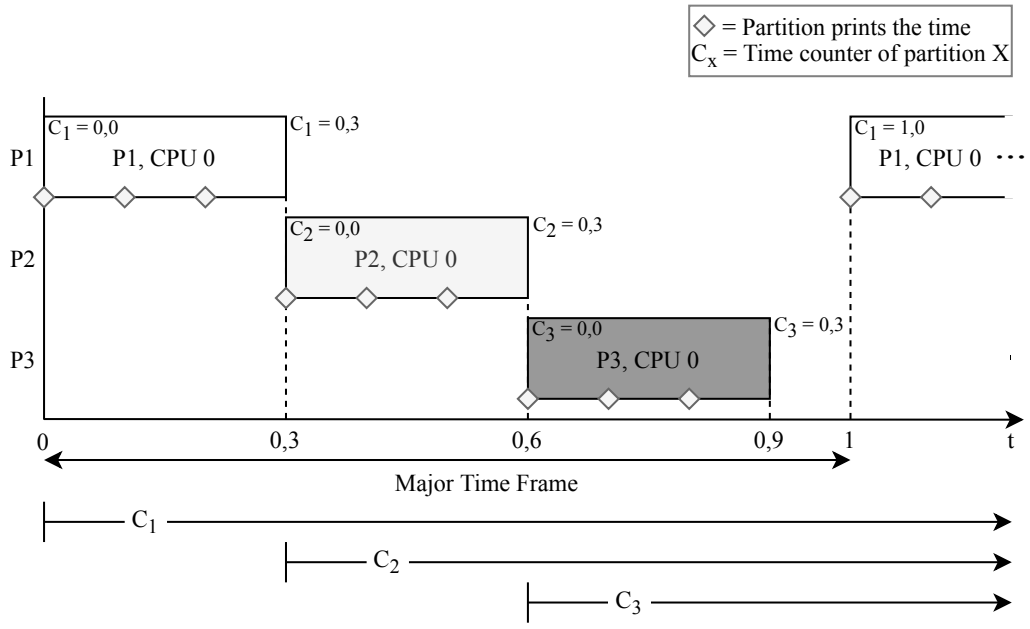

Figure 5.3: Diagram of the Hello_World example schedule.

Table 5.3.2 shows the times, in seconds, printed in the tests. The horizontal lines in between the values represent partition switches. Although the applications are not running at the same time, since each time counter starts at the start of the partition, the three partitions can be viewed as parallel.

There is a considerable difference of up to 50ms between the results obtained from GR740 and those of the Arty Z7 board. One possible reason might be the fact that AIR for SPARC is compiled using optimization flags to improve its performance, while AIR for ARM is currently being compiled without optimization options to ease the debugging. Once AIR for ARM achieves a higher maturity and stability, these options should be explored for a more accurate performance comparison.

SPARC			ARM					
GR740			QEMU			Arty Z7		
P1	P2	P3	P1	P2	P3	P1	P2	P3
0,004	0,006	0,006	0,016	0,014	0,014	0,010	0,010	0,010
0,104	0,104	0,104	0,116	0,114	0,114	0,116	0,116	0,116
0,204	0,208	0,204	0,216	0,214	0,214	0,222	0,222	0,222
1,008	1,008	1,008	1,006	1,006	1,004	1,044	1,044	1,044
1,108	1,108	1,108	1,102	1,102	1,106	1,150	1,150	1,150
1,208	1,208	1,214	1,202	1,202	1,206	1,256	1,256	1,256
2,008	2,008	2,008	2,006	2,006	2,006	2,044	2,044	2,044
2,108	2,108	2,108	2,102	2,106	2,106	2,150	2,150	2,150
2,208	2,208	2,208	2,202	2,206	2,206	2,256	2,256	2,256
3,008	3,008	3,008	3,006	3,006	3,008	3,044	3,044	3,044

Table 5.5: Times printed (in seconds) in the Hello World example provided by AIR.

In the Arty Z7 board, the timings of the three partitions were exactly the same, showing that, although the reason for the lower speeds than those obtained in SPARC must be studied, the system seems to behave in predictable timings, which is the main concern in real time applications.

5.4 AIR Validation Tests

The qualification of AIR for space applications by a recognized entity such as ESA would bring significant value to the hypervisor. To achieve this in the future, extensive testing is required, and the AIR validation and qualification project was started to implement a set of tests with a nearly total code coverage with the goal of validating the required functionalities of a TSP hypervisor.

Currently the AIR validation tests comprise 38 tests designed for the SPARC architecture. 21 of these tests have been successfully executed on AIR for ARM, presenting the same results in both supported architectures:

- **TEST-DEF-00021** Tests different partition mode transitions. Verifies that invalid transitions raise HM errors and that the valid mode transitions are performed correctly.
- **TEST-DEF-00022**: Verifies that a HM error is raised when one partition attempts to change the mode of another.
- **TEST-DEF-00023**: Tests the GET_A_PARTITION_ID syscall.
- **TEST-DEF-00500**: Tests if a branch into a non-code address raises a HM error. Checks that the remaining partitions are not affected.
- **TEST-DEF-00510**: Tests if a branch into an unaligned address raises a HM error. Checks that the remaining partitions are not affected
- **TEST-DEF-01390**: Tests a schedule change during the application execution

- **TEST-DEF-01590**: Tests the TSP Abstraction Layer Initialization (TSAL_INIT) service
- **TEST-DEF-01620**: Tests the Module Level Recovery (RESET)
- **TEST-DEF-01630**: Tests the Module Level Recovery (SHUTDOWN)
- **TEST-DEF-01650**: Tests the SET_PARTITION_MODE system call to an invalid mode.
- **TEST-DEF-01730**: Tests the inter-partition communication through multicast messages
- **TEST-DEF-01740**: Tests READ_UPDATED_SAMPLING_MESSAGE function.
- **TEST-DEF-01741**: Tests GET_SAMPLING_PORT_CURRENT_STATUS function.
- **TEST-DEF-01750**: Tests the routines to get the module schedule ID and status.
- **TEST-DEF-02100**: Tests Multiple Module Schedules (MMS) Data Types.
- **TEST-DEF-02101**: Test the SET_MODULE_SCHEDULE without partition restart.
- **TEST-DEF-02102**: Tests the SET_MODULE_SCHEDULE with partition restart.
- **TEST-DEF-02105**: Tests the SET_MODULE_SCHEDULE with invalid schedule identifiers and partition privilege.
- **TEST-DEF-02107**: Tests the GET_MODULE_SCHEDULE_ID with valid parameters.
- **TEST-DEF-02108**: Tests the GET_MODULE_SCHEDULE_ID with invalid parameters
- **TEST-DEF-80060**: Tests the system partition memory permissions.

From the tests that are currently running successfully on AIR for ARM it can be concluded that the basic hypervisor requirements are met. The validation tests that still need to be investigated and adapted to the ARM architecture include:

- 3 tests that fail due to timing constraints, as AIR for ARM presents different timings than AIR for SPARC, as discussed in Section 5.3;
- 9 tests that rely on SPARC specific language and that have yet to be adapted to run on ARM;
- 5 tests that use cache register handling system calls that have not yet been implemented in AIR for ARM.

Chapter 6

Conclusions

Prior to this dissertation, AIR for ARM only supported a simple barebones guest OS that lacked important functionalities for the deployment on high profile space missions. The goal of this thesis was to further develop the ARM BSP for AIR by allowing it to run RTEMS, the RTOS currently adopted by ESA and NASA in a variety of space missions. This objective was successfully achieved through the virtualization of RTEMS, and through applying the necessary modifications to the AIR hypervisor to ensure that it does not interfere with the behavior of the guest OS.

6.1 Achievements

From a functional standpoint, the virtualization of RTEMS was successfully accomplished. The changes on RTEMS achieved a virtualized OS that behaves as the original RTOS running in a non virtualized environment. This virtualization made AIR for ARM capable of running single core applications designed for this OS, including a wide range of tests that were previously not available in this target architecture due to the simplicity of the barebones OS and that allow the further validation of the hypervisor. These tests include RTEMS Testsuites, AIR examples and the validation tests developed to test AIR for SPARC with a nearly total code coverage. Of the 38 validation tests currently executing in AIR for SPARC, 21 were successfully deployed in AIR for ARM. A comparison between the two architectures currently supported by AIR showed that AIR for ARM achieved the same results regarding the hypervisor functionalities as AIR for SPARC.

From a temporal perspective, the virtualized RTOS allowed for a more detailed analysis of the timings of execution, showing the limitations that the ARM BSP still presents and that require further investigation through the usage of profiling and evaluation tools capable of more accurate timing analysis.

Additionally to the software development, the iterative process used to achieve the stated goals of this dissertation was documented step-by-step, providing a resource that can be used for guiding the virtualization of RTOS in future projects.

6.2 Future Work

While this dissertation further developed the ARM BSP for the AIR hypervisor, there is still work to be done in order to achieve the maturity that the SPARC BSP presents. The three next steps, currently in progress, are virtualizing the RTEMS support for multi-core applications, expanding the number of device drivers supported by AIR for ARM, and executing the 17 remaining validation tests that have yet to run successfully on AIR for ARM, either by adapting them to the ARM architecture or implementing the missing system calls in AIR for ARM.

Although from a functional perspective the behavior of the software corresponds to the expected, the timings of execution still require further study, and in that prospect, it is crucial to acquire tools for a more accurate timing analysis, such as RapiTime [68] or VectorCAST [69], and apply them to AIR to ensure its timely behavior.

In the long term, the qualification of AIR for space applications by ESA would bring value and recognition to the hypervisor. To achieve this, extensive testing is needed, and therefore tools to automatically test, validate and verify AIR should be implemented. A possible tool that is being studied is the Continuous Integration and Continuous Development (CI/CD) tools built into GitLab Runner, that can be used to perform scripts and send the results back to GitLab. The possibility of taking advantage of GitLab Runner along with tools to generate validation tests with the highest code coverage achievable would provide a fast, easy and reliable platform for effortlessly weaving the testing into the development process, allowing the immediate detection of issues and assisting their solution.

Bibliography

[1] H. Butz. Open integrated modular avionic (IMA): State of the art and future development road map at airbus deutschland". *1st International Workshop on Aircraft System Technologies*, 2010.

[2] Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface Part 1 - Required Services, December 2005.

[3] GMV Innovating Solutions SL. AIR: ARINC 653 in real time. [Online] `https://www.gmv.com/en/Products/air/`, . Accessed: 24 February 2020.

[4] B. Gomes, D. Silveira, L. Gouveia, and L. Mendes. AIR hypervisor using RTEMS SMP. *European Workshop on On-Board Data Processing*, 2019.

[5] M. Muñoz, G. Montano, M. Wirkus, K. Höflinger, D. Silveira, N. Tsiogkas, J. Hugues, H. Bruyninckx, I. Dragomir, and A. Muhammad. ESROCOS: A Robotic Operating System For Space And Terrestrial Applications. June 2017.

[6] L. Mendes. Hypervisor board support package migration. Master's thesis, Instituto Superior Técnico, May 2019.

[7] INFANTE Space. INFANTE – Satellite for Maritime Applications, 2019. [Online] `http://www.infante.space/`. Accessed: 24 February 2020.

[8] Portugal Space. PT Space Website, 2019. [Online] `https://www.ptspace.pt/`. Accessed: 5 August 2020.

[9] RTEMS Project and contributors. RTEMS User Manual, 2020. [Online] `https://docs.rtems.org/branches/master/user/index.html`. Accessed: 24 February 2020.

[10] RTEMS Project and contributors. RTEMS Qualification Project, 2017. [Online] `https://qualification.rtems.org`. Accessed: 24 February 2020.

[11] RTEMS Project and contributors. RTEMS documentation project, 2018. [Online] `https://docs.rtems.org/`. Accessed: 24 February 2020.

[12] RTEMS Project and contributors. RTEMS historical timeline, 2020. [Online] `https://devel.rtems.org/wiki/History/Timeline`. Accessed: 24 February 2020.

[13] H. Butz. The airbus approach to open modular avionics (IMA): Tehcnology, methods, processes and future road map. *Department of Avionic Systems at Airbus Deutschland GmbH*, 2007.

[14] C. B. Watkins and R. Walter. Transitioning from federated avionics architectures to Integrated Modular Avionics. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 2007.

[15] H. Kuqshal. An Approach to Electrical Integration: Integrated Modular Avionics. *A Workshop on Futuristic Aerospace Vehicles Integration and Testing*, 2014.

[16] Y. Li, W. Li, and C. Jiang. A survey of virtual machine system: Current technology and future trends. *Electronic Commerce and Security, International Symposium*, July 2010.

[17] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 1970.

[18] R. P. Goldberg. Survey of virtual machine research. *Computer*, June 1974.

[19] J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, New York, USA, 1973. Association for Computing Machinery.

[20] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, July 1974.

[21] R. Mijat and A. Nightingale. Virtualization is coming to a platform near you. White Paper, ARM Limited, 2011. [Online] `https://virtualization.network/Resources/Whitepapers/62e6d178-424d-4faa-81e2-98ef3833313a_System-MMU-Whitepaper-v8.0.pdf`. Acessed: 7 June 2020.

[22] VMWare. Understanding full virtualization, paravirtualization, and hardware assist. White paper. [Online] `https://www.vmware.com/techpapers/2007/understanding-full-virtualization-paravirtualizat-1008.html`, March 2008. Accessed: 7 June 2020.

[23] Airlines Electronic Engineering Committee. Avionics Application Software Standard Interface Part 2 - Extended Services, March 2008.

[24] The Consultative Committee for Space Data Systems. Spacecraft Onboard Interface Services. In *Ccsds 850.0-G-2*, pages 1–88. CCSDS Secretariat, Washington DC, 2007.

[25] J. Windsor. IMA-SP and Security Status of Activies & Roadmap - 4th ESA Workshop on Avionics, Data, Control and Software Systems , 2010. [Online] `https://indico.esa.int/event/63/contributions/2842/`. Accessed: 7 June 2020.

[26] R. Kaiser and S. Wagner. Evolution of the PikeOS microkernel. *International Workshop on Microkernels for Embedded Systems*, January 2007.

[27] M. Masmano, I. Ripoll, and A. Crespo. An overview of the xtratum nanokernel. *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, January 2005.

[28] D. J. A. F. Miranda, M. A. Ferreira, F. Kucinskis, and D. McComas. A Comparative Survey on Flight Software Frameworks for New Space Nanosatellite Missions. *Journal of Aerospace Technology and Management*, 11, 2019. ISSN 2175-9146.

[29] Space Avionics Open Interface Architecture. SAVOIR-IMA. [Online] `https://savoir.estec.esa.int/`, 2017. Accessed: 7 June 2020.

[30] K. G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE*, 1994.

[31] M. H. M. Cheng. A Predictable Real Time Operating System. University of Victoria, October 2003. [Online] `https://webhome.csc.uvic.ca/~mcheng/research/predictable.pdf`. Accessed: 7 June 2020.

[32] L. Beus-Dukic. COTS Real-Time Operating Systems in Space. *Safety Systems: The Safety-Critical Systems Club Newsletter*, pages 1–5, 2001.

[33] E. G. Stassinopoulos and K. A. Label. The Near-Earth Space Radiation Environment for Electronics. *Info Espacio, Boletin Informativo Space Magazine*, pages 1–6, 2004.

[34] O. Hoftberger. Design Approaches for Radiation Hardening in Digital Circuits. December 2014.

[35] A. S. Keys, J. H. Adams, D. O. Frazier, M. C. Patrick, M. D. Watson, M. A. Johnson, J. D. Cressler, and E. A. Kolawa. Developments in Radiation-Hardened Electronics Applicable to the Vision for Space Exploration. pages 1–10, 2007.

[36] T. M. Lovelly. *Comparative Analysis of Space-Grade Processors*. PhD thesis, University of Florida, 2017.

[37] J. Krywko. Space-grade CPUs: How do you send more computing power into space? *Ars Technica*, November 2019. [Online] `https://arstechnica.com/science/2019/11/space-grade-cpus-how-do-you-send-more-computing-power-into-space/`. Accessed: 10 August 2020.

[38] The European Space Agency. Leading up to LEON: ESA's first microprocessors, 2013. [Online] `https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Leading_up_to_LEON_ESA_s_first_microprocessors`. Accessed: 10 August 2020.

[39] J. Gaisler. 25 Years of SPARC - a personal introspective, 2017. [Online] `https://indico.esa.int/event/182/contributions/1526/attachments/1400/1625/0905_-_Gaisler.pdf`. Accessed: 10 August 2020.

[40] J. Gaisler. Fault-Tolerant and Radiation Hardened SPARC Processors. [Online] https://indico.cern.ch/event/11994/contributions/84457/attachments/63921/91833/P2_Gaisler.pdf. Accessed: 10 August 2020.

[41] J. Gaisler. A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8Architecture. *Proceedings International Conference on Dependable Systems and Networks*, 2002.

[42] W. Powell. High-Performance Spaceflight Computing (HPSC) Project Overview. Radiation Hardened Electronics Technology (RHET) Conference, 2018. [Online] https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20180007636.pdf.

[43] J. Keller. Air force, nasa to develop radiation-hardened arm processor for next-generation space computing. *Military & Aerospace Electronics*, June 2016.

[44] C. Christopherson. New ARM-based computing system to enable deep space missions. *ARM Research*, March 2018.

[45] DAHLIA. Deep sub-micron microprocessor for spAce rad-Hard appLIcation Asic, 2017. [Online] https://dahlia-h2020.eu/. Accessed: 7 June 2020.

[46] J. Poupat, T. Helfers, P. Basset, A. G. Llovera, M. Mattavelli, C. Papadas, and O. Lepape. DAHLIA - very high performance microprocessor for space applications. *IEEE Conference on Space Mission Challenges for Information Technology*.

[47] U.S. Army. Real Time Executive for Military Systems, 1993. [Online] https://www.rtems.org/sites/default/files/RTEMS_Army_Brochure_1993.pdf. Accessed: 7 June 2020.

[48] H. Silva, J. Sousa, D. Freitas, S. Faustino, A. Constantino, and M. Coutinho. RTEMS Improvement-Space Qualification of RTEMS Executive. *Library*, 2009.

[49] K. K. Sheridan-Barbian. A survey of real-time operating systems and virtualization solutions for space systems. [Online] https://core.ac.uk/download/pdf/36737386.pdf. Accessed: 7 September 2020.

[50] G. Bloom and J. Sherrill. Scheduling and thread management with rtems. *SIGBED Review, Association for Computing Machinery*, February 2014.

[51] J. Sherrill. First Images From NASA Solar Dynamic Observatory, April 2010. [Online] https://www.rtems.org/node/47. Accessed: 7 June 2020.

[52] The European Space Agency. Software Engineering and Standardization - Operating Systems. [Online] http://www.esa.int/TEC/Software_engineering_and_standardisation/TECLUMKNUQE_2.html. Accessed: 7 June 2020.

[53] Digilent. ZedBoard Zynq-7000 Development Board Reference Manual. [Online] https://reference.digilentinc.com/reference/programmable-logic/zedboard/reference-manual. Accessed: 6 July 2020.

[54] The European Space Agency. IMA Separation Kernel Qualification - preparation. [Online] https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/IMA_Separation_Kernel_Qualification_-_preparation, 2016. Accessed: 11 August 2020.

[55] GMV Innovating Solutions SL. AIR Repository, . [Online] https://spass-git-ext.gmv.com/AIR. Accessed: 7 November 2019.

[56] GMV Innovating Solutions SL. GMV participates in the payload of the Infante satellite, 2017. GMV News [Online] https://www.gmv.com/en/Company/Communication/News/2017/12/Infante.html. Accessed; 11 August 2020.

[57] PLD Space. MIURA Webpage, 2018. [Online] http://pldspace.com/new/2018/11/13/pld-space-miura/. Accessed: 24 February 2020.

[58] B. Walshe. A brief history of ARM: Part 1, 2015. [Online] https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-1. Accessed: 6 May 2020.

[59] B. Walshe. A brief history of ARM: Part 2, 2015. [Online] https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/a-brief-history-of-arm-part-2. Accessed: 6 May 2020.

[60] I. Thornton. ARM: Investing for future growth, ARM limited Q1 2019. [Online] https://www.arm.com/-/media/global/company/investors/PDFs/Arm_SBG_Q1_2019_Roadshow_Slides_FINAL.pdf. Accessed: 6 May 2020.

[61] ARM Limited. ARM ® Architecture Reference Manual ARMv7-A and ARMv7-R edition ARM Architecture Reference Manual, 2018.

[62] Xilinx. Zynq-7000 SoC Technical Reference Manual, July 2018. [Online] https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Acessed: 6 July 2020.

[63] ARM Limited. ARM Generic Interrupt Controller Architecture Specification, 2013.

[64] RTEMS Project and contributors. RTEMS Coverage Analysis, 2018. [Online] https://devel.rtems.org/wiki/GCI/Documentation/CoverageAnalysis/Coverage. Acessed: 7 June 2020.

[65] RTEMS Project and contributors. Test Suites, 2020. [Online] https://docs.rtems.org/branches/master/eng/test-suites.html. Accessed: 7 September 2020.

[66] Qemu. [Online] https://www.qemu.org/. Accessed: 6 July 2020.

[67] Arty Z7 Reference Manual. Digilent. [Online] https://reference.digilentinc.com/reference/programmable-logic/arty-z7/reference-manual. Accessed: 6 July 2020.

[68] Rapita Systems Ltd. RapiTime Webpage, 2020. [Online] https://www.rapitasystems.com/products/rapitime. Accessed: 7 September 2020.

[69] Vector Informatik GmbH. VectorCAST Webpage, 2020. [Online] https://www.vector.com/int/en/products/products-a-z/software/vectorcast/. Accessed: 7 September 2020.

Appendix A

RTEMS Code

```
1  bsp_start_vector_table_begin:
2     ldr   pc, handler_addr_reset
3     ldr   pc, handler_addr_undef
4     ldr   pc, handler_addr_swi
5     ldr   pc, handler_addr_prefetch
6     ldr   pc, handler_addr_abort
7     /* Program signature checked by boot loader */
8     .word 0xb8a06f58
9     ldr   pc, handler_addr_irq
10    ldr   pc, handler_addr_fiq
11
12 handler_addr_reset:
13 #ifdef BSP_START_RESET_VECTOR
14    .word BSP_START_RESET_VECTOR
15 #else
16    .word _start
17 #endif
18
19 handler_addr_undef:
20    .word _ARMV4_Exception_undef_default
21
22 handler_addr_swi:
23    .word _ARMV4_Exception_swi_default
24
25 handler_addr_prefetch:
26    .word _ARMV4_Exception_pref_abort_default
27
28 handler_addr_abort:
29    .word _ARMV4_Exception_data_abort_default
30
31 handler_addr_reserved:
32    .word _ARMV4_Exception_reserved_default
33
34 handler_addr_irq:
35    .word _ARMV4_Exception_interrupt
36
37 handler_addr_fiq:
38    .word _ARMV4_Exception_fiq_default
```

Figure A.1: RTEMS trap table at the start.S file.

Appendix B

AIR Code

```
<ARINC_653_Module ModuleName="bare">
   <!-- partition 0 -->
   <Partition PartitionIdentifier="1" PartitionName="p0"
        Criticality="LEVEL_A" SystemPartition="false" EntryPoint="entry_point">
      <PartitionConfiguration Personality="BARE" Cores="1">
         <Libs>LIBAIR;LIBPRINTF</Libs>
         <Cache>CODE; DATA</Cache>
         <Memory Size="0x2000000" />
         <Permissions>
            FPU_CONTROL; CACHE_CONTROL; GLOBAL_TIME; SET_TOD; SET_PARTITION_MODE;
         </Permissions>
      </PartitionConfiguration>
   </Partition>

   <!-- partition 1 -->
   <Partition PartitionIdentifier="2" PartitionName="p1"
        Criticality="LEVEL_A" SystemPartition="false" EntryPoint="entry_point">
      <PartitionConfiguration Personality="RTEMS5" Cores="1">
         <Libs>LIBAIR;LIBPRINTF</Libs>
         <Cache>CODE; DATA</Cache>
         <Memory Size="0x2000000" />
         <Permissions>
            FPU_CONTROL; CACHE_CONTROL; GLOBAL_TIME; SET_TOD; SET_PARTITION_MODE;
         </Permissions>
      </PartitionConfiguration>
   </Partition>

   <!-- schedule 0 -->
   <Module_Schedule ScheduleIdentifier="1" ScheduleName="test_sched" MajorFrameSeconds="1.0">
      <Partition_Schedule PartitionIdentifier="2" PartitionName="p1"
           PeriodSeconds="1.0" PeriodDurationSeconds="0.5">
         <Window_Schedule WindowIdentifier="101" WindowStartSeconds="0.0"
             WindowDurationSeconds="0.5"
              PartitionPeriodStart="true"/>
         <WindowConfiguration WindowIdentifier="101" Cores="0" />
      </Partition_Schedule>
      <Partition_Schedule PartitionIdentifier="1" PartitionName="p0"
           PeriodSeconds="1.0" PeriodDurationSeconds="0.5">
         <Window_Schedule WindowIdentifier="201" WindowStartSeconds="0.5"
             WindowDurationSeconds="0.5"
              PartitionPeriodStart="true"/>
         <WindowConfiguration WindowIdentifier="201" Cores="0" />
      </Partition_Schedule>
   </Module_Schedule>

   <!-- module configuration -->
   <AIR_Configuration TicksPerSecond="100" RequiredCores="1"/>
```

Figure B.1: Application configuration XML example.

```
<!-- HM configuration -->
<System_HM_Table>
  <System_State_Entry Description="PMK Execution" SystemState="1">
    <Error_ID_Level Description="Power Interrupt" ErrorIdentifier="0" ErrorLevel="MODULE"/>
    <Error_ID_Level Description="Illegal Instruction" ErrorIdentifier="1"
        ErrorLevel="MODULE"/>
    <Error_ID_Level Description="Segmentation Error" ErrorIdentifier="2" ErrorLevel="MODULE"/>
    <Error_ID_Level Description="Unimplemented Error" ErrorIdentifier="3"
        ErrorLevel="MODULE"/>
    <Error_ID_Level Description="Floating Point Error" ErrorIdentifier="4"
        ErrorLevel="MODULE"/>
    <Error_ID_Level Description="Overflow Error" ErrorIdentifier="5" ErrorLevel="MODULE"/>
    <Error_ID_Level Description="Divide by zero" ErrorIdentifier="6" ErrorLevel="MODULE"/>
  </System_State_Entry>
  <System_State_Entry Description="Partition Initialization" SystemState="2">
    <Error_ID_Level Description="Power Interrupt" ErrorIdentifier="0" ErrorLevel="PARTITION"/>
    <Error_ID_Level Description="Illegal Instruction" ErrorIdentifier="1"
        ErrorLevel="PARTITION"/>
    <Error_ID_Level Description="Segmentation Error" ErrorIdentifier="2"
        ErrorLevel="PARTITION"/>
    <Error_ID_Level Description="Unimplemented Error" ErrorIdentifier="3"
        ErrorLevel="PARTITION"/>
    <Error_ID_Level Description="Floating Point Error" ErrorIdentifier="4"
        ErrorLevel="PARTITION"/>
    <Error_ID_Level Description="Overflow Error" ErrorIdentifier="5" ErrorLevel="PARTITION"/>
    <Error_ID_Level Description="Divide by zero" ErrorIdentifier="6" ErrorLevel="PARTITION"/>
  </System_State_Entry>
</System_HM_Table>

<Module_HM_Table>
  <System_State_Entry Description="PMK Execution" SystemState="1">
    <Error_ID_Action Action="SHUTDOWN" Description="Power Interrupt" ErrorIdentifier="0"/>
    <Error_ID_Action Action="SHUTDOWN" Description="Illegal Instruction" ErrorIdentifier="1"/>
    <Error_ID_Action Action="SHUTDOWN" Description="Segmentation Error" ErrorIdentifier="2"/>
    <Error_ID_Action Action="SHUTDOWN" Description="Unimplemented Error" ErrorIdentifier="3"/>
    <Error_ID_Action Action="SHUTDOWN" Description="Floating Point Error"
        ErrorIdentifier="4"/>
    <Error_ID_Action Action="SHUTDOWN" Description="Overflow Error" ErrorIdentifier="5"/>
    <Error_ID_Action Action="SHUTDOWN" Description="Divide by zero" ErrorIdentifier="6"/>
  </System_State_Entry>
</Module_HM_Table>

<Partition_HM_Table PartitionIdentifier="1" PartitionName="p0">
  <System_State_Entry Description="Partition Initialization" SystemState="2">
    <Error_ID_Action Action="IGNORE" Description="Power Interrupt" ErrorIdentifier="0"/>
    <Error_ID_Action Action="COLD_START" Description="Illegal Instruction"
        ErrorIdentifier="1"/>
    <Error_ID_Action Action="COLD_START" Description="Segmentation Error"
        ErrorIdentifier="2"/>
    <Error_ID_Action Action="COLD_START" Description="Unimplemented Error"
        ErrorIdentifier="3"/>
    <Error_ID_Action Action="IGNORE" Description="Floating Point Error" ErrorIdentifier="4"/>
    <Error_ID_Action Action="COLD_START" Description="Overflow Error" ErrorIdentifier="5"/>
    <Error_ID_Action Action="COLD_START" Description="Divide by zero" ErrorIdentifier="6"/>
  </System_State_Entry>
</Partition_HM_Table>
</ARINC_653_Module>
```

Figure B.2: HM configuration XML example.