uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática

2020 — 2021

Bachelor Thesis

DESIGN, IMPLEMENTATION AND VALIDATION OF A DEVELOPMENT FRAMEWORK FOR NANOSATELLITES BASED ON HYPERVISORS

Daniel Alcaide Nombela

Javier Fernández Muñoz

SUMMARY

In recent years, satellite missions have been increasing in a rapid manner. New technologies have permitted new approaches to space exploration never thought before, and have been used to cover new needs.

Among these new frontiers opened in recent years is the nanosatellite field. Standards like CubeSat, rich environments created by Space Agencies like ESA and Open Source Software projects like NASA's cFS and F' have all come together to allow a whole ecosystem to flourish around the developing and deployment of nanosatellites. These facilities have called the attention of engineering businesses like SENER.

This work is done in the context of the SENER/UC3M chair. The purpose of this work is to test and benchmark the viability of using the XtratuM hypervisor as a means of virtualizing nanosatellite applications in ARM platforms, as well as developing a framework that helps common tasks involved in this process.

To solve the problem proposed, a simplified V model plannification approach has been chosen. First, the user requirements, use cases and software requirements have been analysed. For this analysis, the modern state of the art has been taken into account, trying to base the solution in modern technologies like containers to design a clean workflow for user developers. Based on the analysis, a component design has been specified, later followed by a more detailed subcomponent specification. Benchmarks, a crucial part of this work, are also designed.

An implementation is given as a reference for the designed system, and taking into account the needs of the end users when choosing the technologies in which it is based. The implementation is exahustively described based on the component design.

To finish, the implementation of components is verified through thorough testing, and the system is validated against the initially analysed requirements. Also, the produced systems are tested using benchmarks that allow evaluating the adequacy of using the XtratuM hypervisor to isolate hard-real time systems like the needed in nanosatellites. Finally, a review of the planning, budget and legal and social implications is shown.

Keywords: Hypervisor, Nanosatellite, Flight Software, Framework, XtratuM

DEDICATION

I don't like dedications

but to all the people that kept me sane during this odyssey

and to my grandma for discovering her new ribs recipe just in time

and to Leslie Lamport for inventing LaTeX

and to the Free Software Foundation for giving me GNU awk

only God knows what could I have done without those things

and He gave me all of them, so to Him too

CONTENTS

1. INTRODUCTION 1
1.1. Objectives 1
1.1.1. Secondary objectives 2
1.2. Navigating this document 2
2. STATE OF THE ART 4
2.1. Space Exploration & Artificial Satellites 4
2.1.1. Small Satellites & Nanosatellites. 4
2.1.2. CubeSat 4
2.1.3. Flight Software 5
2.2. Hardware 7
2.2.1. Microprocessors & Instruction Set Architectures 7
2.2.2. Development Boards & Platforms 9
2.2.3. Comparative of Hardware Platforms 10
2.3. Software 10
2.3.1. Operating Systems 12
2.3.2. Development Frameworks 13
2.3.3. Hypervisors 15
2.4. Containers 18
2.4.1. Docker 19
2.4.2. OCI 19
2.5. Licensing and Free and Open Source Software 19
2.6. Command Line Interfaces 20
2.7. Build systems 21
3. ANALISIS 22
3.1. Description of the problem 22
3.2. Glossary 23
3.3. User requirements. 23
3.3.1. Capacity requirements 26
3.3.2. Restriction requirements 28

3.4. Use case design . . . 30
3.5. Software requirements . . . 34
3.5.1. Functional requirements . . . 35
3.5.2. Non-functional requirements . . . 41
3.5.3. Traceability matrices . . . 44
4. DESIGN . . . 48
4.1. Solution description . . . 48
4.2. Architecture design . . . 48
4.3. Components . . . 50
4.3.1. Static component diagram . . . 52
4.3.2. Component specification . . . 54
4.3.3. Subcomponent specification . . . 56
4.4. Benchmark design . . . 63
4.4.1. CPU stress time benchmarks (SBC-18) . . . 63
4.4.2. Memory stress time benchmarks (SBC-19) . . . 63
4.5. Traceability matrices . . . 64
5. IMPLEMENTATION . . . 67
5.1. Conventions . . . 67
5.2. CMP-01: Framework manager . . . 68
5.2.1. XtratuM build container image (SBC-06, SBC-01) . . . 68
5.2.2. `xmfw-install` script (SBC-02, SBC-03, SBC-04, SBC-05) . . . 69
5.3. CMP-02: System project manager . . . 70
5.3.1. System build images (SBC-11) . . . 70
5.3.2. `xmfw-system` (SBC-07, SBC-08) . . . 70
5.3.3. Default systems (SBC-16) . . . 71
5.3.4. Application management . . . 71
5.3.5. Systems deployment (SBC-09, SBC-10) . . . 72
5.3.6. Framework support . . . 72
5.4. CMP-03: Benchmarks . . . 74
5.5. CMP-04: XALC++ . . . 77
5.5.1. Memory manager (SBC-21) . . . 77
5.5.2. C++ support library (SBC-20) . . . 78

6. EVALUATION 81
6.1. Verification 81
6.1.1. Framework and system manager scripts verification 81
6.1.2. Benchmarks 102
6.1.3. XALC++ 102
6.2. Validation 102
6.2.1. Software requirements 103
6.2.2. Evaluation of the framework 104
7. PLAN AND BUDGET 107
7.1. Planning 107
7.2. Budget 107
7.2.1. Manpower 109
7.2.2. Hardware 109
7.2.3. Software 110
7.2.4. Others 110
7.2.5. Final budget 110
8. LEGAL FRAMEWORK AND SOCIAL IMPACT 112
8.1. Law 112
8.1.1. XtratuM licensing 113
8.2. Standards 113
8.3. Social and economic impact 114
9. CONCLUSIONS AND FUTURE WORK 115
9.1. Future work 115
9.2. Personal conclusions 116
BIBLIOGRAPHY 117
A. CPU BENCHMARK . . . . .
B. MEMORY BENCHMARK. . . . . .
C. CMAKE FILES FOR F' . . . . . .
D. DOCKERFILES . . . . . .
E. MEMORY MANAGEMENT . . . . . .

LIST OF FIGURES

2.1 Ncube2 CubeSat nanosatellite [3]. 5
2.2 Latest nanosatellite figures [4]. 6
2.3 Three popular development boards. 10
2.4 F' architecture. 14
2.5 cFS architecture. 15
2.6 Screenshots of Celestia in action [37]. 16
2.7 XtratuM architecture [41]. 17
2.8 XtratuM development process [42]. 18
2.9 Docker architecture [44]. 19

3.1 Use Case diagram of the framework. 30

4.1 Architectural view of the framework. 49
4.2 Framework manager installation and system creation. 51
4.3 Static component diagram. 53
4.4 CPU stress benchmarks application flow. 63
4.5 Memory stress benchmark application flow. 64

5.1 `hello-world` Makefile. 73
5.2 Sequence of commands needed to deploy XtratuM Resident Software in the Zedboard. 73
5.3 Semaphore system used to produce UART output in benchmarks. 75
5.4 Default scheduling of benchmarks. 76
5.5 Buddy memory allocator block entry data structure. 77
5.6 Menuconfig option to configure the memory manager allocation blocks. . 80

6.1 Results of the CPU benchmark (FPU mode). 106
6.2 Results of the memory benchmark. 106

7.1 Simplified V development model. 108
7.2 Gantt-like diagram of the planning adapted to the V development model. 108

LIST OF TABLES

2.1 Comparative between nanosatellite-development suited boards. 11
2.2 Comparison of popular Operating Systems. 12

3.1 Use Cases — User Requirements traceability matrix. 44
3.2 Software Requirements — Capacity User Requirements traceability matrix. 45
3.3 Software Requirements — Restriction User Requirements traceability matrix. 46
3.4 Software Requirements — Use Cases traceability matrix. 47

4.1 Components — Software requirements traceability matrix. 65
4.2 Subcomponents — Software requirements traceability matrix. 66

6.1 Values for test cases inputs. 83

1. INTRODUCTION

Space exploration has proven to be a profitable market full of oportunities. One of the companies helping in its realization is SENER. SENER group is a Spanish, Basque company dedicated to engineering and solutions development. One of their most important branches is SENER Aeronáutica, with which they have worked closely with institutions such as the ESA and the INTA in developing parts for space missions.

The Universidad Carlos III de Madrid has collaborated for many years with SENER, and from 2018 they both manage a joint chair for the research and development of space systems. The Computer Engineering department has as of recently been studying possible advances in nanosatellite software development technology, like the integration of frameworks like OpenSatKit with virtualized systems or the use of ARM technology for nanosatellites

This work tries to investigate the possibility of using hypervisors to build a framework on top of which reliable nanosatellite software systems can be built. This kind of software has real-time-like needs, so a rigurous process and reliable benchmarking will be essential to validate the work performed.

1.1. Objectives

The main purpose of this project is to aid future work in building reliable and complex systems. Through the development and integration of a system based in the free software XtratuM hypervisor, many tasks needed to perform further advances in this direction will be automated. Also, this work will try to advance attempts at providing support for developing applications based in other well-tested frameworks, like cFS and F', as SENER has requested.

Upon a fast review of the XtratuM hypervisor codebase and development environments it has been made apparent the need of some kind of abstraction that will separate the host development environment from the XtratuM development environment, due mainly to the outdated dependencies required to build this software and the very specific configuration restrictions that make its installation burdensome and inefficient.

The main objectives of this work are then as follows:

- Providing a framework for developing applications for nanosatellites based on the XtratuM hypervisor for ARM systems.
- Automating tasks common in the process of operating and developing applications for XtratuM.
- Abstract the development environment from the XtratuM build environment.

- Provide support for future work in the direction of advanced framework development for XtratuM.
- Explain some of the specific problems faced when using the XtratuM hypervisor.
- Provide benchmarking utilities to evaluate the validity of the XtratuM hypervisor in ARM systems.

1.1.1. Secondary objectives

Some secondary and more general objectives pursued by this work are:

- Providing a clean, portable and reproducible solution.
- Ensure that the solution is well integrated and coherent.
- Develop a simple, minimal but highly extensible solution.
- Provide well-written documentation that aids extensibility.

1.2. Navigating this document

This document consists on the following chapters:

1. **INTRODUCTION:** this is this chapter, where the problematic and main objectives are exposed.
2. **STATE OF THE ART:** a study in the technologies and solutions already developed and used that are in the interest of this work.
3. **ANALYSIS:** user and software requirements and use cases are analysed in this section.
4. **DESIGN:** in this section, the architecture and components design are specified, as well as the flows of the benchmarks later implemented.
5. **IMPLEMENTATION:** a detailed description of the reference implementation developed.
6. **EVALUATION:** verification of the system followed by its validation against the requirements analysed before, as well as the execution of the implemented benchmarks.
7. **PLAN AND BUDGET:** review of the process that took place to develop this work, as well as an estimation of its total cost.

8. **LEGAL FRAMEWORK AND SOCIAL IMPACT:** review of the law and standards affecting the product of this work, as well as some of its social implications.

9. **CONCLUSIONS AND FUTURE WORK:** the title of this section is self descriptive.

In the digital version of this work, and if the format and reader it is being displayed on support it, all references to sections, figures, tables, appendices, requirements, use cases and components as well as citations and URLs should be clickable links that ease navigation.

2. STATE OF THE ART

In this section we study the current software and hardware technologies, standards and market trends in the scope of space exploration, satellites, and, most specifically, software development for nanosatellites. We also discuss guidelines discovered throughout the study that may be useful in the development of the project.

2.1. Space Exploration & Artificial Satellites

Beginning with the space race, humankind has found the exploration of space very useful. From satellite telecommunications and scientific probing and measurement to defense applications, the space industry has been always full of opportunities.

Hundreds of satellite missions have been conducted since the first human satellite, the Sputnik I, was launched. However, the state of the technology and engineering has evolved exponentially since then. Lately, advances in technology have allowed small satellites to be launched and deployed in swarms at reduced costs, with the current trend being standardization and miniaturization.

2.1.1. Small Satellites & Nanosatellites

The project focuses on tools for the development of nanosatellites. Nanosatellites are a kind of small satellites — common classifications divide small satellites into five categories, most of them agreeing on using mass as the discrimination factor [1], [2]:

- **Minisatellites:** 500kg — 100kg
- **Microsatellites:** 100kg — 10kg
- **Nanosatellites:** 10kg — 1kg
- **Picosatellites:** 1kg — 100g
- **Femtosatellites:** $<$ 100g

Among these categories, nanosatellites have gained special attention as of lately, greatly thanks to the **CubeSat** specification.

2.1.2. CubeSat

CubeSat is a standard developed and maintained by California Polytechnic State University, San Luis Obispo and Stanford University's Space Systems Development Lab with the aim of easing the access to space exploration to students and academics [5]. The standard specifies dimensions, mass and other characteristics that satellites must comply with, allowing deployment in swarms and resulting in easier and cheaper launch, so that

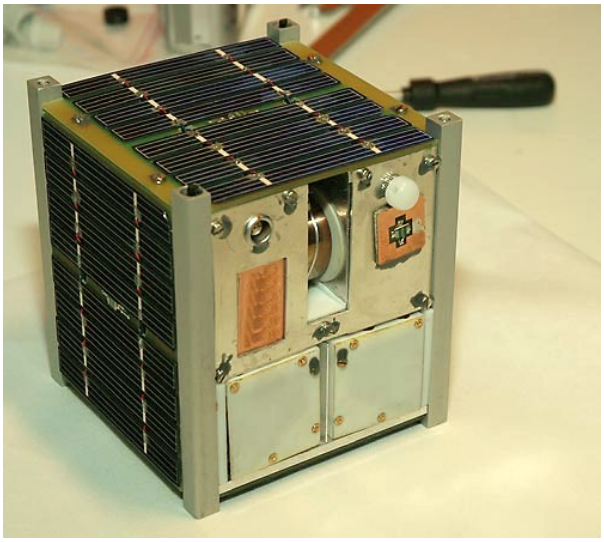


Fig. 2.1. Ncube2 CubeSat nanosatellite [3].

developers may focus on the development of functions of the satellite and not worry about launch-related requirements.

According to the standard, a satellite's mass and dimensions may be measured with *measurement units* (U). One measurement unit (1U) is 1.33kg maximum [6], so CubeSats can be considered a type of nanosatellite as of the classification above. Combined, CubeSats and other nanosatellites are a very important part of space exploration. Figure 2.2 shows the number and types of current nanosatellite missions.

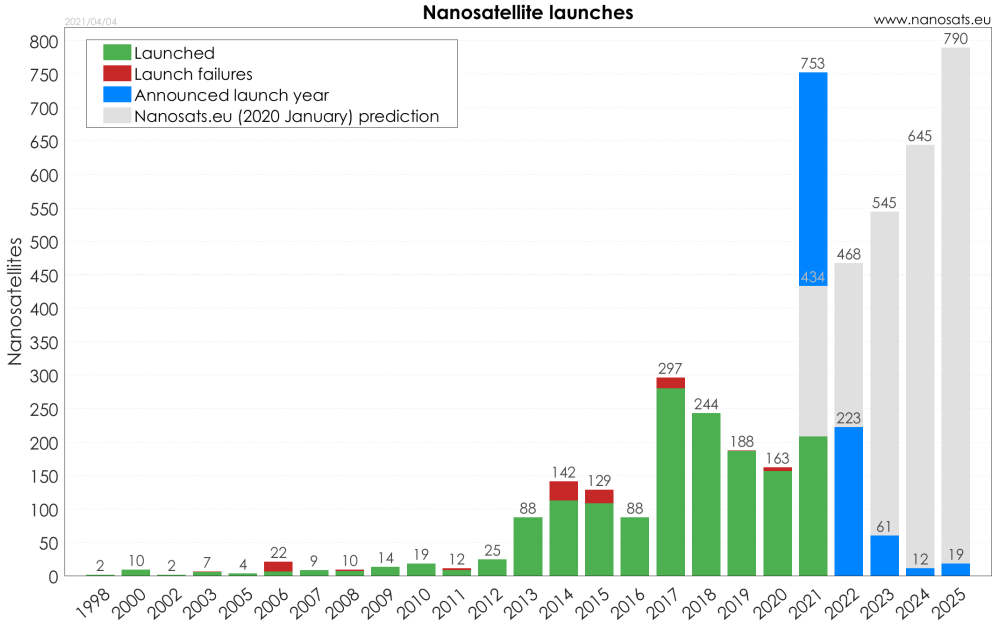
Despite its academic beginnings, CubeSats have gained the attention of many companies exploiting space exploration such as SENER, because of the business opportunities it enables. As we will work on the development of a software development framework, we will not need to address the CubeSat specification anymore — however, the environment this standard has created has unique characteristics that ought to be considered.

2.1.3. Flight Software

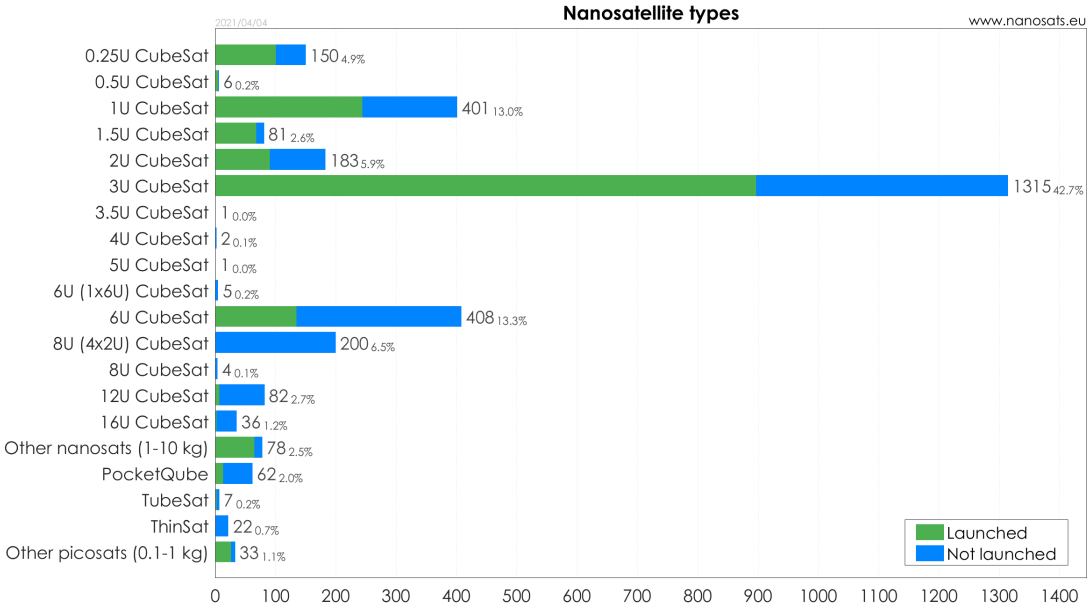
Production software in the context of satellite development is called the **flight software**, that is comprised of the instructions needed to perform all operations within the mission, which can range from science measuring and communication of data to maintenance tasks. It includes software running on the avionics modules and also subsystems and payloads, but this project will focuse on supporting development for payload software.

Complexity of flight software is based on the mission objectives, not spacecraft size, so its complexity in nanosatellites should not be underestimated. However, large computing power in such small spacecrafts has only been possible lately with new miniaturized, radiation hardened, low-power processors, such as ARM-based processors.

According to NASA's Small Spacecraft Institute, "[the flight software] must operate in a real-time environment". This means that avionics and other subsystems "need to be able to supervise several inputs and outputs as well as process and store data within a fixed



(a) Nanosatellite launches per year.



(b) Nanosatellite types.

Fig. 2.2. Latest nanosatellite figures [4].

time period". This "deterministic and reliable" nature of the software is a "fundamental requirement" [7].

Being aware of the specialties of computing in space, our work should support special architectures commonly used in space missions. In the development of our work we should also never underestimate the potential complexity of space applications. The real-time nature of applications developed for nanosatellites is another crucial aspect that must also be considered.

2.2. Hardware

Hardware in small spacecrafts has specific, tight-knit needs. Microprocessors must be small and low-power, and also be radiation hardened. This limits a lot the hardware possibilities when considering options while developing a satellite. In the case of nanosatellites, the reduced size makes hardware requirements very similar to those of embedded systems.

2.2.1. Microprocessors & Instruction Set Architectures

In this section we will study the different trends in microprocessors used in nanosatellite development and their Instruction Set Architectures. An Instruction Set Architecture (ISA) specifies what instructions can a microprocessor execute and how it will execute them. There exists a wide ecosystem of microprocessors and ISAs, and much more in fields where resources are scarce such as embedded computing, control systems and, what is the scope of this project, small satellites.

Some vendors use custom ISAs, while most use well defined, standard architectures (such as ARM). Some microprocessor hardware architectures are distributed as VHDL code that can then be used to program FPGAs (such as LEON). But, as discussed above, not all hardware can be used in space, so the range of widely used microprocessors and their architectures is relatively small. In this section we will discuss some of the most important market trends in microprocessors for nanosatellites.

SPARC & LEON

SPARC is a Instruction Set Architecture specification derived from RISC. It was initially developed by Sun Microsystems and UC Berkeley in 1984. According to its specification, SPARC was thought to be very flexible, allowing implementations "at a variety of price/performance points" and "for a range of applications, including scientific/engineering, programming, real-time, and commercial" [8], what makes it a good choice for the development of space-fit microprocessors. SPARC is supervised and maintained today by the non-profit SPARC International, Inc. They release the architecture under an all-purpose, royalty free proprietary license for $99 [9].

SPARC was the architecture chosen by the ESA for their LEON microprocessor. The LEON was developed by André Pouponnot, Jiri Gaisler and Richard Creasey in the ESA laboratories as an attempt of the agency on gaining control on the hardware used in their missions [10], [11]. The latest LEON microprocessor is the LEON5, which is currently maintained by the Cobham Gaisler company, and it includes radiation-hardened models and other space-suited platforms [12]. Its VHDL code is published under a dual license, GPL for its main components and commercial for advanced optimizations and proprietary integration [13].

We should consider in our work that, being used by the main processor dedicated for ESA missions, the SPARC architecture is very popular and demanded for nanosatellite flight software programming.

ARM & Xilinx Zynq

ARM is an IP company that provides a basic architecture upon which many custom microprocessor variants can be built. ARM describes their work as "a set of rules that dictate how the hardware works when a particular instruction is executed", "a contract between the hardware and the software, defining how they interact" with each other [14].

With such flexibility and support, the work of this company is becoming increasingly popular. Its platform has allowed the development of very energy-efficient processors, a reason why it is present in around 90% of mobile devices [15]. The development of ARM processors for space missions is becoming increasingly interesting for engineering companies, too, with advances being made in the development of radiation hardened ARM based hardware [16].

ARM architecture is divided in 3 families [17]:

- **Cortex A:** Suited for general application. Highly energy efficient and able to run complex software.
- **Cortex R:** Optimized for embedded, real-time critical applications, such as vehicle operation and embedded control systems.
- **Cortex M:** Designed for very low-energy systems, devised for the IoT.

ARM based devices are becoming very popular among embedded system development — and, according to NASA, "a majority of the small spacecraft developers use hardware typically employed in the embedded systems and control world". One example of this is the ARM-based Xilinx Zynq series, initially thought for embedded systems but being used in many integrated on-board computer solutions for nanosatellites offered in the market [18].

Xilinx Zynq is a series of System on Chip (SoC) hardware that integrates an ARM CPU with memory and all the things a general purpose computer needs to run generic software, including interfacing with ports and serial I/O. Xilinx provides a SDK with tools to communicate with the SoCs, as well as a custom build of the GCC toolchain with

the ARM ABI. They also provide tools to install and execute a Linux-based platform in the SoC, but this falls out of the scope of this project.

Our work should have present the current trends in the market, and the dominance and future potential of ARM based systems in the field of space exploration and, most importantly, nanosatellites.

IA32 & AMD64 processors

IA32 (also called x86 or i386) is a 32-bit architecture developed by Intel. AMD64 is the 64-bit successor of IA32 built by AMD, and later widely adopted by Intel. The latter is fully backwards-compatible with the former. Both are currently the most popular architectures among PCs and office workstations by far. It can be assumed that a good share of developers of nanosatellite applications will be used to IA32 or AMD64 systems, so it will be sensible to focus the work in supporting these architectures.

2.2.2. Development Boards & Platforms

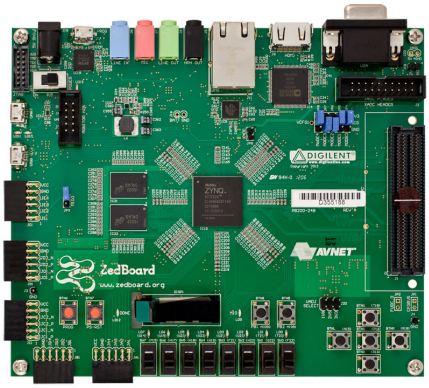
Compiled software does not only depend on the microprocessor and its ISA, but also in the board the chip is installed and the different components installed with it. This hardware bundle comes together with the Application Binary Interfaces and libraries needed to build software that can run on it. The ecosystem of hardware platforms that can be found being used successfully in small satellite missions is very diverse, with businesses offering well integrated solutions ready for production [18].

When developing software, testing it is a crucial and ubiquitous task. Integrated production-ready platforms may not be the best choice for this, for reasons like cost or the need of flexibility while in development. This is why *development boards* exist. They offer easy testing and prototyping of both hardware and software with the purpose of developing systems for some integrated platform.

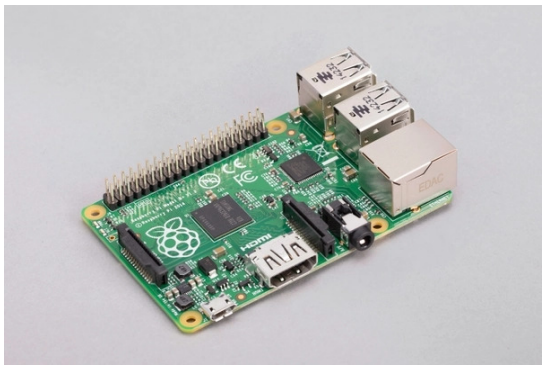
Following there are three popular development boards with targets common in the small satellite industry:

Zedboard Zedboard is a low-cost development board for the Xilinx Zynq-7000 System on Chip. This SoC is based on an ARM Cortex A9 processor. The board offers support to build and test complex OS/RTOS-based software for the Zynq-7000, as well as expansion connectors I/Os for easy user access and several integrated widgets like buttons, switches and displays.

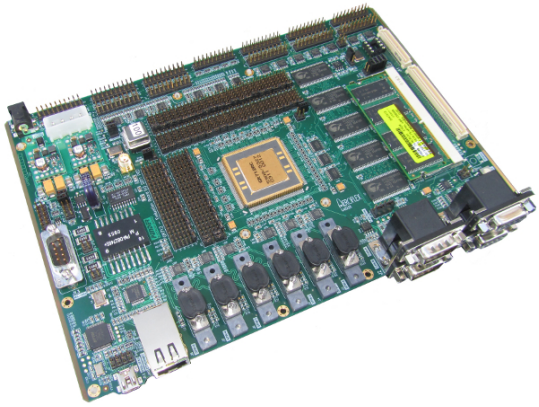
Raspberry Pi RPi is a general purpose hardware platform based on ARM. It is not a development platform, but its flexibility, simplicity and ease of use as well as its reduced cost have made it a popular option for both development and deployment of embedded systems.



(a) Zedboard [20].



(b) Raspberry Pi 1 Model B+ [21].



(c) GR712RC development board [19].

Fig. 2.3. Three popular development boards.

GR712RC Dev. Board Gaisler offers a development board for the GR712RC processor. This processor features a dual-core architecture based on the LEON3 processor and radiation hardening [19].

2.2.3. Comparative of Hardware Platforms

Table 2.1 compares the three development boards discussed above. In this project we will use the Zedboard as the main target for our framework. Reasons include reduced price, having the Zync7000 SoC (common in space-suited hardware integrations [18]) as target and explicit preferences given by SENER of targeting said SoC.

2.3. Software

Due to the restrictive nature of nanosatellite hardware, nanosatellite flight software is usually characterized by things very similar to those that affect embedded software that make it hard to implement complex and portable software platforms. These include:

- Cross compilation

ARM w/ Zedboard	• Xilinx Zynq ARM Cortex A9 • 512MB • Ethernet connection • Multiple displays • In-board buttons, leds and switches • Price: $450
ARM w/ Raspberry Pi	• Broadcom BCM2835, ARMv6 • 512MB • Wireless connectivity (WiFi, Bluetooth) • NASA's cFS well adapted [18] • Extensive SW and OS support (RTEMS) • Price: $50
SPARC w/ GR712RC	• LEON3 • Extensible memory banks • Ethernet connection • JTAG debug interface • In-board buttons, leds and switches • Price: not specified

Table 2.1. COMPARATIVE BETWEEN NANOSATELLITE-DEVELOPMENT SUITED BOARDS.

- Hardware platform specific Board Support Packages (BSP) and Application Binary Interfaces (ABI)
- Diversity of target hardware platforms
- Reduced resources
- Time restrictions

Among these, the most important for the scope of this project is arguably the need for cross compilation. Cross compilation is compiling code that runs in platform different from the machine doing the compilation. In this context, we should remind some commonly used expressions:

Platform Environment on which software is executed, comprised of the ISA, hardware, BSPs and Operating System libraries

Toolchain Set of programs, BSPs, ABIs, libraries and utilities that enable compilation of programs for a platform

Target Platform onto which the production software will be deployed

Host Platform that executes the compiler

In this section we will discuss several software technologies enabling the development of flight software that meets the needs of small satellite space missions.

2.3.1. Operating Systems

Operating systems are complex software frameworks that offer standard libraries and APIs, and in general an abstract machine useful for developing applications. They also implement services like task management, interruption handling and I/O interfacing and abstraction. They are part of what defines a platform.

	VxWorks	**RTEMS**	**FreeRTOS**	**Linux**
IA32	Yes	Yes	Yes	Yes
ARM	Yes	Yes	Yes	Yes
SPARC	Yes	Yes	No	Yes
Other	RISC-V, Power	18 architectures, +200 BSPs [7]	Limited hardware support [22]	—
Services	Wide range of services [23]	POSIX, TCP/IP	Very minimal	Full general purpose OS
Real Time	Yes	Yes	Yes	Limited
Strength	Featured development tools	Free license (freedom of porting)	Minimal and easy portable	Well tested for general purposes
Background	NASA used it for 20 years [7]	Several NASA & ESA missions [7]	CubeSats with limited resources [7]	NASA Ingenuity [24]
License	Proprietary	GPLv2, BSD [25]	MIT [26]	GPLv2 [27]

Table 2.2. COMPARISON OF POPULAR OPERATING SYSTEMS.

As discussed in Section 2.1.3, the software needed in nanosatellites can have complexity disproportionate to the size of the craft. For many applications, complex frameworks are needed, and very often fully fledged Operating Systems are used. An example of the need of this are real time requirements present in many flight software applications, that make so called *Real Time Operating Systems* very adequate for their development.

Real Time Operating Systems are a kind of operating systems that provide support for developing real time systems. Real time systems are those that require that tasks be executed in limited time frames in order to work correctly — very often, this is the case of satellite flight software. Software in these systems must be verifiable and deterministic in time. Real time systems can be classified in many ways:

According to controlled system properties

Hard real-time systems A single task exceeding its time frame makes the system fail fatally.

Soft real-time systems A single time frame violation does not bring the system down. Often, the system can recover from failures.

According to behaivor on failure

Fail safe systems A failure makes the system stop in a secure state.

Fail soft systems A failure does not make the system stop, but it keeps on with degraded services.

Due to the cost and fail-vulnerability of space missions, flight software of satellites usually can be classified as hard real-time, and ideally fail soft, though RTOSs usually offer support for every kind of RTS classification.

Operating systems must be adapted to the board they will be running on. This board-specific code is called Board Support Package. As discussed above, in space engineering there are many integrated solutions in the market — an OS should have a BSP specific to the integration it is to be ran. Some of the most popular Operating Systems used in spacecrafts are compared in Table 2.2.

2.3.2. Development Frameworks

Just like operating systems, frameworks are essential when building the complex software needed for space missions. There are many open source frameworks available for everyone to use, and fit and designed for flight software applications. In this section we discuss some of them.

F'

F' is a software framework developed at NASA's Jet Propulsion Laboratory. It features a modular component architecture, inter-component communication and auto-generation of code with model-based programming. According to its main web site, it provides "core capabilities like queues, threads and operating system abstraction", testing tools and a standard library with common APIs for flight software [28]. It also was made to allow making reusable code, to reduce time wasted in writing boilerplate code and to ease porting between operating systems and architectures [29].

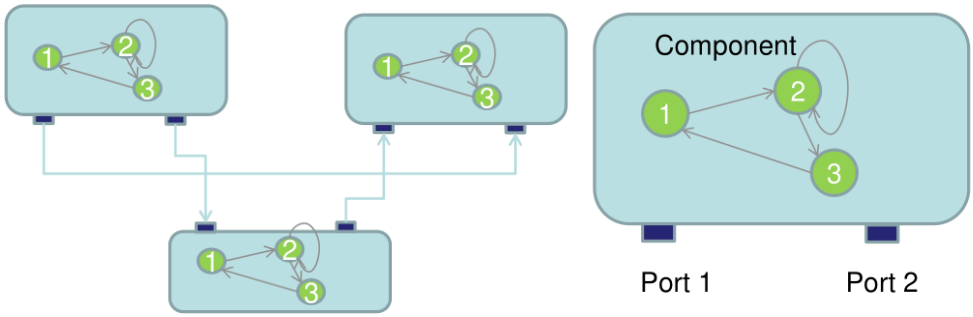


Fig. 2.4. F' architecture.

The framework uses C++ as its main language, and also XML model definitions to auto-generate code. The build system used is CMake, with integration scripts that make it straight-forward to build and generate framework components. Extensive documentation, install instructions and an example reference application are included. Its source code can be found on GitHub [30] under an Apache-2.0 license.

cFS & OpenSatKit

One of the most popular software frameworks for nanosatellites is NASA's core Flight System (cFS), due to its maturity, its wide range of services provided, its accessibility, its reusability and its default reference applications.

OpenSatKit is a desktop integration of cFS with 42 and COSMOS. It is thought for beginners in space application development, to let them learn easily how cFS works and how to make applications on top of it. It features an extensive wiki in its GitHub page [31].

cFS cFS is a bundle of platform independent framework and reusable applications. The framework provides a OS and hardware independent API, so that applications can be portable and reusable. Also, a set of common applications is provided to prevent the need of rewriting them [32]. Its main components are:

- **cFE:** Framework for applications. It provides a stable API and several services common for satellite applications.
- **OSAL:** Operating system abstraction layer.

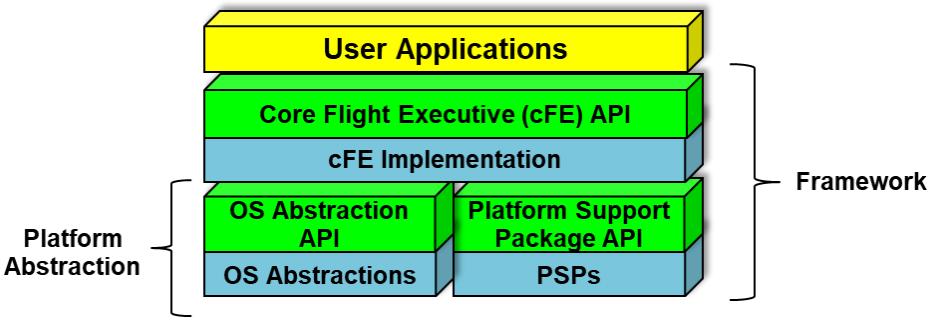


Fig. 2.5. cFS architecture.

- **PSP:** Platform Support Package, that abstracts the platform used.

The framework uses C as its main language to access its APIs. It also features a vibrant community, thanks to it being open source. Its source code is available on GitHub [33] under an Apache-2.0 license.

42 42 is an engine for simulating flight attitude and dynamics. It supports simulating physics from low orbit to a solar-system scale, and modelling the attitude of components in the spacecraft. According to the project, "its primary purpose is to support design and validation of attitude control systems, from concept studies through integration and test" [34]. OpenSatKit integrates it with cFS. Its source code is available on GitHub, under a custom, no-copyright license.

COSMOS COSMOS is an application suite developed by Ball Aerospace that allows controlling embedded systems. It features a client-server architecture, and can be used for ground control in satellite missions. Its source code can be found on GitHub [35], [36] under the AGPLv3 license.

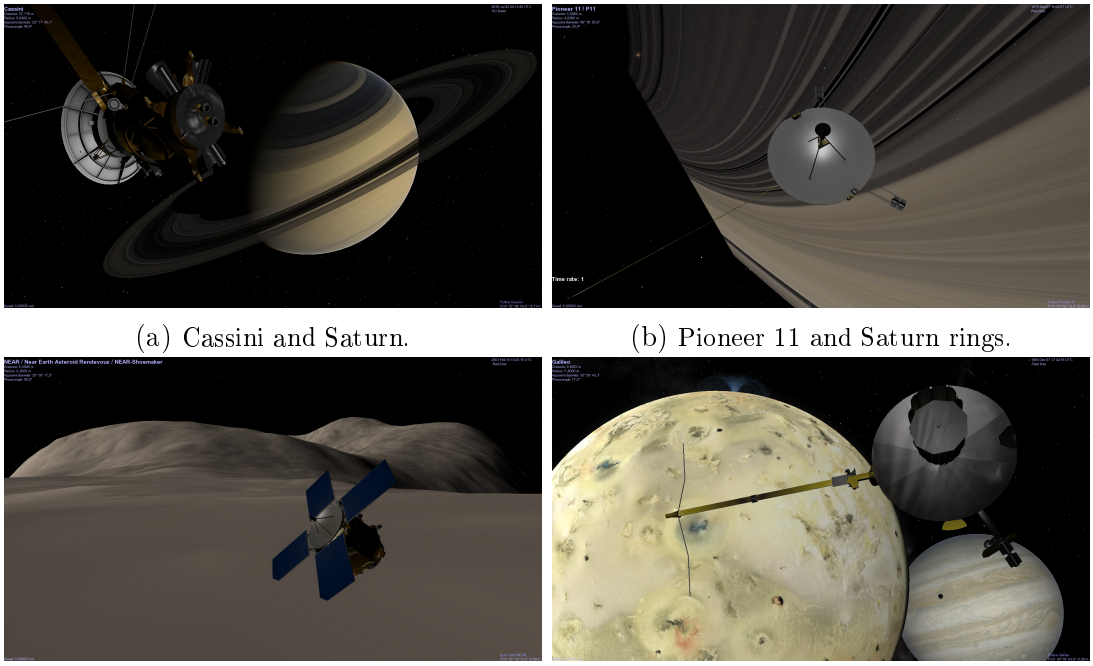
Celestia

Celestia is a real-time 3D simulation tool useful to display orbits and space bodies in an eye-captivating manner. Integration in space missions can be useful both for the control teams and the public that may be following them. Figure 2.6 shows some examples of Celestia in action. Its source code is available on GitHub [38] under the GPLv2 license.

2.3.3. Hypervisors

SENER is interested in the usage of **hypervisors** in nanosatellites. This would allow better hardware resource utilization for projects where several OS frameworks are needed. It also would make it easier to use multi-core systems, such as Zynq7000's dual-core ARM processor.

The hypervisor is a software layer that enables several operating systems or applications to share the same hardware while monitoring them. This partitioning and



(a) Cassini and Saturn. (b) Pioneer 11 and Saturn rings.

(c) Near Earth Asteroid Rendevouz. (d) Galileo on Jupiter.

Fig. 2.6. Screenshots of Celestia in action [37].

abstraction of the hardware is called **virtualization** [39]. The main mechanisms to virtualize systems are:

- **Native virtualization:** Partitions are executed directly on the host's hardware. Clock interruptions give control to the hypervisor, that monitors and manages partitions and their resource usage. Hardware support is needed for this mechanism, as virtualization instructions and execution levels are needed.
- **Paravirtualization:** Software that is to be executed on the hypervisor must be modified and adapted in a way specific to the hypervisor. Typically, trap instructions and system calls must be replaced by **hypercalls**, or calls to the hypervisor. This method does not require hardware support.
- **Emulation:** Software instructions are translated just in time (JIT) from the guest's architecture to the host's, then passed to the hypervisor, that executes them. It is the most versatile way of virtualizing, but also the most resource intensive and complex to implement.

Also, depending on what environment the hypervisor is executing it can be [39]:

- **Native:** The hypervisor is the last software layer in the system before the hardware/CPU, and it is the de-facto operating system. Guest operating systems run above the hypervisor.
- **Hosted:** The hypervisor runs atop a host operating system. Guests operating systems run then atop the hypervisor.

While hypervisors and virtualization are very common and well tested in many environments such as server hosting, applying them in flight software for satellites is not

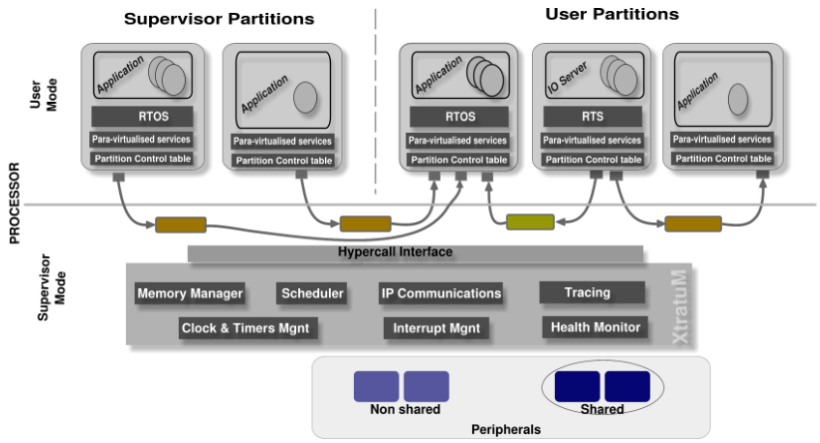


Fig. 2.7. XtratuM architecture [41].

easy, considering the real-time requirements they have.

XtratuM

XtratuM is a real-time capable hypervisor initially developed by the Universidad Politécnica de Valencia, and now maintained by fentISS. XtratuM's features are [40]:

- Native hypervisor
- Paravirtualized
- Strong temporal isolation, with a fixed cyclic scheduler
- Strong spatial istolation, with no shared memory
- Inter-Process Communication
- Health monitoring of partitions

The XtratuM hypervisor is comprised of the kernel (XtratuM core), user libraries for system calls (libxm) and the XAL environment extending libxm. It also includes the utilities necessary to parse configurations, compile partitions and build production software ready to execute, as well as scripts and utilities to install a minimal development environment based on GNU make.

XtratuM + XAL can be considered a platform different but equivalent to other operating systems, like RTEMS or Linux. However, its paravirtualization nature makes it difficult to use, as OSs and applications need to be adapted, and some functionality may not be yet implemented in the XtratuM core or the XAL. To allow porting of some C++ frameworks, like F', extending XAL with C++ support and memory allocation may be a good first step.

The XtratuM documentation includes guidelines as to how organize XtratuM-based systems development. These guidelines are shown in Figure 2.8. This can serve as inspiration in some phases of our work.

Despite XtratuM being licensed under the GPLv2, fentISS does not freely provide the source of XtratuM, not without requiring identification (registration) in their website.

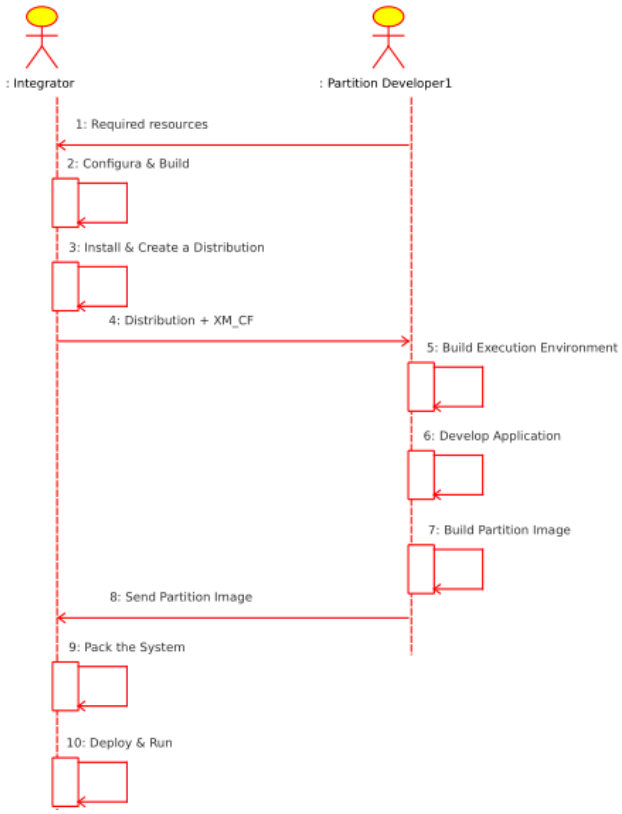


Fig. 2.8. XtratuM development process [42].

Moreover, ports to different architectures are only provided by fentISS through commercial contracts. However, we have access to XtratuM ported to ARM [41], SPARC [42] and IA32 [43]. Manuals for these XtratuM ports show that they have been tested to build in Ubuntu 14.04, so virtualization or containerization should be used in up-to-date systems.

2.4. Containers

Containers are a very popular method of isolating development and production environments while saving up on resource utilization. Containers are usually described as virtualization with just one OS: they effectively duplicate the operating system while keeping only one instance of the kernel running [44]. They can do this through the use of a service offered by the operating system called **namespaces**. A namespace is a logical view of a resource, like the root file system or the network stack. Processes can be assigned a namespace for a resource to make them have their own isolated view of said resource. Thus, containers are just processes isolated in different namespaces.

Containers are used in development when specific configurations are needed in the host machine to build and test the software under development. Many times it is not desirable to install such configurations directly in the host machine, for example when the development framework needs outdated software that it would be insecure to install in the main host. The use of containers may be considered when in need of isolate

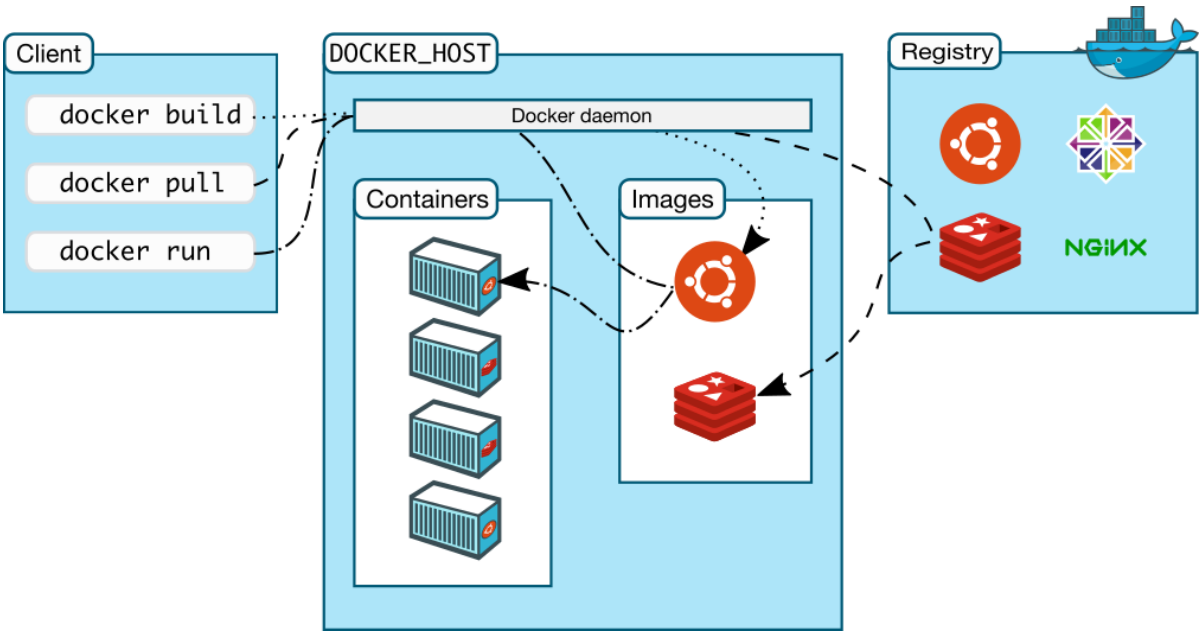


Fig. 2.9. Docker architecture [44].

configurations for development with frameworks.

2.4.1. Docker

One of the most popular platforms for containers is **Docker**. Docker provides an online repository for container **images**, Docker Hub. Images are a snapshot of the view of the file system of the container and enable, for example, emulation of any Linux distribution on a container hosted on any other distribution. Images can also have pre-installed all the software needed for development, or for the deployment of an application. The process needed for building images is specified in a **Dockerfile** [45].

A client can download or import images in its host, and then create containers through the docker daemon using an image. Also, images can be based on other images.

2.4.2. OCI

Docker was the reference implementation for what later became the Open Container Initiative. Lead by Docker, CoreOS and other leaders in the container industry, this committee was founded to oversee the standardization of container management systems. This standardization has given way to a whole ecosystem of container managers, like Podman [46] or Sysbox [47] — all of them are compatible between them and operate in similar ways thanks to the OCI standard.

2.5. Licensing and Free and Open Source Software

XtratuM is free software, licensed under the GPL license. In this section we introduce and explain the concepts and implications of licensing and what free software is.

Software in many countries is protected by copyright/author's rights. Authors may include licenses on their software, conditioning how, when and where can be executed and how the source code can be distributed. According to GitHub's licensing guide, "when you make a creative work (which includes code), the work is under exclusive copyright by default. Unless you include a license that specifies otherwise, nobody else can copy, distribute, or modify your work without being at risk of take-downs, shake-downs, or litigation" [48].

However, due to the nature of software, this can lead to unethical, freedom-coercing practices. The Free Software movement, backed by the Free Software Foundation, promotes what is called Free Software. The FSF provides 4 rules to know whether software is free (as in freedom):

- "The freedom to run the program as you wish, for any purpose (freedom 0)" [49].
- "The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this" [49].
- "The freedom to redistribute copies so you can help others (freedom 2)" [49].
- "The freedom to distribute copies of your modified versions to others (freedom 3). Access to the source code is a precondition for this" [49].

Many licenses can be considered *free* as by the description above. **Copyleft** licenses (word game for *opposed to* ***copyright***), like the GPL, are a special kind of license that enforces derivative works to maintain the original license, and so forces that the source code of the software is always available and proprietary closed-sourced versions cannot be legally produced.

Some of the most popular free software licenses are the **GNU General Public License (GPL)**, the **Berkeley Software Distribution (BSD)** license, the **Massachusetts Institute of Technology (MIT)** license and the **Apache** license. There are more, and each of them has its own conditions and details [50].

2.6. Command Line Interfaces

Despite of the popularity of the Graphical User Interfaces, Command Line Interfaces are widely used in certain sectors, among them systems development. The standard for the CLI is the POSIX shell, although there are extensions that are more popular, like the Bash shell. These programs provide programming language capabilities, and systems can be built using them. Recently, scripting languages like Python and JavaScript have also become popular platforms for implementing CLIs.

2.7. Build systems

A build system automates the compiling of source code and linking of binaries in large codebases. Build systems provide tools like version checking to avoid generating the same object twice, and rule declaration to define what must be done to produce each object. The standard has been for many decades the `make` utility introduced by UNIX systems, and later the GNU version GNU make.

More lately, CMake has been a popular frontend for build systems. CMake abstracts many build systems in many platforms to allow maximum portability of code and configurations. It generates instructions and rules for the underlying build system according to instructions defined by its own scripting language.

3. ANALISIS

In this chapter we will study the problem proposed, and analize requirements and use cases. First we will start discussing a detailed description of the problem and objectives, followed by a full specification of user and technical requirements, as well as use cases.

3.1. Description of the problem

The main objective of this work is to define a development framework for nanosatellite applications based on virtualization by a hypervisor. The purpose of this is to execute different platforms concurrently in the same hardware. However, some of this platforms may be Real Time Operating Systems (see 2.1.3), so the hypervisor will need to have features that preserve the real-time abilities of those RTOSs.

The quality of the produced software produced on top of the framework will be in great extent determined by the specification of the framework itself — therefore, the framework should be designed to allow maintainable, simple and modular applications. The focus of the framework must be providing mechanisms and configurations that are close to human intuition, and automate burdensome repetitive tasks that ballast application development, thus minimizing human effort of installing, producing and deploying software.

Nanosatellite Flight Software is sometimes hard-real time, and so it is critical that different applications and platforms be completely isolated from each other. If they are not, interferences might lead to fatal consequences. Thus, part of the framework must also include benchmarks for the evaluation and verification of the produced systems.

SENER, the company member of the chair that is context of this work, will be considered as our client. They have stated some preferences for this project:

- Using Xilinx Zynq7000 SoC as hardware platform
- Using XtratuM as hypervisor
- Allow cFS to run on top of XtratuM

These preferences are not conclusive, however, and may be considered against other options throughout the analysis and development of the project — for example, cFS support is outside the scope of this project. Also, we should consider desirable supporting hardware and software platforms discussed in the State of the Art study.

We will be using XtratuM in this project. However, as discussed in Section 2.3.3, the building of XtratuM works best on Ubuntu 14.04 with a specific environment of tools installed. Containers may be a good option to solve this problem.

It will be also convenient that the framework supports a minimal subset of the C++

standard, so that frameworks like F' can be compiled. For this, XtratuM will need to have implemented some minimal memory management.

3.2. Glossary

In this section we define some important terms used throughout the analysis, and further development of the project.

Application Part of a system that may be executed. It can be a bare application, an application on top of a framework, or a full OS-based platform. Applications have their own source code working tree, that is inside the working tree of a system.

Build Process of compiling code and linking libraries to generate an executable that can run on the target hardware.

Configure Specifying parameters needed to perform an action (generally building) in configuration files.

Deployment Executing a built executable in a hardware target or, by default, in a virtual environment that emulates the hardware.

Framework In this work, framework can mean two things:

- The object of this project, i.e., "a development **framework** for nanosatellites based on hypervisors"
- Already developed and well-tested application **frameworks** that can be the base of applications developed on top of the object of this project.

Hardware target Either a physical hardware platform or an emulated ISA.

Install Integrate a framework or tools with the development environment. It may imply copy files to required locations, building containers, compiling tools, etc.

System Bundle of one or more applications built on top of the framework object of this work. Altogether, the system is the complete software solution developed using the framework.

Working tree Filesystem tree containing the source code and configurations of a project (an application or a system).

3.3. User requirements

In this section user requirements are defined. User requirements are redacted taking into account the first approach to the problem, as well as the client's requirements. They are divided in two families:

- **Capacity requirements** specifying what must the solution be able to do
- **Restriction requirements** specifying under what conditions should the solution work, and how it should do it

Requirements must have some desirable characteristics essential for their validity. These characteristics are:

Correctness Requirements must represent accurately the needs of the system.

Completeness Requirements must cover all the needs of the system and define it to the greatest extent possible.

Unambiguity Requirements must be precise in their meaning, and have only one possible interpretation. They should also describe only one need of the system each.

Verifiability A verification of the requirements must be possible to perform.

Consistency Requirements must not contradict each other. They also should not specify the same thing twice.

Ranking Requirements must be to be ranked by importance and stability.

Taking into account these desired properties, user requirements will be defined using the following template:

UR-[CA / RS]-N	
Necessity: E | C | O	**Origin:** C | P
Priority: H | M | L	**Stability:** S | M | T
Description	

User requirements will be described using grammatic templates, i.e., all their descriptions will have a similar grammatic structure. Requirements descriptions should be only a simple statement long, and it is recommended that the subject always be as follows:

```
The <subject> must [have|offer|support|...] <object> [<conditions>]
```

where the `<subject>` can be the "framework" being described or any of its parts, and the `<condition>` is optional. The properties of user requirements will be:

ID Code : **UR-CA-N** if it is a capacity requirement or **UR-RS-N** if it is a restriction requirement. **N** is a number.

Necessity Importance of a requirement to achieve a solution. Values:

- Essential
- Convenient
- Optional

Origin Who came up with the requirement. Values:

- Client (SENER)
- Project (us)

Priority Priority during development. High priority requirements should be implemented before. Values:

- High
- Medium
- Low

Stability Whether a requirement could be changed in the future. Values:

- Stable
- May change

A requirement that is labeled as "May change" may be overlooked in future phases of the project. Usually, these are requirements that are also labeled as "Convenient" or "Optional", so that were them not possible to implement, the validity of the system will not be affected. These requirements will not be of Hight priority.

3.3.1. Capacity requirements

From here on follows a list of all the capacity user requirements.

UR-CA-01	
Necessity: Essential	**Origin:** Client
Priority: High	**Stability:** Stable
The framework must enable users to implement multiple nanosatellite applications that run sharing the same hardware	

UR-CA-03	
Necessity: Essential	**Origin:** Client
Priority: High	**Stability:** Stable
The framework must enable users to configure the CPU time utilization of the nanosatellite applications	

UR-CA-04	
Necessity: Essential	**Origin:** Client
Priority: High	**Stability:** Stable
The framework must enable users to configure the memory utilization of the nanosatellite applications	

UR-CA-05	
Necessity: Essential	**Origin:** Project
Priority: High	**Stability:** Stable
The framework must enable users to configure parameters of the framework	

UR-CA-06	
Necessity: Essential	**Origin:** Project
Priority: High	**Stability:** Stable
The framework must automate the framework installation	

UR-CA-07			
Necessity:	Essential	**Origin:**	Project
Priority:	High	**Stability:**	Stable
The framework must automate the creation of systems			

UR-CA-08			
Necessity:	Essential	**Origin:**	Project
Priority:	High	**Stability:**	Stable
The framework must automate the building of systems			

UR-CA-09			
Necessity:	Convenient	**Origin:**	Project
Priority:	Medium	**Stability:**	May change
The framework must provide means for the deployment of produced systems			

UR-CA-14			
Necessity:	Essential	**Origin:**	Client
Priority:	High	**Stability:**	Stable
The framework must support the development of hard-real time applications			

UR-CA-15			
Necessity:	Essential	**Origin:**	Project
Priority:	Medium	**Stability:**	Stable
The framework must offer a benchmark that allows validating its hard-real time properties			

UR-CA-18			
Necessity:	Convenient	**Origin:**	Project
Priority:	Medium	**Stability:**	Stable
The framework must support a minimal subset of the C++ standard			

3.3.2. Restriction requirements

From here on follows a list of all the restriction user requirements.

UR-RS-01			
Necessity:	Essential	**Origin:**	Client
Priority:	High	**Stability:**	Stable
The framework must be able to build systems for ARM based systems			

UR-RS-03			
Necessity:	Essential	**Origin:**	Project
Priority:	High	**Stability:**	Stable
The framework must be usable for development in up-to-date host systems			

UR-RS-06			
Necessity:	Optional	**Origin:**	Client
Priority:	Low	**Stability:**	May change
The framework must support development of nanosatellite software systems for SPARC architectures			

UR-RS-07			
Necessity:	Optional	**Origin:**	Client
Priority:	Low	**Stability:**	May change
The framework must support development of nanosatellite software systems for IA32 architectures			

UR-RS-09			
Necessity:	Convenient	**Origin:**	Project
Priority:	Medium	**Stability:**	Stable
The framework must be extensible			

UR-RS-10			
Necessity:	Convenient	**Origin:**	Project
Priority:	Medium	**Stability:**	Stable
The framework must be easy to use			

UR-RS-11			
Necessity:	Essential	**Origin:**	Client
Priority:	High	**Stability:**	Stable
The framework must isolate nanosatellite applications from each other			

UR-RS-13			
Necessity:	Convenient	**Origin:**	Project
Priority:	Medium	**Stability:**	Stable
The framework must minimize the outdated software installed in the host developer machine			

3.4. Use case design

The use case diagram in Figure 3.1 describes what is expected from the framework. The framework itself is an agent that should help the developer in the process of developing applications, and ultimately deploying them. Use cases are derived from the user requirements analyzed above, so for each use case we specify what user requirement it represents.

Following, the use cases are formally defined. It is assumed that the developer will interact with the framework through the execution of programs, without user interface or with a CLI at most.

As a remark, in some use cases, the framework performs actions on itself, such as building and installing. The framework will later be divided in components, some of which will not need to be built and installed to be used by the developer.

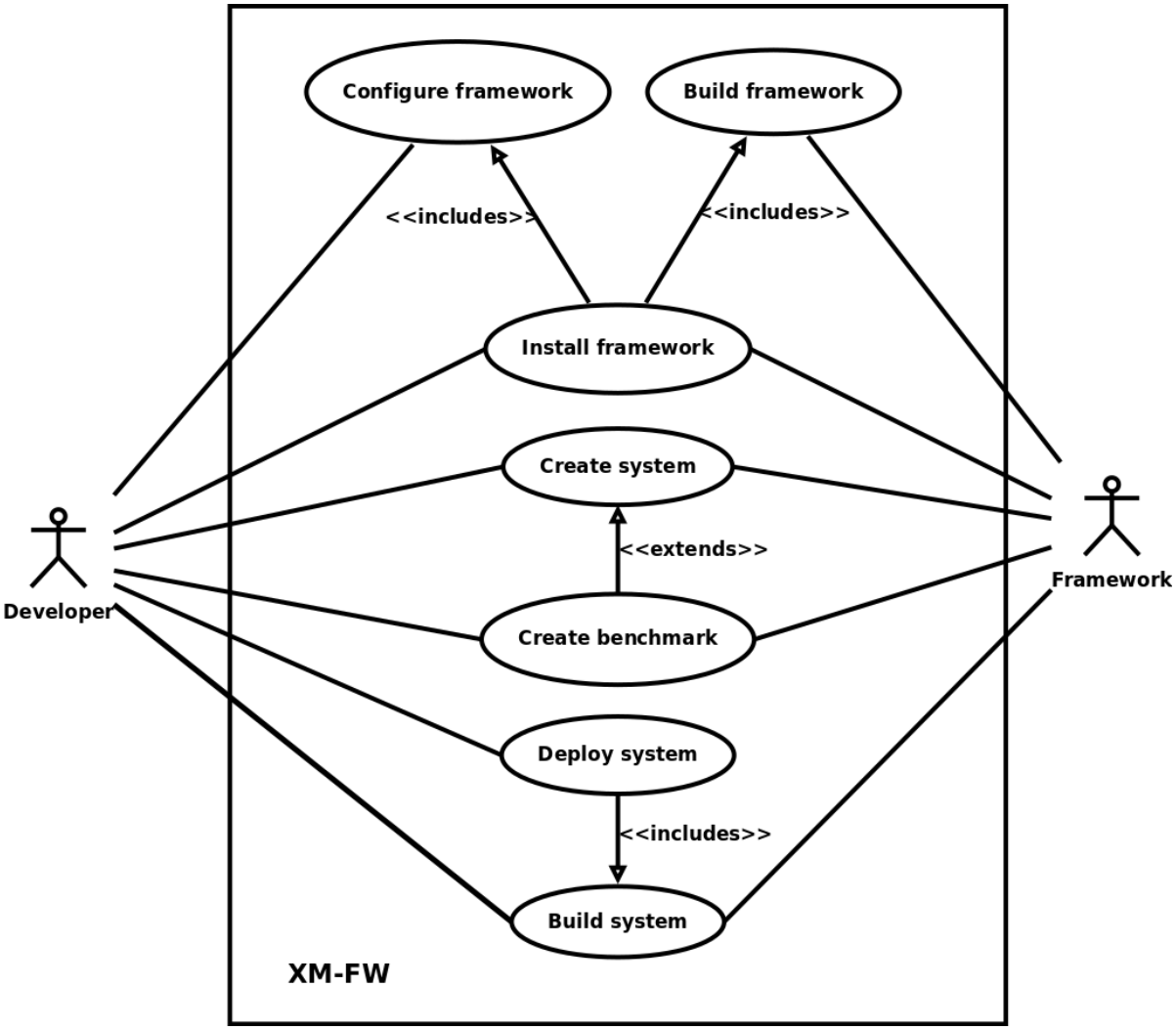


Fig. 3.1. Use Case diagram of the framework.

UC-01	
Title:	Configure framework
Agents:	Developer
Description:	Configuring parameters of framework components, such as the hypervisor
Flow:	1. Run configuration script 2. Choose parameters
Preconds:	None
Postconds:	Framework is configured and ready for installation
User reqs.:	[UR-CA-03] [UR-CA-04] [UR-CA-05]

UC-02	
Title:	Build framework
Agents:	Framework
Description:	Compile and build all binaries and libraries of the framework
Flow:	1. Framework builds the framework 2. Framework packages and readies the framework for installation
Preconds:	Configured framework [UC-01]
Postconds:	Built framework
User reqs.:	[UR-CA-06]

UC-03	
Title:	Install framework
Agents:	Developer, Framework
Description:	Installing the framework with specified parameters
Flow:	1. Developer instructs installation 2. Developer configures framework 3. Framework builds framework 4. Developer specifies installation parameters 5. Framework installs the framework according to specified parameters
Preconds:	None
Postconds:	Installed framework
User reqs.:	[UR-CA-06]

UC-04	
Title:	Create system project
Agents:	Developer, Framework
Description:	Create a work tree for a system of applications with default example configuration and applications
Flow:	1. Developer instructs system creation 2. Developer specifies Framework the name of the system and its creation path 3. Framework creates working tree 4. Framework sets up default configuration and files 5. Developer develops system
Preconds:	Installed framework [UC-03]
Postconds:	Created system
User reqs.:	[UR-CA-01] [UR-CA-07] [UR-CA-18]

UC-06	
Title:	Build system
Agents:	Developer, Framework
Description:	Build a system specifying its location
Flow:	1. Developer instructs system build 2. Build system 3. Package executable
Preconds:	Created system [UC-04]
Postconds:	Ready-to-deploy executable built
User reqs.:	[UR-CA-08] [UR-CA-18]

UC-07	
Title:	Deploy system
Agents:	Developer
Description:	Install built system into the hardware and prepare for execution
Flow:	1. Developer instructs deployment of system 2. The system starts executing in the target machine
Preconds:	Built system [UC-06]
Postconds:	System software is loaded into the hardware and ready to start execution
User reqs.:	[UR-CA-09]

UC-08	
Title:	Create benchmark
Agents:	Developer, Framework
Description:	Create benchmark tests for a specified target
Flow:	1. Developer specifies the benchmark to create 2. Framework creates the benchmark as a system 3. Developer deploys the system
Preconds:	Installed framework[UC-03]
Postconds:	System is deployed alongside with the benchmark added
User reqs.:	[UR-CA-14] [UR-CA-15]

3.5. Software requirements

In this section we analyze the software requirements of the framework, based on the analyzed user requirements and the use cases described above. Software requirements are specifications of user requirements that focus in technical details that the product must fulfill to cover them.

Software requirements must have the same desirable properties as user requirements. Because of this, the template used for their definition is similar to the one used for user requirements before, but changing the *Origin* field for *References* to the user requirements and use cases the software requirement implements. Requirements description will also be written as defined above. Software requirements are divided in two families:

- **Functional requirements requirements** specifying what functionality the software must implement.
- **Restriction requirements** specifying restrictions for the software and functionalities implemented.

Similarly to user requirements, a requirement that is labeled as "May change" may be overlooked in future phases of the project. Usually, these are requirements that are also labeled as "Convenient" or "Optional", so that were them not possible to implement, the validity of the system will not be affected. These requirements will not be of Hight priority.

3.5.1. Functional requirements

SR-FN-01		
Necessity: Convenient	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-18]		
Description: The framework must implement the `malloc` function as defined in the POSIX standard		
Verification: 1. `malloc` function is implemented 2. `malloc` function complies with the POSIX standard		

SR-FN-02		
Necessity: Convenient	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-18]		
Description: The framework must implement the `free` function as defined in the POSIX standard		
Verification: 1. `free` function is implemented 2. `free` function complies with the POSIX standard		

SR-FN-03		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-05] [UC-01] [SR-NF-01]		
Description: The framework must include a XtratuM configurer that allows the user to configure XtratuM		
Verification: 1. The XtratuM configurer configures XtratuM according to specified parameters		

SR-FN-04		
Necessity: Convenient	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-05] [UR-RS-09] [UR-RS-10] [UC-01]		
Description: The XtratuM configurer must provide means of exporting XtratuM configuration in a `.tar.gz` archive		
Verification: 1. The XtratuM configurer packs XtratuM configuration into a `.tar.gz` 2. The archive is accessible to the developer		

SR-FN-05		
Necessity: Convenient	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-05] [UR-RS-09] [UR-RS-10] [UC-01]		
Description: The XtratuM configurer must provide means of importing XtratuM configuration from a `.tar.gz` archive		
Verification: 1. The XtratuM configurer configures XtratuM according to previously exported configuration		

SR-FN-06		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-06] [UC-03]		
Description: The framework must install XtratuM and have it ready to build systems based on it		
Verification: 1. The framework builds XtratuM (see [SR-FN-07]) 2. If the build was successful, XtratuM is installed		

SR-FN-07		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-06] [UC-02]		
Description: The framework must build XtratuM before installation		
Verification: 1. Building is done before installation 2. If XtratuM is not configured, the XtratuM configurer is executed 3. The framework builds XtratuM 4. Upon error, the framework reports and aborts		

SR-FN-08		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-07] [UR-RS-10] [UC-04]		
Description: The framework must be able of creating a system given its name, as per user instruction		
Verification: 1. The framework creates a dummy system with default configuration 2. The created system can be built		

SR-FN-09		
Necessity: Convenient	**Priority:** Medium	**Stability:** May change
References: [UR-CA-08] [UR-RS-10] [UC-06]		
Description: The framework must be able of building a system given its name		
Verification: 1. The framework builds a system 2. The built system can be successfully executed in the hardware target		

SR-FN-13		
Necessity: Convenient	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-01] [UR-CA-06] [UR-CA-07] [UR-RS-09]		
Description: The framework must provide means of choosing the hardware target for a system		
Verification: 1. The hardware target can be chosen 2. Built applications are fit for the hardware target chosen		

SR-FN-17		
Necessity: Convenient	**Priority:** Medium	**Stability:** May change
References: [UR-CA-09] [UR-RS-01] [UC-07]		
Description: The framework must provide means of deploying a system in a Zynq7000 SoC connected via means compatible with the Xilinx SDK		
Verification: 1. The framework builds a system 2. If there was an error, the framework aborts and reports 3. Else, the built system can be loaded on the Zynq7000 SoC using the Xilinx SDK following instructions of the framework		

SR-FN-18		
Necessity: Optional	**Priority:** Low	**Stability:** May change
References: [UR-CA-09] [UR-RS-06] [UC-07]		
Description: The framework must provide means of deploying a system in a IA32 virtual machine		
Verification: 1. The framework builds the system 2. If there was an error, the framework aborts and reports 3. Else, the built system can be ran on a IA32 virtual machine following instructions of the framework		

SR-FN-20		
Necessity: Convenient	**Priority:** Medium	**Stability:** May change
References: [UR-CA-15] [UR-RS-09] [UC-08]		
Description: The framework must provide means of creating benchmark systems		
Verification: 1. The framework creates a system that is the specified benchmark 2. The system is built and deployed successfully with the specified benchmark		

SR-FN-21		
Necessity: Essential	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-15] [UC-08]		
Description: The framework must include a time stress isolation benchmark that overloads the CPU and checks for time frame violations		
Verification: 1. The benchmark overloads the CPU 2. The benchmark allows verification of time frame violations		

SR-FN-22		
Necessity: Essential	**Priority:** Medium	**Stability:** Stable
References: [UR-CA-15] [UC-08]		
Description: The framework must include a stress isolation benchmark that overloads the memory and checks for time frame violations		
Verification: 1. The benchmark overloads the memory access 2. The benchmark allows verification of time frame violations		

SR-FN-27		
Necessity: Convenient	**Priority:** Low	**Stability:** May change
References: [UR-CA-18]		
Description: The framework must provide a definition for the `new` operator		
Verification: 1. The `new` operator works as defined in the C++ standard		

SR-FN-28		
Necessity: Convenient	**Priority:** Low	**Stability:** May change
References: [UR-CA-18]		
Description: The framework must provide a definition for the `delete` operator		
Verification: 1. The `delete` operator works as defined in the C++ standard		

3.5.2. Non-functional requirements

SR-NF-01		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-01] [UR-CA-03] [UR-CA-04] [UR-CA-14] [UR-RS-11]		
Description: The framework must use XtratuM hypervisor		
Verification: 1. XtratuM will be used to isolate applications and virtualize them in the same hardware		

SR-NF-08		
Necessity: Convenient	**Priority:** Low	**Stability:** May change
References: [UR-CA-18]		
Description: The framework must include standard C++ header files		
Verification: 1. F' Ref builds without undeclared symbols		

SR-NF-09		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-06] [UR-RS-03] [UR-RS-13] [UC-02]		
Description: The framework must build XtratuM in a container		
Verification: 1. The framework makes use of containers when building XtratuM 2. The host system environment remains untouched		

SR-NF-10		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-08] [UR-CA-09] [UR-RS-01] [UC-06]		
Description: The framework must be able to build systems for the Zynq7000 SoC		
Verification: 1. A component of the framework, when executed, builds an executable ready to be loaded on the Zynq7000 SoC		

SR-NF-11		
Necessity: Optional	**Priority:** Low	**Stability:** May change
References: [UR-CA-08] [UR-CA-09] [UR-RS-06] [UC-06]		
Description: The framework must be able to build systems for the SPARC		
Verification: 1. A component of the framework, when executed, builds an executable ready to be loaded on a SPARC virtual machine		

SR-NF-12		
Necessity: Optional	**Priority:** Medium	**Stability:** May change
References: [UR-CA-08] [UR-CA-09] [UR-RS-07] [UC-06]		
Description: The framework must be able to build systems for the IA32 architecture		
Verification: 1. A component of the framework, when executed, builds an executable ready to be loaded on a IA32 architecture virtual machine		

SR-NF-13		
Necessity: Essential	**Priority:** High	**Stability:** Stable
References: [UR-CA-14] [UR-CA-15] [UC-08]		
Description: The applications developed on the framework must comply with their configured time plans		
Verification: 1. Benchmarks included in the framework verify that produced systems comply with configured time plans		

3.5.3. Traceability matrices

In this section we discuss the results of the traceability matrices of the analysis. These matrices will allow to verify the completeness and consistency of the requirements.

Table 3.1 shows the traceability of use cases against capability user requirements. Restriction user requirements, due to their nature, cannot be labeled as the source of any use case, as use cases define actions performed by actors that are enabled by the capabilities of the system. As it can be seen, capability user requirements are all covered by at least one use case, and all use cases are related to at least one user requirement.

Tables 3.2 and 3.3 show the traceability of software requirements against capability and restriction user requirements respectively. All user requirements are covered by software requirements, and as it is natural, the same user requirement is implemented by many software requirements.

Table 3.4 shows the traceability of software requirements against use cases. This is not needed to verify the integrity of the requirements, but it is included for completion and future reference.

UR-CA-	01	03	04	05	06	07	08	09	14	15	18
UC-01		•	•	•							
UC-02					•						
UC-03					•						
UC-04	•					•					•
UC-06							•				•
UC-07								•			
UC-08									•	•	

Table 3.1. USE CASES — USER REQUIREMENTS TRACEABILITY MATRIX.

UR-CA-	01	03	04	05	06	07	08	09	14	15	18
SR-FN-01											●
SR-FN-02											●
SR-FN-03				●							
SR-FN-04				●							
SR-FN-05				●							
SR-FN-06					●						
SR-FN-07					●						
SR-FN-08						●					
SR-FN-09							●				
SR-FN-13	●				●	●					
SR-FN-17								●			
SR-FN-18								●			
SR-FN-20										●	
SR-FN-21										●	
SR-FN-22										●	
SR-FN-27											●
SR-FN-28											●
SR-NF-01	●	●	●						●		
SR-NF-08											●
SR-NF-09					●						
SR-NF-10							●	●			
SR-NF-11							●	●			
SR-NF-12							●	●			
SR-NF-13									●	●	

Table 3.2. SOFTWARE REQUIREMENTS — CAPACITY USER REQUIREMENTS TRACEABILITY MATRIX.

UR-RS-	01	03	06	07	09	10	11	13
SR-FN-01								
SR-FN-02								
SR-FN-03								
SR-FN-04					•	•		
SR-FN-05					•	•		
SR-FN-06								
SR-FN-07								
SR-FN-08						•		
SR-FN-09						•		
SR-FN-13					•			
SR-FN-17	•							
SR-FN-18			•					
SR-FN-20					•			
SR-FN-21								
SR-FN-22								
SR-FN-27								
SR-FN-28								
SR-NF-01							•	
SR-NF-08								
SR-NF-09		•						•
SR-NF-10	•							
SR-NF-11			•					
SR-NF-12				•				
SR-NF-13								

Table 3.3. SOFTWARE REQUIREMENTS — RESTRICTION USER REQUIREMENTS TRACEABILITY MATRIX.

UC-	01	02	03	04	06	07	08
SR-FN-01							
SR-FN-02							
SR-FN-03	•						
SR-FN-04	•						
SR-FN-05	•						
SR-FN-06			•				
SR-FN-07		•					
SR-FN-08				•			
SR-FN-09					•		
SR-FN-13							
SR-FN-17						•	
SR-FN-18						•	
SR-FN-20							•
SR-FN-21							•
SR-FN-22							•
SR-FN-27							
SR-FN-28							
SR-NF-01							
SR-NF-08							
SR-NF-09		•					
SR-NF-10					•		
SR-NF-11					•		
SR-NF-12					•		
SR-NF-13							•

Table 3.4. SOFTWARE REQUIREMENTS — USE CASES TRACEABILITY MATRIX.

4. DESIGN

In this chapter, a solution will be given for the problem analysied in the previous chapter. First, we describe what will the solution consist on. Then, we explain the architecture of the solution and specify its components.

4.1. Solution description

In this section we describe informally the design that will be described and specified thoroughly in next sections. Taking into account all requirements and use cases, the solution will be divided into four high-level components:

Framework manager Builds, configures and installs XtratuM versions.

System project manager Creates, builds and deploys systems and applications

Benchmarks Bundle of default benchmarks

XALC++ Support library for a minimal subset of C++

In previous chapters it has been made evident the need for containers to isolate building environments. Due to XtratuM needing an outdated environment to build, this is the most secure approach — also, this way the user can have several XtratuM versions each for a different hardware target, all managed by the framework. Containers will also be used to build systems based on XtratuM — this gives maximum extensibility and adaptability for supporting frameworks, while maintaining reproducibility over time.

The framework will consist largely in a set of scripts and utilities that automate creation, configuration, building and deployment of systems. It also will include default system configurations and benchmarking systems that allow an easy verification of the framework for hardware targets

The framework will also include support libraries such as XALC++, that is an extension of XAL (included in XtratuM) that must offer basic memory management utilities and successful compilation of C++ applications.

The framework will not offer solutions for unit testing and version control. This is considered to be out of the scope of the work, but the framework will be designed with extensibility in mind also in these aspects, so that it would be easy to adapt in the future.

4.2. Architecture design

As explained above, the framework consists in four high-level architectural components. These components can be seen in Figure 4.1. The *Framework manager* will manage

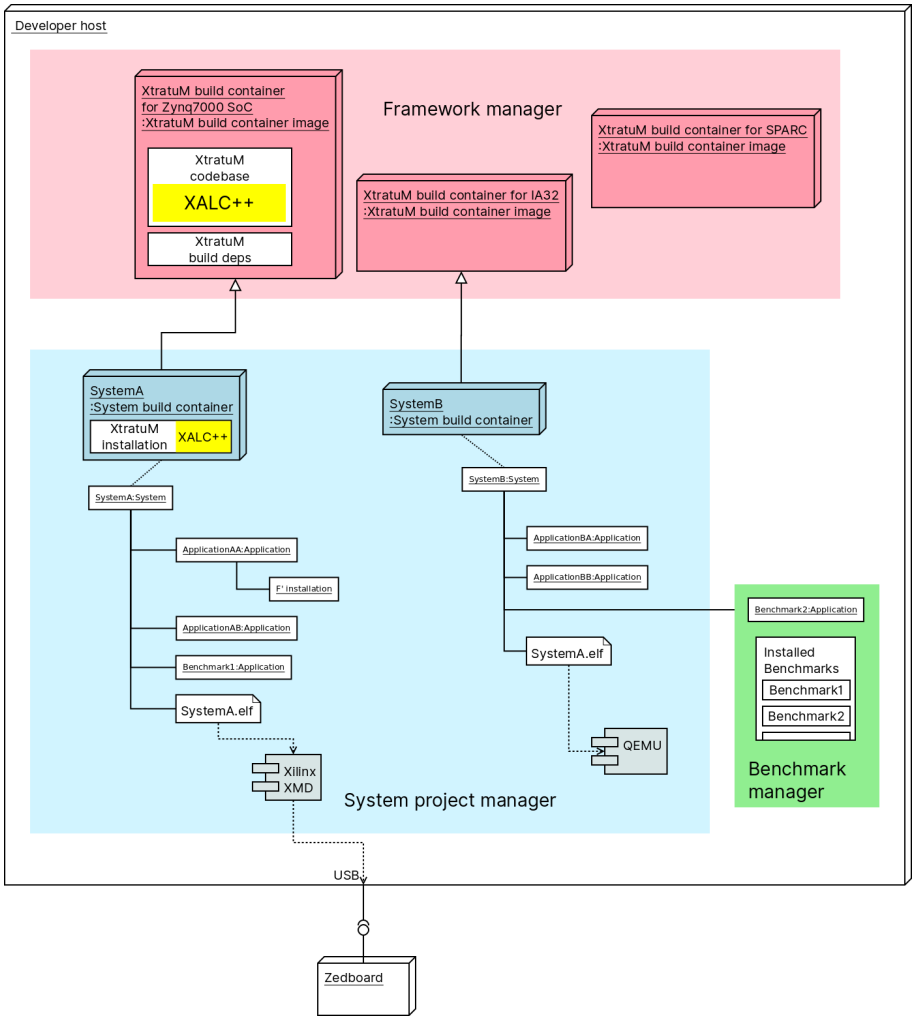


Fig. 4.1. Architectural view of the framework.

containers made to build XtratuM — it will also provide utilities for configuring and installing it.

Installing the framework is done by creating a container image that includes XtratuM installed and is capable of building systems. Containers based on this image, as well as system and application working trees are managed by the *System project manager*. This level is also in charge of deploying systems depending on the hardware target. A utility will also be provided by this component to install frameworks in applications.

Framework manager Figure 4.2 shows the flow of the actions that the framework manager does upon installation. First, the *System base image* and the *XtratuM Framerwork base image* are provided. Based on the *XtratuM Framerwork base image*, a *XtratuM Framework source (or build) image* is created including the XtratuM codebase and all the dependencies needed to build it. Upon installation, the framework manager will create a *System build image* upon which *System build containers* will be based.

The *XtratuM Framework base image* should be version and architecture independent, while *XtratuM Framework build images* must include all specific architecture default configuration and systems. The *System build image* may not be based on any of the

XtratuM framework's images, but it should support building XtratuM systems, so it is recommended that it is based on at least the *XtratuM Framework base image*.

System project manager When a system is created, a container is created with is. This container will build the system when executed. For this, the working tree of the system and the toolchain needed to build it will be mounted as volumes in the container. The produced executable will be in placed in the working tree of the system for the developer to use with the deployer components. Figure 4.2 shows this process. This is performed by the system project manager.

Benchmarks The framework will allow creating benchmark systems based on predefined benchmarks. There will be an internal database of systems managed by the XtratuM build images for each architecture. When creating a system, the developer will be able to chose among the example systems as templates for the new one.

XALC++ The source code of this library will be included with XtratuM in the XtratuM build image, and will be compiled and packaged fully integrated with the hypervisor installation. In fact, images supporting XALC++ should be shipped in their own XtratuM build images as an extension of a specific version and architecture.

4.3. Components

To design the application, in this section we divide the solution into components. These components are the four high-level architectural components described earlier, and will be divided into several subcomponents in order to further specify their functions and to ease later verification and maintainability.

Each subcomponent will depend on others and will implement some requirements defined in the analysis. For each component we define the following attributes:

Type Can be one of the following (type names are self descriptive):

- Application
- Benchmark bundle
- C library
- Container image
- Function
- Script/Command
- Script/Utility
- System

Parent (Optional) Component parent of a subcomponent

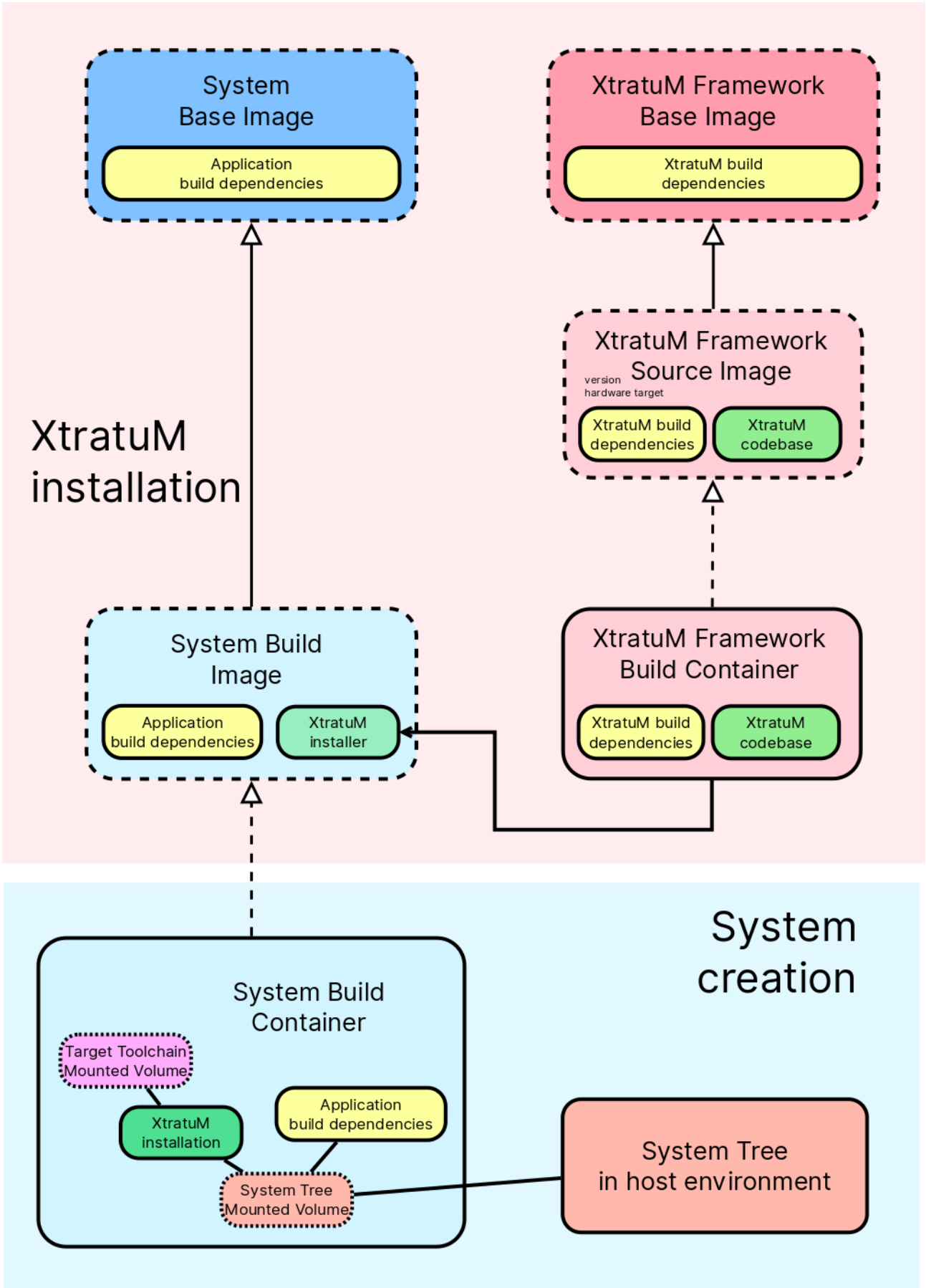


Fig. 4.2. Framework manager installation and system creation.

Attributes (Optional) Data used by the component, or informal properties it must have

Functionality (Optional) Can be either: series of steps the component must perform, or: list of functionalities the component must implement

Depends on External and internal dependencies of the component — these are represented in Figure 4.3

References Requirements and use cases for traceability

4.3.1. Static component diagram

Among the desirable properties of the system is the **decoupling** and **cohesion** of components. Coupling is defined as "the degree of interdependence between components" [51], and cohesion as the property that is fulfilled if the "degree to which components belong together" [52].

Components must ideally be decoupled, i.e., not depend on other components, and they must also have high cohesion, i.e, that functionality they implement is similar uniquely implemented by them.

Figure 4.3 shows the static component diagram of the system. The diagram has been drawn using the UML standard plus a color language explained in the legend. High-level components are represented as packages, while subcomponents are represented as UML components.

The static component diagram helps analyze if the components in the design follow the desired properties explained above. As it can be seen, not depending any component on any other, high-level components are highly decoupled — also, their semantic cohesion is evidently very high, with each component implementing very isolated functionalities. Cohesion will be later analyze using traceability matrices in Section 4.5.

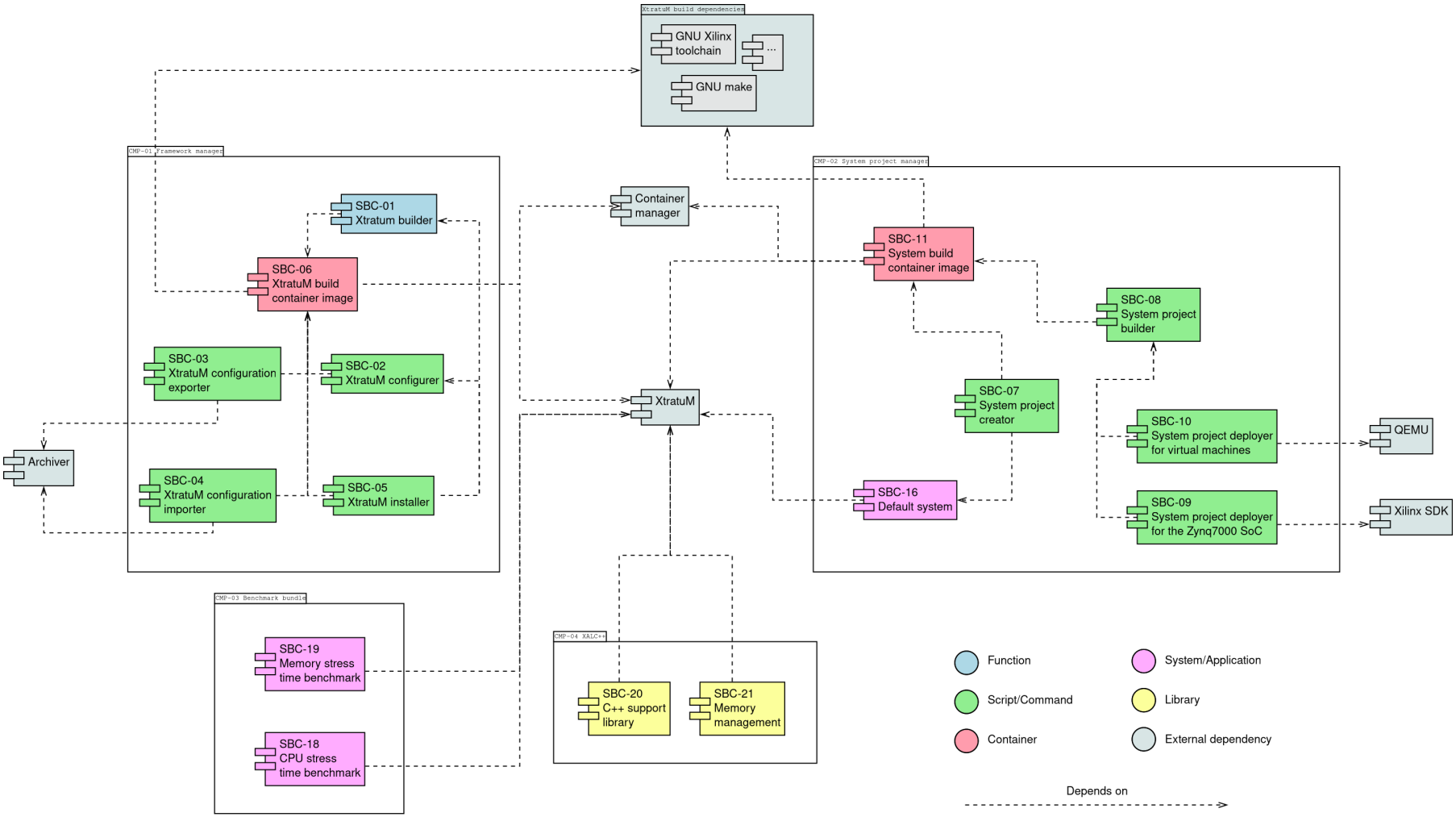


Fig. 4.3. Static component diagram.

4.3.2. Component specification

In this section, high-level components are specified.

CMP-01	
Name:	Framework manager
Type:	Script/Utility
Functionality:	1. Build XtratuM 2. Install XtratuM 3. Configure XtratuM 4. Import XtratuM configuration 5. Export XtratuM configuration
Depends on:	Container manager, XtratuM codebase, XtratuM build dependencies
References:	[SR-FN-03] [SR-FN-04] [SR-FN-05] [SR-FN-06] [SR-FN-07] [SR-NF-01] [SR-NF-09] [UC-01] [UC-02] [UC-03]

CMP-02	
Name:	System project manager
Type:	Script/Utility
Functionality:	1. Create systems 2. Build systems 3. Deploy systems 4. Create applications 5. Add frameworks to applications
Depends on:	Container manager, XtratuM, Xilinx SDK
References:	[SR-FN-08] [SR-FN-09] [SR-FN-13] [SR-NF-10] [SR-NF-11] [SR-NF-12] [SR-FN-17] [SR-FN-18] [SR-FN-20] [UC-06] [UC-07]

CMP-03	
Name:	Benchmarks
Type:	Benchmark bundle
Functionality:	1. CPU stress time benchmark application 2. Memory stress time benchmark application
Depends on:	XtratuM
References:	[SR-FN-21] [SR-FN-22] [SR-NF-13] [UC-08]

CMP-04	
Name:	XALC++
Type:	C library
Functionality:	1. Memory allocation 2. Basic C++ support
Depends on:	XtratuM
References:	[SR-FN-01] [SR-FN-02] [SR-FN-27] [SR-FN-28] [SR-NF-08]

4.3.3. Subcomponent specification

In this section, subcomponents are specified.

SBC-01	
Name:	XtratuM builder
Type:	Function
Parent:	[CMP-01]
Properties:	• XtratuM codebase • XtratuM configuration • Name of the container containing the former
Functionality:	1. Build XtratuM in the XtratuM build container
Depends on:	[SBC-06]
References:	[SR-FN-07] [SR-NF-01] [SR-NF-09] [UC-02]

SBC-02	
Name:	XtratuM configurer
Type:	Script/Command
Parent:	[CMP-02]
Properties:	• XtratuM configuration • Name of the container containing the former
Functionality:	1. Configure XtratuM with the default configuration for the chosen codebase 2. If the user chooses to use a menu, prompt the user with the XtratuM configuration menu
Depends on:	[SBC-06]
References:	[SR-FN-03] [UC-01]

SBC-03	
Name:	XtratuM configuration exporter
Type:	Script/Command
Parent:	[CMP-01]
Properties:	• XtratuM configuration files • Name of the container where they reside • Target path for the export file
Functionality:	1. Pack configuration files in an archive file 2. Copy the file to the target path
Depends on:	[SBC-06], Archiver
References:	[SR-FN-04] [UC-01]

SBC-04	
Name:	XtratuM configuration importer
Type:	Script/Command
Parent:	[CMP-01]
Properties:	• Path of archive file containing exported XtratuM • Name of the container where to import the configuration
Functionality:	1. Restore the configuration files in the archive file on their respective locations
Depends on:	[SBC-06], Archiver
References:	[SR-FN-05] [UC-01]

SBC-05	
Name:	XtratuM installer
Type:	Script/Command
Parent:	[CMP-01]
Properties:	• Name of the new container • Name of the container where XtratuM will be built • Host-Container directory mappings
Functionality:	1. If XtratuM is not configured, calls [SBC-02] 2. Builds XtratuM using [SBC-01] and [SBC-02] 3. Creates a new system build image with XtratuM ready to install
Depends on:	[SBC-01], [SBC-02], [SBC-06]
References:	[SR-FN-06] [UC-03]

SBC-06	
Name:	XtratuM build container image
Type:	Container image
Parent:	[CMP-01]
Properties:	• XtratuM codebase • Hardware target for XtratuM codebase • Name of the container • XtratuM build dependencies
Depends on:	XtratuM, XtratuM build dependencies, Container manager
References:	[SR-NF-09]

SBC-07	
Name:	System project creator
Type:	Script/Command
Parent:	[CMP-02]
Properties:	• Creation path • Name of the system • Hardware target
Functionality:	1. Create a [SBC-11]-based container with the name of the system for the hardware target 2. Install XtratuM in the newly created container 3. Create the default system working tree
Depends on:	[SBC-11], [SBC-16]
References:	[SR-FN-08] [SR-FN-20] [SR-FN-13] [UC-04]

SBC-08	
Name:	System project builder
Type:	Script/Command
Parent:	[CMP-02]
Properties:	• Name of the system • Valid system source tree
Functionality:	1. Run the [SBC-11]-based container associated to the name of the system to build each application of the system 2. Run the same container to pack the system 3. Copy the generated executable in the host environment 4. Mantain a cache of object code in [SBC-11]
Depends on:	[SBC-11]
References:	[SR-FN-09] [SR-NF-10] [SR-NF-11] [SR-NF-12] [UC-06]

SBC-09	
Name:	System project deployer for the Zynq7000SoC
Type:	Script/Command
Parent:	[CMP-02]
Properties:	• Name of the system • Built system executable
Functionality:	1. Set up the SoC through Xilinx SDK's XMD 2. Load the built system executable in the SoC 3. Wait for user commands for the XMD CLI
Depends on:	[SBC-08], Xilinx SDK
References:	[SR-FN-17] [UC-07]

SBC-10	
Name:	System project deployer for virtual machines
Type:	Script/Command
Parent:	[CMP-02]
Properties:	• Name of the system • Built system executable • QEMU configuration (architecture, board)
Functionality:	1. Run QEMU with the command line needed to execute the system executable for the architecture specified for the system
Depends on:	[SBC-08], QEMU
References:	[SR-FN-18] [UC-07]

SBC-11	
Name:	System build container image
Type:	Container image
Parent:	[CMP-02]
Properties:	• XtratuM installation • XtartuM dependencies
Depends on:	XtratuM dependencies, Container manager
References:	[SR-FN-09] [SR-NF-10] [SR-NF-11] [SR-NF-12] [UC-06]

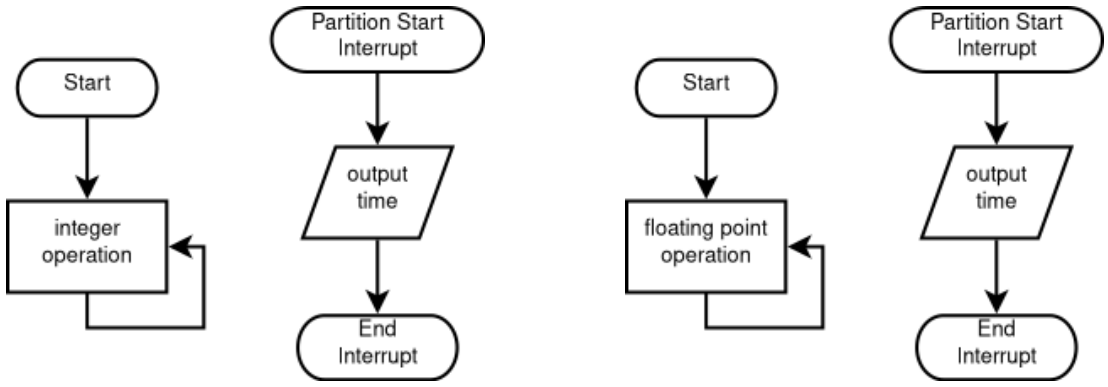
SBC-16	
Name:	Default system
Type:	System
Parent:	[CMP-02]
Functionality:	1. This default system will be a dummy system single "Hello World" application
Depends on:	XtratuM
References:	[SR-FN-08] [UC-04]

SBC-18	
Name:	CPU stress time benchmark
Type:	Application
Parent:	[CMP-03]
Functionality:	1. Perform CPU intensive operations 2. Log results as output
Depends on:	XratuM
References:	[SR-FN-21] [SR-NF-13]

SBC-19	
Name:	Memory stress time benchmark
Type:	Application
Parent:	[CMP-03]
Functionality:	1. Perform CPU intensive operations 2. Log results as output
Depends on:	XtratuM
References:	[SR-FN-22] [SR-NF-13]

SBC-20	
Name:	C++ support library
Type:	C library
Parent:	[CMP-04]
Properties:	• Basic standard C++ header files • Implements `atexit`, `new`, `delete` • Calls global object constructors and destructors on program initialization and finalization
Depends on:	XtartuM
References:	[SR-FN-27] [SR-FN-28] [SR-NF-08]

SBC-21	
Name:	Memory manager
Type:	C library
Parent:	[CMP-04]
Properties:	• Implements `malloc` • Implements`free`
Depends on:	XtratuM
References:	[SR-FN-01] [SR-FN-02]



(a) ALU mode. (b) FPU mode.

Fig. 4.4. CPU stress benchmarks application flow.

4.4. Benchmark design

Some of the subcomponents defined above (SBC-18 and SBC-19) are benchmarks that need to be designed. Both the CPU stress and the memory stress are time benchmarks: they will monitor the time passed between events to verify that, despite the hardware being busy, time frames are fulfilled.

In this section we will design them specifying what they should do, including the flow of the programs and the format of the outputs.

4.4.1. CPU stress time benchmarks (SBC-18)

The CPU stress time benchmark will overload the CPU with the purpose of evaluating if the partitions execute at the right time. For this the applications of the system will execute an infinite, and use the hardware interruptions generated by XtratuM on partition time slot start to log and verify the beginning of time frames.

In many systems, the FPU is separate from the main CPU — for example, in the case of the Zynq7000 it can be disabled completely. Because of this, the benchmark will include a configuration option to choose whether the work done inside infinite loop is integer or floating point arithmetic. Figure 4.4 shows what the flow of both of the configurations should be.

A basic CPU benchmark should consist on two or more partitions following the same flow (ALU or FPU). Verification of the benchmark should consist in checking whether the partition time slot beginnings are as configured in the XtratuM system configuration.

4.4.2. Memory stress time benchmarks (SBC-19)

The memory stress time benchmark will be similar to the CPU stress benchmark, but instead of being busy executing arithmetic operations the hardware will execute memory accesses in the infinite loop.

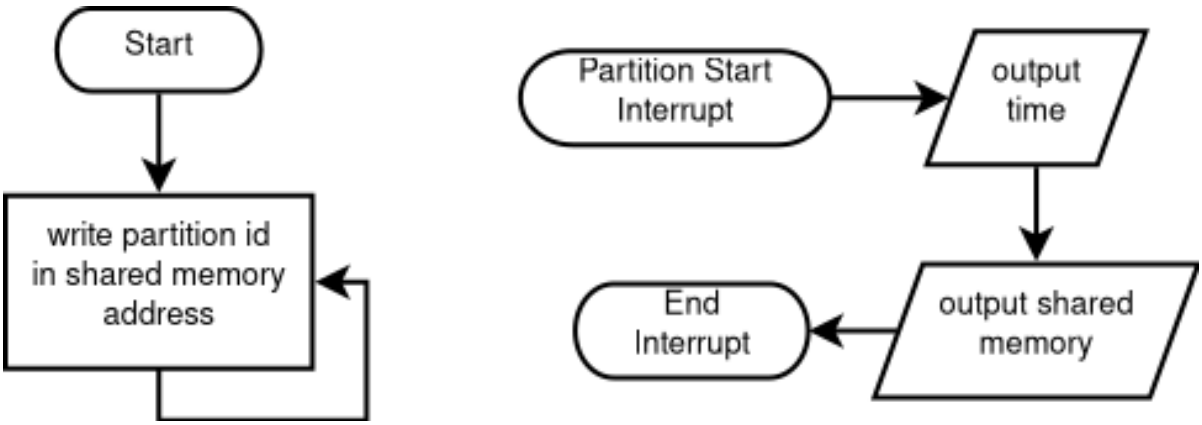


Fig. 4.5. Memory stress benchmark application flow.

Memory accesses will be done in a shared region: each partition will write in the same address its partition ID, so that at the beginning of the time slot the partition prints the ID of the previous one. Partitions will also log the time of the time slot start. Figure 4.5 shows the flow of the partitions.

A basic memory benchmark should consist on two or more partitions following the defined flow. Verification of the benchmark will consist in checking that the read values from memory are the correct ones, and that the beginning of the time slots are not performed late.

4.5. Traceability matrices

Tables 4.1 and 4.2 show the traceability of software requirements in components and subcomponents respectively. As it can be seen, all software requirements are covered by at least one component, and every subcomponent covers at least one software requirement.

Traceability matrices are good to appreciate the cohesion of the components of the system. If a software requirement is implemented by many components, cohesion can be determined as low. The matrices show that both at the component and subcomponent levels, cohesion is present in the components of the design.

CMP-	01	02	03	04
SR-FN-01				•
SR-FN-02				•
SR-FN-03	•			
SR-FN-04	•			
SR-FN-05	•			
SR-FN-06	•			
SR-FN-07	•			
SR-FN-08		•		
SR-FN-09		•		
SR-FN-13		•		
SR-FN-17		•		
SR-FN-18		•		
SR-FN-20		•		
SR-FN-21			•	
SR-FN-22			•	
SR-FN-27				•
SR-FN-28				•
SR-NF-01	•			
SR-NF-08				•
SR-NF-09	•			
SR-NF-10		•		
SR-NF-11		•		
SR-NF-12		•		
SR-NF-13			•	

Table 4.1. COMPONENTS — SOFTWARE REQUIREMENTS TRACEABILITY MATRIX.

SBC-	01	02	03	04	05	06	07	08	09	10	11	16	18	19	20	21
SR-FN-01																•
SR-FN-02																•
SR-FN-03		•														
SR-FN-04			•													
SR-FN-05				•												
SR-FN-06					•											
SR-FN-07	•															
SR-FN-08							•					•				
SR-FN-09								•			•					
SR-FN-13							•									
SR-FN-17									•							
SR-FN-18										•						
SR-FN-20							•									
SR-FN-21													•			
SR-FN-22														•		
SR-FN-27															•	
SR-FN-28															•	
SR-NF-01	•															
SR-NF-08															•	
SR-NF-09	•					•										
SR-NF-10								•			•					
SR-NF-11								•			•					
SR-NF-12								•			•					
SR-NF-13													•	•		

Table 4.2. SUBCOMPONENTS — SOFTWARE REQUIREMENTS TRACEABILITY MATRIX.

5. IMPLEMENTATION

In this chapter we explain the reference implementation of the designed framework that is provided alongside this work. This implementation is called `xmfw` (XtratuM Framework).

The framework has been implemented to target developers with a specific profile. It is assumed that developers of nanosatellites are experienced in programming and they are at ease with using a Command Line Interface to set up environments, frameworks and debugging embedded applications. Part of this work is aimed to reduce the burden of some of the common tasks developers must perform in their job.

With this in mind, the *Script/Utility* and the *Script/Command* components specified in the Design have been implemented in standard shell languages, like the POSIX shell and the Bash shell. It is also assumed that the developers' host envionment will be UNIX-like, like MacOS or Linux, as this is predominant among this profile and easies a lot the development process when compared with alternative environments like MS Windows.

For containers, Docker will be used as the container manager. While it is true that there are alternatives to Docker and the OCI standard has made it possible to customize them more freely, the Docker environment, with Dockerhub and its container image management system, will be very useful to make an easy implementation of the designed system. What is more, Docker is still the industry standard, and many developers will have it installed and know how to use it already.

The C++ and C memory management support libraries have been implemented for XtratuM for the Zynq7000 because this is higher priority, taking into account the preferences of the client.

5.1. Conventions

When implementing the framework, an approach of convention over configuration has been taken, in order to simplify the solution and accelerate development. Regarding container image names, the conventions are:

- XtratuM build images are named in the format:

  `[repo/]xmfw:[tag][-arch][-var]`

  where `[arch]` is the target architecture, `[name]` an arbitrary string for the variant and `[tag]` the tag of the image, preferably the XtratuM version plus any necessary extension. `[arch]` cannot contain hyphens (`-`), and is optional.

- XtratuM system build images are named in the format:

`[repo/]xmfw-[base]-system:[tag][-arch][-var]`

where `[arch]`, `[var]` and `[tag]` are as above and `[base]` is the image in which the build image is based. If the system base image is based in a XtratuM build image, then the name of the image will be just `xmfw-system`. `[arch]` cannot contain hyphens (`-`), and is optional.

Regarding paths in containers, the following conventions have been followed:

- `/opt/toolchain` will contain the target toolchain. This path will be the target of a mounted volume from the host machine. Developers can specify paths inner to that toolchain that contain its binaries, but keeping in mind that some toolchains require not just the binaries but also resources in parent and sibling directories.
- `/opt/xtratum` will contain the XtratuM codebase in XtratuM build images.
- `/opt/xm-install` will be the XtratuM installation script in System build images.
- `/opt/xm` will be the XtratuM installation path in System build containers.
- `/opt/system` will be the target for the mounted volume containing the system project tree in System build containers.

5.2. CMP-01: Framework manager

The framework manager is comprised of a set of scripts that implement its components, some Docker images and well-defined conventions that standardize the environment of the framework.

5.2.1. XtratuM build container image (SBC-06, SBC-01)

Following the dependencies of components, the first one to build is the XtratuM build container image (SBC-06).

The framework uses a Docker container to build XtratuM. This container is based on an image that includes all the required dependencies to build the XtratuM installer and the XtratuM source codebase. Many adaptations of XtratuM for the same version have different source codebases but common dependencies, so these are abstracted in another container image called the base image.

With this work, Dockerfiles are included to build the base image and also source images for the repositories that are available, i.e.: XtratuM v1.0.8 for SPARC, XtratuM v2.0.5 for ARM and XtratuM v2.6.0 for IA32, and another image of XtratuM v2.0.5 for ARM plus the XALC++ extensions — these images are tagged, respectively, as follows:

```
xmfw:base
xmfw:1.0.8-sparc
xmfw:2.0.5-arm
xmfw:2.6.0-ia32
xmfw:2.0.5-arm-xalcxx
```

The base image includes two scripts, `build` and `config` that are in the `$PATH` and help automate the building process when installing (this is SBC-01). The Dockerfiles that define the images fetch the XtratuM codebase from custom repositories in GitHub. They are available in the danoloan/xmfw public DockerHub repository. They are also included in the appendix D

5.2.2. `xmfw-install` script (SBC-02, SBC-03, SBC-04, SBC-05)

This script allows for the configuration management and installation of XtratuM codebases that are in the images previously described. The usage of the script is as follows:

```
xmfw-install [-mv] [-i PATH] [-o PATH] <xmfw image> <base image>
```

Options MUST come before the arguments. The options and arguments of this script are described below:

- `-m`: Prompt the configuration menu when configuring XtratuM. If this option is not provided, the default configuration will be used.
- `-v`: Verbose output. Gives additional information in the build.
- `-c`: Only configure, do not build XtratuM or produce any image. If this flag is active, the `-o` option must be present.
- `-i PATH`: Import configuration from `PATH` in gzip format.
- `-o PATH`: Export configuration to `PATH` in gzip format.
- `-a ARCH`: Which architecture configure XtratuM for.
- `<xmfw image>`: XtratuM build image to use (see Section 5.2.1).
- `<base image>`: System base image to use. This image must support the development of the application, so `xmfw` images are recommended but not mandatory. This argument is optional; if not provided, the `xmfw` image will be used as system base image.

This script first builds XtratuM using the codebase included in the `<xmfw image>` and creates a new system base image based on `<base image>` with XtratuM ready to be installed. The newly created image can be then exported using Docker (with the `docker save` command), thus enabling the implementation of the work flow showed in Figure 2.8. `<xmfw image>` must be named according to the convention (see Section 5.1).

5.3. CMP-02: System project manager

The system project manager allows creating and building systems based on XtratuM. It is, like the framework project manager, comprised of container images (the system build images) and a helper script (`xmfw-system`).

5.3.1. System build images (SBC-11)

The *system build images* are created by the `xmfw-install` script. However, they are based on *the system base image* that must also be defined. This last image can be defined by the developer according the application framework that they choose to use for the system — however, the previously described `xmfw:base` image (see Section 5.2.1) should suffice for building simple systems based solely on XtratuM.

5.3.2. `xmfw-system` (SBC-07, SBC-08)

This script enables the creation and building of systems. As explained in Section 4.2, every system must have an associated *system build container* in charge of building the system. This script manages the creation and invocation of this container. It also allows to define a base benchmark or example application when creating a system.

It was decided to integrate the SBC-15 component with SBC-07 because this way we can implement both default templates and benchmarks as XAL examples, that will be the default templates on system creation.

This script implements two commands: `create` and `build`. Its usage is as follows:

```
Usage
    xmfw-install [-lv] <command> <args>

COMMANDS
    create <image> <path> [template]
    build  <path>
```

Options MUST come before the arguments. The options and arguments of this script are described below:

- `-l`: List available templates and exit.
- `-v`: Verbose output. Gives additional information in the build.
- `<image>`: System build image created by `xmfw-install` to use.
- `<path>`: Path of the system to create or build.
- `<template>`: Name of the benchmark to use as template.

The `create` command will create the system only if the destination path is empty and the system build image exists. The script does not check that the system build image

can build XtratuM — it only requires that the XtratuM installation script is inside and can be executed successfully. This command will create the working tree of the system in the destination path and create a container able to build its contents. The created container follows the naming conventions described in Section 5.1. A template must be chosen when creating a system, that will be copied from the XAL examples included with XtratuM.

The `build` command will build the system residing in the target path. If the container of the system does not exist, it will prompt for its creation. The built binaries are created by the root user of the container; this is an issue that should be fixed in future revisions of the implementation.

5.3.3. Default systems (SBC-16)

The default and systems included with the framework will be the example applications included with XAL and the benchmarks. These can be used by specifying them as a template in the creation of a system in the `xmfw-system` script.

5.3.4. Application management

Application management has not been automated. In this section we describe the steps required to add a partition to a system based on the `hello-world` example included in the `xmfw:2.0.5-arm` image.

XtratuM uses GNU make as its application building system, and includes some standard rules with XAL. Figure 5.1 shows the Makefile of the `hello-world` application. XtratuM builds the file `resident_sw` as the result executable. For this it packs the `.xef` files using `xmpack` and the command line in the `PACK_ARGS` variable. To build each `.xef` file, XtratuM uses the file with the same name without the extension — this is what rules in lines 20 and 23 are for.

To add an application, the developer must take the following steps:

1. Add the partition to the `xm_cf.[arm].xml` XtratuM configuration file.
2. Configure there its name, memory regions and time slots.
3. Add a new `.xef` file to the `PARTITIONS` variable.
4. Add a rule building a file named the same as `.xef` file without the extension. This file must be the standalone, statically linked executable of the partition.
    - The partition should be linked to have its code based on the start address of the memory assigned to the partition. This is what the `-Ttext=` option is for in lines 21 and 24.
    - To dynamically get the appropriate value, the `xpathstart` utility is provided by XtratuM, and its executable path is available on the `XPATHSTART` variable.

- This utility takes two arguments: the ID of the partition and the configuration file; and outputs the start address of the partition. The configuration file's name is available in the `XMLCF` variable defined in line 12.

5. Add the `.xef` file of the new partition to the command line of `xmpack`. This is done by adding `-p <partno>:<file.xef>` to the `PACK_ARGS` variable in line 26, where `<partno>` is the ID for the new partition.

After these steps, `xmfw-build` should build the system with the new partition successfully, and generate the file `resident_sw` that can be loaded into the hardware to be executed.

5.3.5. Systems deployment (SBC-09, SBC-10)

Deployment of systems has not been automated. In this section we describe the steps required to deploy system built with the `xmfw:2.0.5-arm` and `xmfw:2.6.0-ia32` images.

IA32 and QEMU Once the `resident_sw` file is generated for the IA32 target using `xmfw-build`, it can be executed with the following command line:

```
qemu-system-i386 -kernel resident_sw -nographic
```

ARM and Zynq7000 SoC Once the `resident_sw` file is generated for the IA32 target using `xmfw-build`, it should be loaded to the board. To do this, the XMD utility of the Xilinx SDK is used. This utility can be used standalone by executing its binary from the command line of a terminal. Figure 5.2 shows the secuence of XMD command needed to deploy the produced file in a Zedboard. In order to load the software into the board, the board itself must first be configured. In the case of the Zedboard, the hardware configuration is included with the Xilinx SDK — this is line 2. The path of the `.tcl` file is relative to the Xilinx SDK.

As a remark, the default XtratuM configuration enables the L2 cache, but this makes the applications stay in an infinite loop trying to successfully execute the `strex` instruction. In order to make built applications work with the Zedboard, the L2 cache must be disabled when configuring XtratuM before installation using the menuconfig prompt.

5.3.6. Framework support

While trying to compile the reference implementation of F', Ref, with XtratuM, many compilation and linking errors were thrown, and all of them were eventually resolved successfully, some by adapting the XtratuM and XAL libraries and some by disabling some components for which it is almost impossible for XtratuM to provide support — this resulted in F' building successfully, although the built systems could not be executed correctly. In this section we explain how to replicate the results.

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=../..

CONTAINER = container.bin
RSW = resident_sw

all: $(CONTAINER) $(RSW)

include $(XAL_PATH)/common/rules.mk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SRCS := $(sort $(wildcard *.c))
OBJS := $(patsubst %.c,%.o, $(SRCS)) $(patsubst %.S,%.o, $(ASRCS))

PARTITIONS=partition0.xef #partition1.xef

partition0: $(OBJS) $(XMLCF)
    $(TARGET_LD) -o $@ $< $(TARGET_LDFLAGS) -Ttext=$(shell $(XPATHSTART
        ) 0 $(XMLCF))

partition1: $(OBJS) $(XMLCF)
    $(TARGET_LD) -o $@ $< $(TARGET_LDFLAGS) -Ttext=$(shell $(XPATHSTART
        ) 1 $(XMLCF))

PACK_ARGS=-h $(XMCORE):xm_cf.xef.xmc \
    -p 0:partition0.xef# \
    -p 1:partition1.xef

$(CONTAINER): $(PARTITIONS) xm_cf.xef.xmc
    $(XMPACK) check xm_cf.xef.xmc $(PACK_ARGS)
    $(XMPACK) build $(PACK_ARGS) $@
```

Fig. 5.1. `hello-world` Makefile.

```
connect arm hw
source data/embeddedsw/lib/hwplatform_templates/zed_hw_platform/
   ps7_init.tcl
ps7_init
dow resident_sw
continue
```

Fig. 5.2. Sequence of commands needed to deploy XtratuM Resident Software in the Zedboard.

F' uses CMake as its back-end build system, with a set of python utilities that provide a user-friendly front-end. In order to build F' with XtratuM, the first step is to create the CMake files for the toolchain and platform used — in our case, they will be the Xilinx SDK toolchain with the XtratuM platform.

Some components of the Ref application must be disabled partly or completely, due to the lack of platform APIs and C++ ABIs and builtins provided by XtratuM and the toolchain. These components are: `LinuxSerialDriver`, `LinuxTimeImpl`, `FileManager` and `TcpClient`. Also, the `SignalGen` component inside the application uses unimplemented functions to generate random and sine signals, so these should be disabled too.

In appendix C both CMake files for F' are provided. The modified F' version that is able to build with XtratuM is available as a fork in GitHub [53]. A Dockerfile is also provided to build a System base image with all the necessary dependencies to build F'. This image will be named `fprime`. XtratuM can be installed in this base image through the utilities of the framework.

After installing `xmfw:2.0.5-arm-xalcxx` in the `fprime`, a `xmfw-fprime-system:2.0.5-arm-xalcxx` image is created. The following steps allow building Ref from this point:

1. Create an interactive Docker container:

   ```
   docker create -it --name fprime \
       -v $XILINX_TOOLCHAIN:/opt/toolchain \
       [ more mounts and options ] \
       xmfw-fprime-system:2.0.5-arm-xalcxx bash
   ```

2. Install XtratuM:

   ```
   /opt/xm-install -- -a -d /opt/xm -t /opt/toolchain/gnu/arm/lin/bin
   ```

3. Go to the Ref directory and activate the `venv` environment

   ```
   cd /opt/fprime/Ref && source ../venv/bin/activate
   ```

4. Execute `fprime-util generate xtratum-zynq7000`
5. Execute `fprime-util build xtratum-zynq7000`
6. The target binary should be located in `build-fprime-automatic-xtratum-zynq7000/bin/Xtratum/Ref`

5.4. CMP-03: Benchmarks

The client SENER has expressed its preference of providing support for the development of nanosatellite applications in ARM-based systems, so in this work we will evaluate the framework for the Zynq7000 SoC. The framework versions for IA32 and SPARC have not been requested by the client (UR-RS-06 and UR-RS-06 are optional requirements),

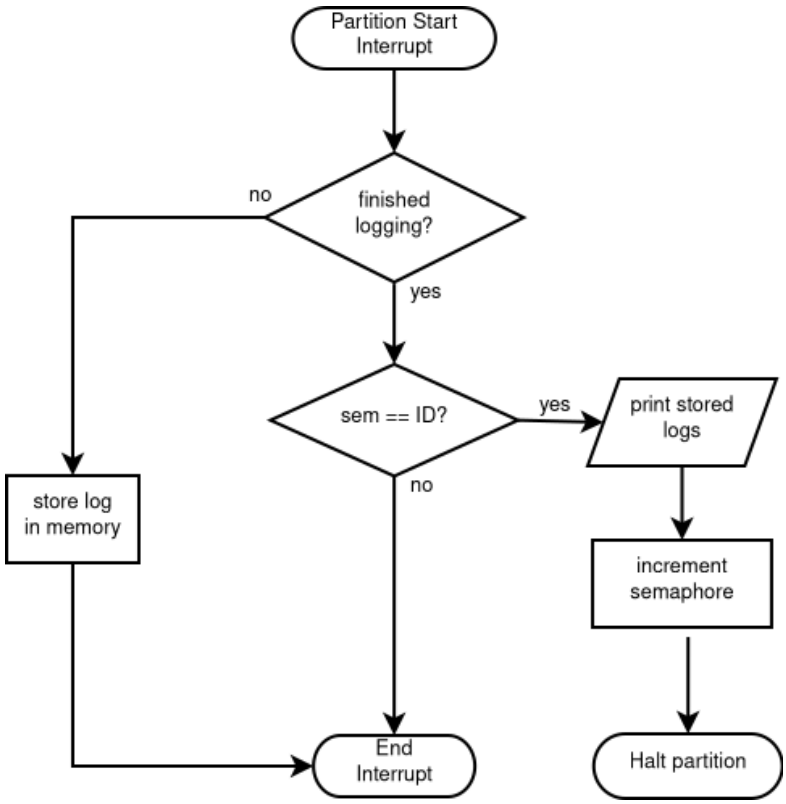


Fig. 5.3. Semaphore system used to produce UART output in benchmarks.

so they will not be evaluated. Because of this, the benchmarks will be implemented for the platform XtratuM ARM v2.0.5.

Xtratum provides support for detecting the start of a slot assigned to a partition using interruptions. For ARM v2.0.5, XtratuM offers the following definitions:

- `InstallIrqHandler()` to install handler functions for interruptions.
- `XM_VT_EXT_CYCLIC_SLOT_START` is the ID for the cyclic slot start event, implemented through a trap instruction in ARM systems.
- `HwSti()` activates interruptions.
- `XM_clear_irqmask()` and `XM_set_irqmask()` control the interrupt mask. All extended interruptions are masked by default.

The benchmarks have been implemented as designed in the previous chapter. Due to delay and interferences, the standard UART output cannot be reliably used to verify the execution of the benchmark in very high frequencies of the cyclic scheduling. Because of this, each partition stores the start times of each cyclic slot in memory and prints them all before halting. This also allows reducing the execution time of the logging itself — when not printing, the interruption handler has been measuerd to complete execution in 4-6 μs.

To prevent partitions writing output over each other when finishing up, a custom semaphore has been implemented. The semaphore is an integer stored in a shared memory address. When partitions start, they initialize it to zero. When partitions finish

Fig. 5.4. Default scheduling of benchmarks.

logging times, they check whether the semaphore is equal to their partition ID before printing them. If it is not, they wait for the next cycle. When the semaphore indicates it is the turn of a partition, it prints the logged times and increments the semaphore by one, giving way to the next. This process is described in Figure 5.3.

CPU stress benchmark (SBC-18) This benchmark has two modes as designed previously: one to stress the ALU and the other to stress the FPU. ALU mode is the default, and FPU mode can be activated defining `FPU_MODE` on compilation. The output is a list of the cycle start times the partitions have recorded in the following format:

```
[ <partno>:<start time> ]
```

It will be verified if the start times printed follow the scheduling specified in the configuration file — Figure 5.4 shows the default scheduling for this benchmark.

Memory stress benchmark (SBC-19) The memory stress benchmark produces output similar to the previous benchmark:

```
[ <partno>:<start time>, <prev> ]
```

where `prev` is the ID number of the partition previously executed. The default scheduling of this benchmark is the same as the previous (see Figure 5.4).

The implementation of these benchmarks is included in the directory `./user/xal/examples` of XtratuM for ARM and are available in the `xmfw:2.0.5-arm` and `xmfw:2.0.5-arm-xalcxx` images, and all the images where this version of XtratuM is installed. Their code and configuration files are appended in A and B

Integration with other components

These benchmarks are integrated in the XtratuM codebase, so that the images `xmfw:2.0.5-arm` and `xmfw:2.0.5-arm-xalcxx` have them ready and available to be used as templates for new systems.

```
struct __buddy_entry {
    bool free;      // u8 - 1B: free block
    xm_u8_t order;  // u8 - 1B: order of
       free block
};
```

Fig. 5.5. Buddy memory allocator block entry data structure.

5.5. CMP-04: XALC++

To implement the memory management support library, a buddy memory allocator has been implemented. Also, XtratuM headers have been made compatible with C++, standard C++ headers have been included and GNU make rules have been added to allow building C++ applications.

The memory management component is implemented in the `user/xal/common/stdlib.c` file of the XtratuM codebase included in the `xmfw:2.0.5-arm-xalcxx` image. The original `std_c.c` file was divided into the standard `string.c` and `stdlib.c` files.

5.5.1. Memory manager (SBC-21)

In the `stdlib.c` file a simple implementation of `malloc` and `free` using a buddy memory allocator is included. Appendix E includes the source code of this allocator.

The memory manager uses the first memory region tagged with the `flag0` flag in the `flags` attribute as specified in the XML XtratuM configuration file (see appendix E for an example). This region cannot be the same as the one where the executable is loaded, as this would cause conflict with the `.text` region and provoke a memory access violation.

This region is divided in two parts: the "buddy block" list and the heap. The stack of the partition is not relevant as it is managed by XtratuM in the main memory region. The block list is stored at the end of the memory, and the heap starts at the beginning.

The list is a list of entries indicating the status of a block, and is part of the implementation of the buddy memory allocator. The size of each entry in the list is 2 Bytes as of the current implementation. Figure 5.5 shows the data structure used.

The heap is divided in blocks of size 2^k. The heap and the number of blocks are related, and the proportion of the space occupied by the buddy block list remains constant with the size of the memory and only depends on the block size, as shown in formula 5.2. In these formulas, N is the number of blocks, k the block byte order (minimum block order), H is the heap size, M the memory size and T the size of the block entry in the list.

$$N = \frac{H}{2^k}; \quad H = 2^n; \quad M = 2^n + 2^{n-k} \cdot T \tag{5.1}$$

$$p = \frac{N \cdot T}{M} = \frac{2^n}{2^n + 2^{n-k} \cdot T} = \frac{2^k \cdot 2^{n-k}}{2^{n-k} \cdot (2^k + T)} = \frac{2^k}{2^k + T} \tag{5.2}$$

Formula 5.1 shows how to calculate the memory needed for a desired heap size. Due to the nature of the buddy allocator, the number of blocks in the heap must be a power of two — were the memory a byte smaller, the total size of the heap would be halved, so it is important to use this formula to calculate the appropriate value for the memory region.

Parameter k can be tweaked through the XtratuM configuration and defaults to 6. This configuration value is stored in the defined macro `BLOCK_BYTE_ORDER`. For bigger values of k, the buddy list entries will take less memory but the heap will be more prone to fragmentation. Figure 5.6 shows the configuration option in the menuconfig of XtratuM.

5.5.2. C++ support library (SBC-20)

The initial approach for the implementation of these extensions was to provide support for F', as explained in Section 5.3.6. Compilation and linking errors while trying to build the reference application Ref were the source of the changes described in this section.

Standard headers

The following changes have been made to the standard headers included in XtratuM and XAL to provide support for C++:

- `assert.h` has been modified so that the `assert()` macro works.
- `stdio.h` now includes the standard `vsnprintf()` function.
- `string.h` now includes the standard `strnlen()` function.
- All standard C headers have been added the `extern "C"` directive in the case C++ is being used.
- `core/include/arm/xmconf.h` file has been modified to solve a warning comprising an anonymous union that fired when compiling with C++.
- `core/include/guest.h` has been modified to solve implicit casting warnings shown by the C++ compiler.
- Standard C++ headers were copied from the GNU GCC toolchain, these were: `cmath cstdbool cstdint cstdio cstdlib cstring exception` and `new`. Some parts of these headers have been omitted because the library does not provide implementation for them.

`new` and `delete`

The `new` and `delete` operators are trivially implemented by calling `malloc` and `free` as needed. The object bind overloads of these operators are defined in the `new` header.

C++ application development

The default rules needed to compile C++ applications with the default toolchain have been added to the XAL GNU make configuration, so adapting any example to include C++ should be straightforward. If the file where `PartitionMain` is a C++ source file, it should include the header `xal.h` — once this is done, object files required by any target will be compiled using the C++ compiler when their respective source files are named with the `.cpp` extension. Linking should be performed in the same way as any C application. An example `c++` example application is included in the `xmfw:2.0.5-arm-xalcxx`.

C++ global object constructors and destructors

The global object constructors and destructors are not called as defined in the standard. To achieve this, the code of these functions must be loaded at run time from the partition code into the special regions `.init_array` and `.fini_array` in the case of ARM based systems. However, this task requires modifying how XtratuM loads the `.xef` file format it defines — this falls out of the scope of this work.

Example applications

The `malloc` example system works as a benchmark to verify the correct functioning of the SBC-21 memory manager component. It stresses the system by trying to allocate memory in different ways. It verifies that the maximum size can be allocated, but not one byte more.

The `c++` example system is a template for C++ applications. All the modifications mentioned in Section 5.5.2 are applied here.

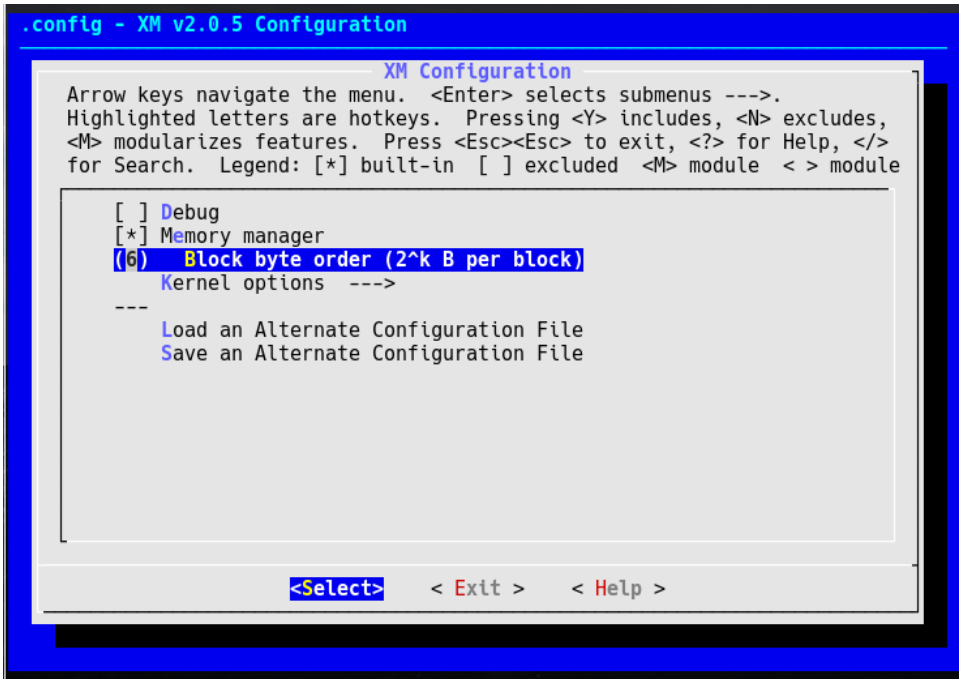


Fig. 5.6. Menuconfig option to configure the memory manager allocation blocks.

6. EVALUATION

The evaluation of the implemented framework will be done in three steps: first, we will verify the correct functioning of the implemented components. This will be done according to the priority of requirements associated to each component: through thorough testing of different input classes for the scripts and images and through simple example verification for the XALC++ library functions. Then, we will evaluate the framework using the benchmarks implemented.

6.1. Verification

The verification is a process that consists in checking whether the implementation proposed matches the designed components specification. In this section we verify the framework and system manager scripts, Docker images, benchmarks and library functions implemented.

6.1.1. Framework and system manager scripts verification

The verification of the `xmfw-install` and `xmfw-system` scripts will be performed using exhaustive, well defined test cases based in a class analysis of the inputs. This way, not only the correctness of separate components will be verified, but also their integration with the rest.

In this process, all the XtratuM build images but the `xmfw:1.0.8-sparc` will also be verified. This image is excluded because its codebase is too old to be considered useful, and in addition we have not been able to gain access to a compatible SPARC toolchain.

Each test case defines inputs, an action script defining the sequence of steps to execute and the conditions that must hold after the execution of the actions. The inputs are the following:

`xmfw-install:`

- **I1** `-m` option (menuconfig)
- **I2** `-o` option (export)
- **I3** `-i` option (import)
- **I4** XtratuM build image
- **I5** System base image
- **I6** `-c` option (config only)
- **I7** Toolchain
- **I8** `-a` option (architecture)

`xmfw-system create:`

C1 System build image

C2 System creation path

C3 System container (does it exist)

C4 Toolchain

C5 Template

`xmfw-system build:`

B1 System path

B2 System container (does it exist)

B3 System build image

The possible values for the inputs are described in table 6.1. They are divided in two classes: valid (V) and not valid (N). Valid inputs can be verified together, but invalid inputs must be tested separately. Inputs **I7**, **C4** and **B4** are handled by the same code so they just need to be verified once. Following is the definition of test cases:

Note: The environment required to perform these tests needs to have the Xilinx SDK installed or symlinked in `/opt/xilinx`.

	V1	V2	V3
I1	present	not present	–
I2	valid path	not present	–
I3	valid path	not present	–
I4	2.0.5-arm	2.6.0-ia32	2.0.5-arm-xalc++
I5	empty	valid (xmfw)	valid (not xmfw)
I6	no	yes, with export	–
I7	valid	–	–
I8	not present	supported	–
C1	2.0.5-arm	2.6.0-ia32	2.0.5-arm-xalc++
C2	free	–	–
C3	does not exist	–	–
C4	valid	–	–
C5	exists	–	–
B1	exists	–	–
B2	exists	does not exist	–
B3	2.0.5-arm	2.6.0-ia32	2.0.5-arm-xalc++

(a) VALID CASES.

	N1	N2
I1	invalid option	–
I2	invalid path	–
I3	invalid path	–
I4	does not exist	–
I5	does not exist	not conventional
I6	yes, w/o export	–
I7	does not exist	binaries not inside
I8	not supported	–
C1	does not exist	not conventional
C2	occupied	parent does not exist
C3	container exists	–
C4	does not exist	binaries not inside
C5	does not exist	–
B1	does not exist	not a directory
B2	–	–
B3	does not exist	not conventional

(b) INVALID CASES.

Table 6.1. VALUES FOR TEST CASES INPUTS.

<table>
<tr><th colspan="8">TC-01</th></tr>
<tr><th>I1</th><th>I2</th><th>I3</th><th>I4</th><th>I5</th><th>I6</th><th>I7</th><th>I8</th></tr>
<tr><td>V1</td><td>V2</td><td>V1</td><td>V1</td><td>V1</td><td>V1</td><td>V1</td><td>V1</td></tr>
<tr><th>C1</th><th>C2</th><th>C3</th><th>C4</th><th>C5</th><th>B1</th><th>B2</th><th>B3</th></tr>
<tr><td>V1</td><td>V1</td><td>V1</td><td>V1</td><td>V1</td><td>V1</td><td>V2</td><td>V1</td></tr>
<tr><td colspan="2">Preconds.:</td><td colspan="6">Docker image <code>danoloan/xmfw:2.0.5-arm</code> available
<code>/tmp/config.tar.gz</code> available</td></tr>
<tr><td colspan="2">Actions:</td><td colspan="6">

```
# Toolchain
# /opt/xilinx/
# /opt/xilinx/gnu/arm/lin/bin/

set -e

name=$(mktemp -d)

xmfw-install -m -i /tmp/config.tar.gz danoloan/xmfw
    :2.0.5-arm

if [ -z "$(docker image ls --quiet --filter 'reference=
    xmfw-system:2.0.5-arm')" ]; then
    echo 'error' >&2
    exit 1
fi

docker rm -f xmfw-$name-build

xmfw-system create xmfw-system:2.0.5-arm $name/test hello
    -world <<EOF
/opt/xilinx
/opt/xilinx/gnu/arm/lin/bin
EOF

if [ -z "$(docker ps --all --quiet --filter "name=xmfw-
    test-build")" ]; then
    echo 'error' >&2
    exit 1
fi

docker rm xmfw-test-build

xmfw-system build $name/test <<EOF
xmfw-system:2.0.5-arm
/opt/xilinx
/opt/xilinx/gnu/arm/lin/bin
EOF

echo $name
```

</td></tr>
</table>

Postconds.:	• Menuconfig prompted • Configuration loaded properly • Image xmfw-system:2.0.5-arm created • Image xmfw-system:2.0.5-arm has /opt/xm-install • xmfw-test-build container created • resident_sw generated in system worktree • resident_sw can be executed correctly
Result:	YES

TC-02							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V1	V2	V1	V2	V2	V1	V1	V2
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V2	V1	V1	V1	V2	V1	V2	V2
Preconds.:	• Docker image `danoloan/xmfw:2.6.0-ia32` available • `/tmp/config.tar.gz` available						

Actions:	```
	# Toolchain
	# default
	
	set -e
	
	name=$(mktemp -d)
	
	xmfw-install -m -a ia32 -i /tmp/config.tar.gz danoloan/
	    xmfw:2.6.0-ia32 danoloan/xmfw:2.6.0-ia32
	
	if [-z "$(docker image ls --quiet --filter 'reference=
	    xmfw-system:2.6.0-ia32')"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	docker rm -f xmfw-test-build
	
	xmfw-system create xmfw-system:2.6.0-ia32 $name/test
	
	if [-z "$(docker ps --all --quiet --filter "name=xmfw-
	    test-build")"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	docker rm xmfw-test-build
	
	xmfw-system build $name/test <<EOF
	xmfw-system:2.6.0-ia32
	
	EOF
	``

Postconds.:	• Menuconfig prompted • Configuration loaded properly • Image xmfw-system:2.6.0-ia32 created • Image xmfw-system:2.6.0-ia32 has /opt/xm-install • xmfw-test-build container created • resident_sw generated in system worktree • resident_sw can be executed correctly
Result:	YES

TC-03							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V1	V2	V1	V3	V3	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V3	V1	V1	V1	V1	V1	V2	V3

Preconds.:	• Docker image `danoloan/xmfw:2.0.5-arm-xalcxx` available • `/tmp/config.tar.gz` available • Image test:base exists and is able to build XtratuM applications
Actions:	

```
# Toolchain
# /opt/xilinx/
# /opt/xilinx/gnu/arm/lin/bin/

set -e

name=$(mktemp -d)

xmfw-install -m -i /tmp/config.tar.gz danoloan/xmfw
    :2.0.5-arm-xalcxx test:base

if [ -z "$(docker image ls --quiet --filter 'reference=
    xmfw-system:2.0.5-arm-xalcxx')" ]; then
    echo 'error' >&2
    exit 1
fi

docker rm -f xmfw-$name-build

xmfw-system create xmfw-test-base-system:2.0.5-arm-xalcxx
     $name/test hello-world <<EOF
/opt/xilinx
/opt/xilinx/gnu/arm/lin/bin
EOF

if [ -z "$(docker ps --all --quiet --filter "name=xmfw-
    test-build")" ]; then
    echo 'error' >&2
    exit 1
fi

docker rm xmfw-test-build

xmfw-system build $name/test <<EOF
xmfw-test-base-system:2.0.5-arm-xalcxx
/opt/xilinx
/opt/xilinx/gnu/arm/lin/bin
EOF
```

Postconds.:	• Menuconfig prompted • Configuration loaded properly • Image xmfw-test-build-system:2.0.5-arm-xalcxx created • Image xmfw-system:2.0.5-arm-xalcxx has /opt/xm-install • xmfw-test-build container created • resident_sw generated in system worktree • resident_sw can be executed correctly
Result:	YES

TC-04							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V1	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	V1	V1	V1	V1	V1	V1	–
Preconds.:	• Docker image `danoloan/xmfw:2.0.5-arm` available • `/tmp/config.tar.gz` available						

Actions:	```
	# Toolchain
	# /opt/xilinx/
	# /opt/xilinx/gnu/arm/lin/bin/
	
	set -e
	
	name=$(mktemp -d)
	
	xmfw-install -i /tmp/config.tar.gz danoloan/xmfw:2.0.5-
	    arm
	
	if [-z "$(docker image ls --quiet --filter 'reference=
	    xmfw-system:2.0.5-arm')"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	docker rm -f xmfw-$name-build
	
	xmfw-system create xmfw-system:2.0.5-arm $name/test hello
	    -world <<EOF
	/opt/xilinx
	/opt/xilinx/gnu/arm/lin/bin
	EOF
	
	if [-z "$(docker ps --all --quiet --filter "name=xmfw-
	    test-build")"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	xmfw-system build $name/test <<EOF
	xmfw-system:2.0.5-arm
	/opt/xilinx
	/opt/xilinx/gnu/arm/lin/bin
	EOF
	
	echo $name
	``

Postconds.:	• Menuconfig not prompted • Configuration loaded properly • Image xmfw-system:2.0.5-arm created • Image xmfw-system:2.0.5-arm has /opt/xm-install • xmfw-test-build container created • resident_sw generated in system worktree • resident_sw can be executed correctly
Result:	YES

TC-05							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V1	V1	V2	V1	–	V2	–	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**

–	–	–	–	–	–	–	–
Preconds.:	• Docker image `danoloan/xmfw:2.0.5-arm` available						

Actions:	`rm /tmp/config.tar.gz -f`
	`xmfw-install -m -c -o /tmp/config.tar.gz danoloan/xmfw:2.0.5-arm`
	`stat /tmp/config.tar.gz`						

Postconds.:	• Menuconfig prompted • `/tmp/config.tar.gz` created • Does not ask for toolchain • Does not build XtratuM						
Result:	YES						

TC-06							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V1	V1	V2	V2	–	V2	–	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• Docker image `danoloan/xmfw:2.6.0-ia32` available						

Actions:	`rm /tmp/config.tar.gz -f`
	`xmfw-install -m -c -o /tmp/config.tar.gz danoloan/xmfw:2.6.0-ia32`
	`stat /tmp/config.tar.gz`						

Postconds.:	• Menuconfig prompted • `/tmp/config.tar.gz` created • Does not ask for toolchain • Does not build XtratuM						
Result:	YES						

TC-07

I1	I2	I3	I4	I5	I6	I7	I8
V1	V1	V2	V3	–	V2	–	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• Docker image `danoloan/xmfw:2.0.5-arm-xalcxx` available						

Actions:	`rm /tmp/config.tar.gz -f`
	`xmfw-install -m -c -o /tmp/config.tar.gz danoloan/xmfw:2.0.5-arm-xalcxx`
	`stat /tmp/config.tar.gz`						

Postconds.:	• Menuconfig prompted • `/tmp/config.tar.gz` created • Does not ask for toolchain • Does not build XtratuM						
Result:	YES						

TC-08							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
N1	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Actions:	`xmfw-install -g danoloan/xmfw:2.0.5-arm`						
Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-09							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	N1	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–

Preconds.:	• `/nexist` is not a directory
Actions:	`xmfw-install -o /nexist/nexist danoloan/xmfw:2.0.5-arm`
Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-10							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	N1	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• `/nexist` is not a file						
Actions:	`xmfw-install -i /nexist danoloan/xmfw:2.0.5-arm`						
Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-11							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	N1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• `danoloan/xmfw:2.0.5-arm-nexist` is not an image in this computer						
Actions:	`xmfw-install danoloan/xmfw:2.0.5-arm-nexist`						

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-12							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	N2	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• `ubuntu:latest` pulled						
Actions:	`xmfw-install ubuntu:latest`						
Postconds.:	• Reports naming convention error • Does nothing else						
Result:	YES						

TC-13							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	N1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• `nexist:latest` is not an image in this computer						
Actions:	`xmfw-install danoloan/xmfw:2.0.5-arm nexist:latest`						
Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-14							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V1	V2	V1	V1	N1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Actions:	```xmfw-install -c danoloan/xmfw:2.0.5-arm```						
Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-15							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	N1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Actions:	```# Toolchain # /opt/next # /opt/next/bin xmfw-install danoloan/xmfw:2.0.5-arm```						
Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-16							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	N2	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Actions:	```# Toolchain # /opt/next # /usr/bin xmfw-install danoloan/xmfw:2.0.5-arm```						

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-17							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
N1	V1	V1	V1	V2	–	–	–
Actions:							

```
name=$(mktemp -d)
xmfw-system create danoloan/xmfw-system:2.0.5-arm-nexist
    $name/test hello-world
rmdir $name
```

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-18							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
N2	V1	V1	V1	V2	–	–	–
Actions:							

```
name=$(mktemp -d)
# xmfw:2.0.5-arm is not valid because it does not have
    the -system suffix
xmfw-system create danoloan/xmfw:2.0.5-arm $name/test
    hello-world
rmdir $name
```

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-19							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**

V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	N1	V1	V1	V2	–	–	–
Preconds.:	• The case must build XtratuM first						

Actions:	`# Toolchain`
	`# /opt/xilinx/`
	`# /opt/xilinx/gnu/arm/lin/bin/`
	
	`xmfw-install danoloan/xmfw:2.0.5-arm`
	
	`if [ -z "$(docker image ls --quiet --filter 'reference=xmfw-system:2.0.5-arm')" ]; then`
	`  echo 'error' >&2`
	`  exit 1`
	`fi`
	
	`xmfw-system create xmfw-system:2.0.5-arm /tmp hello-world <<EOF`
	`/opt/xilinx`
	`/opt/xilinx/gnu/arm/lin/bin`
	`EOF`							

Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-20							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	N2	V1	V1	V2	–	–	–
Preconds.:	• The case must build XtratuM first						

Actions:	```
	# Toolchain
	# /opt/xilinx/
	# /opt/xilinx/gnu/arm/lin/bin/
	
	xmfw-install danoloan/xmfw:2.0.5-arm
	
	if [-z "$(docker image ls --quiet --filter 'reference=
	    xmfw-system:2.0.5-arm')"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	xmfw-system create xmfw-system:2.0.5-arm /next/next hello
	    -world <<EOF
	/opt/xilinx
	/opt/xilinx/gnu/arm/lin/bin
	EOF
	``

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-21							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	V1	N1	V1	V2	–	–	–
Preconds.:	• The case must build XtratuM first						

Actions:	```
	# Toolchain
	# /opt/xilinx/
	# /opt/xilinx/gnu/arm/lin/bin/
	
	xmfw-install danoloan/xmfw:2.0.5-arm
	
	if [-z "$(docker image ls --quiet --filter 'reference=
	    xmfw-system:2.0.5-arm')"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	docker create --name xmfw-$name-build xmfw-system:2.0.5-
	    arm
	
	xmfw-system create xmfw-system:2.0.5-arm /next/next hello
	    -world <<EOF
	/opt/xilinx
	/opt/xilinx/gnu/arm/lin/bin
	EOF
	
	docker rm xmfw-$name-build
	``						

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-22							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	V1	V1	V1	N1	–	–	–
Preconds.:	• The case must build XtratuM first						

Actions:	`# Toolchain`
	`# /opt/xilinx/`
	`# /opt/xilinx/gnu/arm/lin/bin/`
	
	`name=$(mktemp -d)`
	
	`xmfw-install danoloan/xmfw:2.0.5-arm`
	
	`if [ -z "$(docker image ls --quiet --filter 'reference=xmfw-system:2.0.5-arm')" ]; then`
	`  echo 'error' >&2`
	`  exit 1`
	`fi`
	
	`docker rm -f xmfw-$name-build`
	
	`xmfw-system create xmfw-system:2.0.5-arm $name/test nex <<EOF`
	`/opt/xilinx`
	`/opt/xilinx/gnu/arm/lin/bin`
	`EOF`						

Postconds.:	• Reports error • Does nothing else						
Result:	YES						

TC-23							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**

V1	V1	V1	V1	V2	N1	V1	–
Actions:		`xmfw-system build /next`					
Postconds.:		• Reports error • Does nothing else					
Result:		YES					

TC-24							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	V1	V1	V1	V2	N2	V1	–
Actions:		`xmfw-system build /dev/null`					
Postconds.:		• Reports error • Does nothing else					
Result:		YES					

TC-25							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	V1	V1	V1	V2	V1	V2	N1
Preconds.:		• The case must build XtratuM first					

Actions:	```
	# Toolchain
	# /opt/xilinx/
	# /opt/xilinx/gnu/arm/lin/bin/
	
	name=$(mktemp -d)
	
	xmfw-install danoloan/xmfw:2.0.5-arm
	
	if [-z "$(docker image ls --quiet --filter 'reference=
	    xmfw-system:2.0.5-arm')"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	docker rm -f xmfw-$name-build
	
	xmfw-system create xmfw-system:2.0.5-arm $name/test hello
	    -world <<EOF
	/opt/xilinx
	/opt/xilinx/gnu/arm/lin/bin
	EOF
	
	if [-z "$(docker ps --all --quiet --filter "name=xmfw-
	    test-build")"]; then
	    echo 'error' >&2
	    exit 1
	fi
	
	docker rm xmfw-test-build
	
	xmfw-system build $name/test <<EOF
	nexist:latest
	/opt/xilinx
	/opt/xilinx/gnu/arm/lin/bin
	EOF
	
	echo $name
	``

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-26							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V2	V2	V2	V1	V1	V1	V1	V1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
V1	V1	V1	V1	V2	V1	V2	N2

Actions:	`# Toolchain`
	`# /opt/xilinx/`
	`# /opt/xilinx/gnu/arm/lin/bin/`
	
	`xmfw-system build test <<EOF`
	`xmfw:base`
	`/opt/xilinx`
	`/opt/xilinx/gnu/arm/lin/bin`
	`EOF`

Postconds.:	• Reports error • Does nothing else
Result:	YES

TC-27							
I1	**I2**	**I3**	**I4**	**I5**	**I6**	**I7**	**I8**
V1	V1	V2	V1	–	V2	–	N1
C1	**C2**	**C3**	**C4**	**C5**	**B1**	**B2**	**B3**
–	–	–	–	–	–	–	–
Preconds.:	• Docker image `danoloan/xmfw:2.0.5-arm` available						

Actions:	`xmfw-install`
	
	`xmfw-system create`
	
	`xmfw-system build`						

Postconds.:	• Configuration not supported is prompted						
Result:	YES						

Test case verification

All tests have been executed successfully, so not only the correctness of the implementation of separate components is verified, but also the integration among them. The framework is able of building and installing XtratuM, creating XtratuM based systems and building them ready to be deployed in hardware or in virtual machines.

6.1.2. Benchmarks

The benchmarks are pieces of software small enough to be verified by visual inspection. However, their integration with the rest of the system is essential. This integration will be verified in this section.

The `xmfw-system` script can be used to create applications from templates based on the examples — but examples can include benchmarks. Therefore, benchmarks must be available and valid to be used as templates of systems. To test this, we used this tool to set up the environment that runs the evaluation of the framework performed in Section 6.2.2 below. The framework was successfully used to perform this task, and it resulted easy and comfortable tu use.

6.1.3. XALC++

Example applications have been successfully created with the framework utilities and later executed in the Zedboard without errors. The example applications used to test this are included on Appendix **??**, and they are also included as templates in the `2.0.5-arm-xalcxx xmfw` build image.

The C memory management works as expected, verifying all the asserts in the code. The C++ example application also works as expected. The global constructor does not execute at initialization, but this was expected as explained in Section 5.5.2.

It is worth noting however that while in other systems would be segmentation fault, in XtratuM for ARM illegal memory accesses result sometimes in a trap handled by a silent halt. This halt is usually located at the address `<start> + 0x10`.

6.2. Validation

After verifying that the implemented components work as expected when designed, the system must be validated against the user and software requirements analyzed at the start of this work. In this section, we perform said validation and evaluate the framework using the implemented benchmarks.

6.2.1. Software requirements

In this section the different components are validated against the software requirements analysed in Section 3.5. This is performed using the traceability matrix shown in Table 4.1: we check if each component verifies the software requirements associated to it.

Framework manager (CMP-01) The component CMP-01 implemented by the `xmfw-install` script and the `xmfw:base`-based Docker images verifies the following software requirements:

- SR-FN-03: The `xmfw-install` script together with the `config` script integrated in the `xmfw:base` image are capable of successfully configure XtratuM, either with or without prompting a menu.
- SR-FN-04: The `-o` flag of the `xmfw-install` script successfully exports XtratuM configuration in a `.tar.gz` archive accessible for the developer.
- SR-FN-05: The `-i` flag of the `xmfw-install` script successfully imports XtratuM configuration from a `.tar.gz` archive.
- SR-FN-06: The `xmfw-install` script successfully install XtratuM in a new container image.
- SR-FN-07: The `xmfw-install` script together with the `build` script integrated in the `xmfw:base` image flag of the `xmfw-install` script successfully build XtratuM before installing it.
- SR-NF-01: XtratuM is included in `xmfw:base` and is the base of this whole component.
- SR-NF-09: XtratuM is built in `xmfw:base`-based containers, maintaining the host environment clean of updated software needed to do it.

System project manager (CMP-02) The component CMP-02 implemented by the `xmfw-system` script and its `create` and build commands verifies the following software requirements:

- SR-FN-08: The `create` command of the `xmfw-system` script successfully creates systems with default configuration. These systems are examples that can be built out-of-the-box with the `build` command.
- SR-FN-09: The `build` command of the `xmfw-system` script successfully builds a created system.
- SR-FN-13: The `<xmfw image>` parameter of the `xmfw-system` script allows specifying an image with XtratuM built for a specific architecture with the `-a` option of `xmfw-install`.
- SR-FN-17: Systems built for the Zynq7000 can be deployed in the Zedboard following the instructions detailed in the framework documentation.
- SR-FN-18: Systems built for the IA32 architecture can be deployed in a virtual machine following the instructions detailed in the framework documentation.

- SR-FN-20: `xmfw-system create` allows creating, building and deploying successfully benchmarks provided by the framework.
- SR-NF-10: The `xmfw:2.0.5-arm` and `xmfw:2.0.5-arm-xalcxx` allow building systems for the ARM-based Zynq7000 SoC. This has been further verified by running built systems with the Zedboard.
- SR-NF-11: The `xmfw:1.0.8-sparc` allows building systems for SPARC and LEON3FT [42].
- SR-NF-12: The `xmfw:2.6.0-ia32` allows building systems for IA32. The built systems can be run using QEMU (see Section 5.3.5).

Benchmarks (CMP-03) The component CMP-02 is implemented by the benchmarks developed for the `xmfw:2.0.5-arm` image. These benchmarks can be easily adapted for other target hardware, but verification and evaluation for ARM systems was higher priority in this project. This component verifies the following software requirements:

- SR-FN-21: The CPU stress isolation benchmark is implemented
- SR-FN-22: The memory stress isolation benchmark is implemented

XALC++ (CMP-04) The following requirements are validated in this component:

- SR-FN-01: `malloc` function works as defined by the POSIX standard.
- SR-FN-02: `free` function works as defined by the POSIX standard.
- SR-FN-27: The `new` operator works as defined in the C++ standard.
- SR-FN-28: The `delete` operator works as defined in the C++ standard.
- SR-NF-08: The `Ref` application can be built with XtratuM with undefined symbols as explained in Section 5.3.6

6.2.2. Evaluation of the framework

Among all the user requirements, the most important are arguably UR-CA-14 and UR-CA-15. As studied in the State of the Art, hard-real time systems needed in nanosatellite applications are part of what makes nanosatellite flight software that much more complex than software in other fields. In this section we study the viability of using software produced with this framework in systems where fulfillment of time frames is crucial.

Platform and method

To perform the study we have used the Zedboard, a development board for the Zynq7000 SoC. This has been considered the best option because the ARM version of XtratuM v2.0.5 comes with extensive documentation specific for this board on how to deploy systems.

The benchmarks designed in Section 4.4 and implemented in Section 5.4 were used to perform the evaluation. For this analysis we chose executing 1024 iterations of the full cycle, and the scheduling cycle designed in 5.4 was used with values of 10, 100, 250, 500 and 1000 μs for the periods of the cycles of partitions. As explained in Section 5.4, the interrupt handler executes in 4-6 μs, so it would not be sensible to try any period under 10 μs.

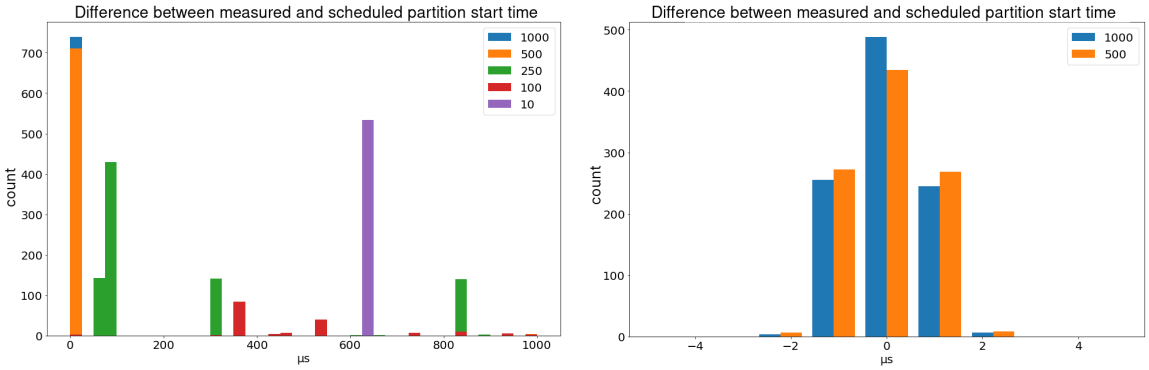
Once configured and built, benchmarks were deployed following the instructions detailed in Section 5.3.5. The output was stored in a computer to be later analyzed.

Benchmark results

XtratuM for the Zynq7000 SoC on the Zedboard has proven suitable for applications that require periods of no less than 500μs for their tasks. Figures 6.1 and 6.2 show the distribution of the deviations in time between cycles of a partition for the test cases described in the previous section, the ideal being a deviation of 0 or less.

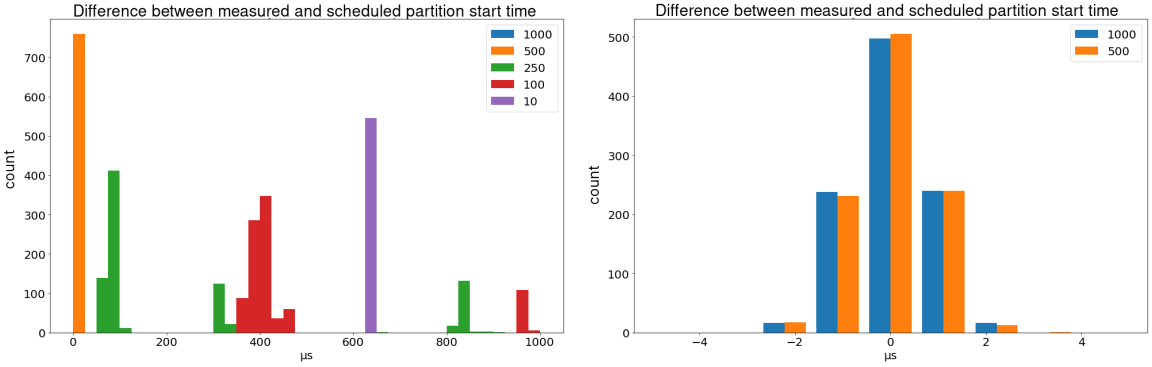
With these criteria, the framework performs far from well for periods less than 250 μs, while for periods from 500μs onwards it works as expected, producing deviations of the order of the unit of measurement in a normal distribution. The ideal would be that the deviation always be negative, but these results are still acceptable for the majority of use cases.

Applications that fit he profile suited for the framework include complex systems with many inner applications, that may be based in component-based frameworks like F' or in full task-based operating systems. This framework can be used to virtualize these systems and allow them to share hardware, while maintaining the ability to verify the fulfillment of the time frames of tasks. This can also be a good start to support them in multi-core systems.



(a) All cases. (b) 500 and 1000 μs cases only.

Fig. 6.1. Results of the CPU benchmark (FPU mode).



(a) All cases. (b) 500 and 1000 μs cases only.

Fig. 6.2. Results of the memory benchmark.

7. PLAN AND BUDGET

7.1. Planning

The planning of the project was very tight. The client requested that the solution be available by June, but due to several factors the development of the project was delayed. This let a small time frame to perform all activities in a rigorous manner.

The V model is the most widely used in the context of space software development and critical systems engineering. Due to its robustness in producing verified systems, it has been chosen by many space agencies and governments to produce critical software [54]. Because of this, the V model has been deemed the most adequate for the development of this project. This work is not too complex, but the need for correctness in this field should not be disregarded. However, we have seen appropriate to simplify the process of the model to better fit the reality of the project.

Figure 7.1 shows the simplified V model followed. First, the analysis of requirements was performed, followed by the specific component design of the system. Next, the correctness of the implementation provided was verified against the components design. Finally, the global results were compared with the initial requirements. It was crucial that in every step previous phases were affected by the current as far as possible. Also, phases were overlapped when needed. This flexibility and continuous feedback allowed the project to develop in a quasi-agile way, without giving back on the rigorousness of the V model.

Figure 7.2 shows the time distribution of the development in a modified version of the Gantt diagram, fit for the V model. Due to the reduced time-frame, some tasks that could be parallelized needed to be overlapped, but the process still remains true to the initially devised model. Processes are represented as candles: the wax of the candle indicates the main phase in a time frame, while strands represent tasks being executed as secondary.

At first, the analysis of the state of the art was performed, knowing the problem of the client and looking for potential approaches to the solution. Because this field was new us, attempts at the implementation were started right at the time of the analysis, with the hope that this helped gaining insight into the possibilities and problems potentially to be encountered.

7.2. Budget

In this section the project is put price, considering the different costs that took place in its development and the profit margins expected.

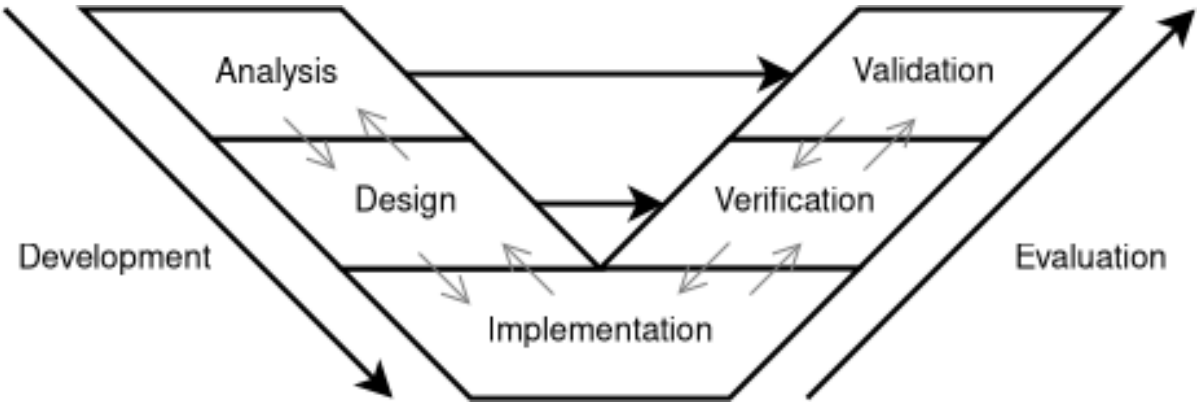


Fig. 7.1. Simplified V development model.

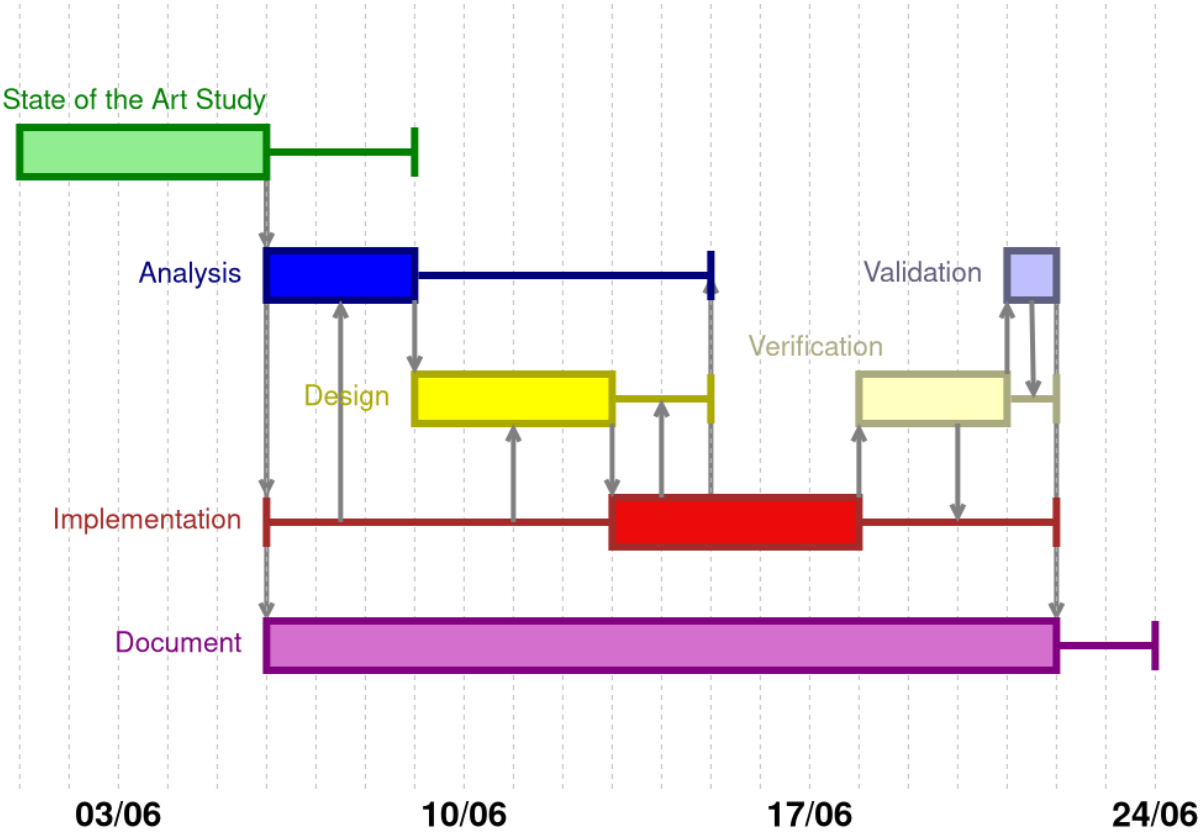


Fig. 7.2. Gantt-like diagram of the planning adapted to the V development model.

7.2.1. Manpower

The duration of the project was rapid, being completed in under a moth, but with the drawback of needing high-intensity working sessions of around 12h a day. Although our team consists of a single junior engineer, the work performed has been divided into the different phases of the project, assigning different cost to each of them according to the equivalent role performed. The following table shows the breakdown of manpower costs:

Phase	days	hours p. day	cost p. hour	**Subtotal**
State of the Art Study (Researcher)	5	6	30.00€	900.00€
Analysis (Analyst)	3	12	45.00€	1,620.00€
Design (Engineer)	4	12	38.00€	1,824.00€
Implementation (Developer)	5	12	30.00€	1,800.00€
Verification (Engineer)	3	12	38.00€	1,368.00€
Validation (Analyst)	1	12	45.00€	540.00€
			TOTAL	8,052.00€

7.2.2. Hardware

The needed hardware material for the project is listed below:

Concept	Cost
Desktop computer	
• Custom built CPU	600.00€
• Acer G226HQL monitor	130.00€
• Packard bell KB-0420 keyboard	10.00€
Thinkpad Yoga 260 (Used)	270.00€
Zedboard	378.50€
USB Type A to micro USB cable x2	10.00€
TOTAL	1,398.50€

The total cost of hardware ascends to 1,398.50€. Considering a depreciation period of 5 years (1,825 days) for all the hardware, the costs would be as described below:

	1,398.50	€
÷	1,825	days
	0.771	€ per day
×	21	days
	16.09	€ total

7.2.3. Software

Here follows a list of software used during the development of the project:

- Neovim — text editing
- Void Linux and Artix Linux — operating systems
- Dia — diagram editor
- TexLive — document formatting
- Docker — container manager
- GNU Xilinx Toolchain — hardware support
- XtratuM — hypervisor
- F', cFS — application frameworks
- Xilinx SDK — provides XMD (connection to hardware)

All these tools but the Xilinx SDK are free and open source software and are available for no cost. The Xilinx SDK has a proprietary license, but this license can be obtained also free of cost for the components used (Vitis).

Therefore, the total software costs of the development of this project ascend to $0.00€$.

7.2.4. Others

Other costs include fungible material and transport needed to perform the job. Since the past pandemic situation working from home has been the standard for software development, so no cost in transport further from taking away the Zedboard from University has been needed.

Concept	Cost
Fungible material	
• Paper	8.00€
• Pens	2.00€
• Paper prints	10.00€
Transport	20.00€
TOTAL	40.00€

In total other costs ascend to $40.00€$.

7.2.5. Final budget

The risk of associated costs surpassing the planned is fairly high considering the tightness of the project; also, high-performance demands from the client make it reasonable to ask for high profit margins. To calculate the final price of the project, the total cost is calculated and risks and profit margins are added accordingly. In total, the final balance is as follows:

Manpower costs	8,052.00€
Hardware costs	16.09€
Software costs	0.00€
Others	40.00€
Subtotal	8,108.09€
Risk (15%)	1,216.21€
Profit (50%)	4,054.05€
TOTAL	**13,378.35€**
VAT (21%)	2,809.45€
TOTAL (plus VAT)	**16,187.81€**

The total final price for the development of this project will then be of **THIRTEEN THOUSANDS THREE HUNDREDS AND SEVENTY EIGHT EURO WITH THIRTY FIVE EURO CENTS** without VAT included, and **SIXTEEN THOUSANDS ONE HUNDRED AND EIGHTY SEVEN EURO WITH EIGHTY ONE EURO CENTS** with VAT included.

8. LEGAL FRAMEWORK AND SOCIAL IMPACT

This work will aid in the development of In this section we discuss the current state of law, standards and societal impact in the field of space exploration. The main focus will rely on the european context, taking special consideration to the standards and actions of the european space agencies, mainly the ESA.

8.1. Law

Since the launch of the first artificial satellite, the Sputnik I, the need of laws for Space exploration and exploitation has been made apparent. Thus, Space Law was created as a new branch of international Law. Just like Maritime Law regulate the seas and Air Law the airspaces of nations, Space Law regulates how nations, agencies and private entrepreneurs may explore space and what they can and cannot do.

In Europe, the ESA leads efforts for the standardization and regulation of space. ESA's main focus has been on enabling the access to Space to the most agents possible, all while promoting cooperation between them. As they explain in their website, "the reason for laws on space is not just to facilitate access to, and the use of, space, but to favour cooperation between all nations, not only between space powers"[55]. In 1989, the European Centre for Space Law was established by the ESA with the objective of fomenting discourse and development of Space Law. Since then, hot topics like space exploitation, the satellite space and the space debris situation have been addressed with the hope of finding proper legal solutions [56].

Outside of Europe, the history of Space Law has been one of cooperation since the end of the Cold War. International treaties and projects like the International Space Station have coordinated the efforts of exploring and coordinating space. However, the situation has been more complex as of lately, mainly due to tensions between space powers and the appearance of new private actors such as SpaceX or Blue Origin in the Space scene. These situations show that Space Law is a rapidly changing field, and no comprehensive consensus has been reached yet.

However, general guidelines are followed by the majority of actors in Space. An example of this is the Committee on the Peaceful Uses of Outer Space — with 95 members as of 2021, the Committee coordinates efforts for the exploration and the use of Space. Part of the work of the COPUOS is the Space Law Treaties and Principles. These include five Treaties for Outer Space, as well as five Declarations of Legal Principles. These are [57]:

Treaties for Outer Space

- The "Treaty on Principles Governing the Activities of States in the Exploration and Use of Outer Space, including the Moon and Other Celestial Bodies"

- The "Agreement on the Rescue of Astronauts, the Return of Astronauts and the Return of Objects Launched into Outer Space"
- The "Convention on International Liability for Damage Caused by Space Objects"
- The "Convention on Registration of Objects Launched into Outer Space"
- The "Agreement Governing the Activities of States on the Moon and Other Celestial Bodies"

Legal Principles

- The "Declaration of Legal Principles Governing the Activities of States in the Exploration and Uses of Outer Space"
- "The Principles Governing the Use by States of Artificial Earth Satellites for International Direct Television Broadcasting"
- "The Principles Relating to Remote Sensing of the Earth from Outer Space"
- "The Principles Relevant to the Use of Nuclear Power Sources in Outer Space"
- "The Declaration on International Cooperation in the Exploration and Use of Outer Space for the Benefit and in the Interest of All States, Taking into Particular Account the Needs of Developing Countries"

8.1.1. XtratuM licensing

As explained in Section 2.5, XtratuM is licensed under the GPL software. This makes it a legal requirement to distribute and make available all modifications to its codebase. Modifications performed in this work have been published to publicly available GitHub repositories [43], [58], [59]. urthermore, all the framework scripts and Dockerfiles have also been licensed under the GPLv3 license and made publicly available in GitHub [60].

8.2. Standards

Standardization is essential for cooperation. This is why the ESA, a Space Agency fruit of cooperation, has lead the effort of standardizing space exploration. The European Cooperation for Space Standardization board oversees standards that must be applied by the ESA and their collaborators. Among these, standards for Space Software Engineering are provided, like the *ECSS-E-ST-40C – Software* and the *ECSS-Q-ST-80C Rev.1 – Software product assurance* [61], [62]. Future work could further consider these documents and provide means of automating some of the tasks specified in them, easing the compliance of nanosatellite developers to the standards.

While the CubeSat standard discussed in Section 2.1.2 is very important when designing the hardware characteristics of a nanosatellite, it provides little specification for the payload and the software of the craft, so it is not relevant for our work.

8.3. Social and economic impact

With a global turnover of almost half a billion dollars [63], the Space exploration is a very providing field, employing thousands of people around the world. In Europe, the openness of the ESA has provided the grounds for an ecosystem of enterprises and engineering businesses to flourish collaborating with their missions. The client SENER is an example of this.

ESA missions have proven useful for society in many aspects. They include many ways in which our lives are bettered through the use of Space, like Metereological observation [64], Telecommunication services [65] and Global Navigation Satellite Systems like Gallileo [66].

The ESA is present in many nations, Spain among them. The institute dedicated to space exploration in Spain is the Instituto Nacional de Técnica Aeroespacial. The INTA has collaborated with the ESA in many missions, like the development of the Rover Environmental Monitoring Station in the Curiosity rover [67] or the Mars Environmental Dynamics Analyzer in the recently launched Perseverance rover [68]. SENER has collaborated with the INTA for jobs like such. Also, the INTA provides services for many engineering businesses, like jet engine benchmarking and electromagnetic device standard compliance [69].

Just as with any part of the creation, space must also be taken care of. The increment in satellite and nanosatellite launches have provoked the undesirable situation of the launch debris being an impediment for future launches. In this regard, the ESA coordinates efforts on possible solutions for this problem, like a Sustainability rating for satellite launches [70].

9. CONCLUSIONS AND FUTURE WORK

This project has produced a framework capable of building systems based on the XtratuM hypervisor. In this section we review whether this work has achieved the objectives stated in the introduction:

- The framework provides support for developing hard-real time applications on top of the XtratuM Hypervisor
- The framework automates most of the tasks common for developing XtratuM-based applications
- The framework abstracts the development environment from the XtratuM build environment through the use of containers
- The framework provides support on top of which future work regarding the use of the XtratuM hypervisor on ARM for nanosatellite application development can be based.
- This work has documented and solved many common problems faced when deploying and developing XtratuM applications for the Zynq7000 SoC
- The XtratuM hypervisor has been evaluated when running in the ARM-based Zynq7000 SoC and the framework provides default benchmarks

Overall, XtratuM has proved being a useful tool to develop real-time applications, among which nanosatellite applications can be included. Thanks to this work, it now supports C++ development and memory management, and it is closer to supporting complex frameworks used in space exploration like F' or cFS.

9.1. Future work

Future lines of work that could continue the research performed in this project could include:

- Provide an implementation that completely covers the designed system
- Standardize the organization of the files in a system working tree to ease portability of applications
- Automate the management of XtratuM applications/partitions inside systems, through a tool that manages both the XtratuM XML configuration and the Makefiles.

- Provide support for external application frameworks like cFS and F'
- Automate the integration of said external application frameworks in systems
- Automate the deployment of systems in hardware
- Extend C++ support for XtratuM to further cover the standard
- Optimize the memory management library implemented and extend it for more general use cases

9.2. Personal conclusions

This project has been a very good opportunity to learn about many things that not only are related to this degree, but also that I enjoy.

XtratuM has been real fun exploring and adapting. The documentation, despite being scarce and sometimes cumbersome, is really comprehensive and well-written. This piece of software is beautiful in its simplicity, but very capable and extensible. XtratuM truly deserves more attention from researchers and engineers around the world.

Expanding XtratuM to support C++ has been a real adventure, too, with each thing I thought I knew being questioned by the absence of support like standard libraries. I have learned a lot about the inner workings of compilers, linkers and loaders, as well as how to use new build systems that I have not had the opportunity to use that much, such as CMake.

Also, it has been very revealing seeing the power of systematic approaches to software development applied to a project like this. Without such a planned and rigorous process, the end result very likely would not have end up with a quality like it has now.

What I like the most about this project has been the opportunity of developing a system entirely based on free software. Except the Xilinx SDK, all software used, including XtratuM, has been licensed under free and open source software that respect the freedom of their users. I think this should not be underrated, and that more projects should chose the path of XtratuM and licensing their products with licenses like the GPL and the like.

XtratuM has so much potential, and it could be the center of its own ecosystem. I really hope that this study about it serves as inspiration for future work, and a stepping stone for the University, SENER and the community to start using it more and build great and wonderful things with it.

BIBLIOGRAPHY

[1] Defense Industry Daily staff. ‘ Small Is Beautiful: US Military Explores Use of Microsatellites.’ (30th Jun. 2011), [Online]. Available: https://www.defenseindustrydaily.com/Small-Is-Beautiful-US-Military-Explores-Use-of-Microsatellites-06720/ (visited on 29/05/2021).

[2] *What is a CubeSat & other picosatellites*. [Online]. Available: https://www.nanosats.eu/cubesat (visited on 30/05/2021).

[3] Wikipedia, the Free Encyclopedia, Ed. ‘ CubeSat.’ (), [Online]. Available: https://en.wikipedia.org/wiki/CubeSat (visited on 15/06/2021).

[4] *Nanosats Database – Figures*. [Online]. Available: https://www.nanosats.eu/#figures (visited on 30/05/2021).

[5] *The CubeSat Program*. [Online]. Available: https://www.cubesat.org/about (visited on 30/05/2021).

[6] *CubeSat Design Specification*, version 13th Revision, 6th Apr. 2015. [Online]. Available: https://www.cubesat.org/s/cds_rev13_final2.pdf.

[7] ‘ State of the Art of Small Spacecraft Technology — Flight Software,’ NASA Ames Research Center, Small Spacecraft Systems Virtual Institute, Report NASA/TP—2020–5008734, Oct. 2021, pp. 178–185. [Online]. Available: https://www.nasa.gov/sites/default/files/atoms/files/2020soa_final.pdf.

[8] *The SPARC Architecture Manual*, Revision SAV080SI9308, version 8, pp. 1–7. [Online]. Available: https://sparc.org/wp-content/uploads/2014/01/v8.pdf.gz (visited on 04/06/2021).

[9] *SPARC Architecture License*. [Online]. Available: https://sparc.org/technical-documents/#ArchLic (visited on 04/06/2021).

[10] J. Andersson, J. Gaisler and R. Weigand, ‘Next Generation Multi-Purpose Microprocessor,’ *DAta System In Aerospace (DASIA)*, 2010. [Online]. Available: https://amstel.estec.esa.int/tecedm/website/biblio/AnderssonDASIA2010.pdf.

[11] E. S. Agency. ‘Leading up to LEON: ESA’s first microprocessors.’ (7th Jan. 2013), [Online]. Available: http://www.esa.int/Enabling_Support/Space_Engineering_Technology/Leading_up_to_LEON_ESA_s_first_microprocessors (visited on 04/06/2021).

[12] *Cobham Gaisler*. [Online]. Available: https://www.gaisler.com/ (visited on 04/06/2021).

[13] *LEON5*. [Online]. Available: https://www.gaisler.com/index.php/products/processors/leon5 (visited on 04/06/2021).

[14] *ARM — Architecture*. [Online]. Available: https://www.arm.com/why-arm/architecture (visited on 04/06/2021).

[15] *ARM — Company*. [Online]. Available: https://www.arm.com/company (visited on 04/06/2021).

[16] R. Aitken. ' Radiation-Hardened Arm Chips Aim for the Stars.' (27th Feb. 2020), [Online]. Available: https://www.arm.com/blogs/blueprint/arm-rad-hard-chips-space (visited on 04/06/2021).

[17] *ARM — CPU Architecture*. [Online]. Available: https://www.arm.com/why-arm/architecture/cpu (visited on 04/06/2021).

[18] ' State of the Art of Small Spacecraft Technology — Command and Data Handling,' NASA Ames Research Center, Small Spacecraft Systems Virtual Institute, Report NASA/TP—2020–5008734, Oct. 2021, pp. 166–175. [Online]. Available: https://www.nasa.gov/sites/default/files/atoms/files/2020soa_final.pdf.

[19] *GR712RC Board*. [Online]. Available: https://www.gaisler.com/index.php/products/boards/gr712rc-board (visited on 15/06/2021).

[20] *Zedboard*. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html (visited on 15/06/2021).

[21] *Rasbperry Pi 1 Model B+*. [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-1-model-b-plus/ (visited on 15/06/2021).

[22] *FreeRTOS Kernel Ports*. [Online]. Available: https://wwww.freertos.org/RTOS_ports.html (visited on 05/06/2021).

[23] *VxWorks Product Overview*. [Online]. Available: https://bit.ly/2UBvsuF (visited on 05/06/2021).

[24] E. Brown. ' NASA's Martian helicopter runs Linux.' (22nd Feb. 2021), [Online]. Available: https://www.theverge.com/2021/2/19/22291324/linux-perseverance-mars-curiosity-ingenuity (visited on 05/06/2021).

[25] *RTEMS License Information*. [Online]. Available: https://www.rtems.org/license (visited on 05/06/2021).

[26] *License Details*. [Online]. Available: https://wwww.freertos.org/a00114.html (visited on 05/06/2021).

[27] *Frequently asked questions*. [Online]. Available: https://www.kernel.org/category/faq.html (visited on 05/06/2021).

[28] NASA, *F'*, 2020. [Online]. Available: https://nasa.github.io/fprime/ (visited on 07/06/2021).

[29] ——, *F' Features*, 2020. [Online]. Available: https://nasa.github.io/fprime/features.html (visited on 07/06/2021).

[30] ——, *nasa/fprime*. [Online]. Available: https://github.com/nasa/fprime (visited on 07/06/2021).

[31] *OpenSatKit — About*. [Online]. Available: https://opensatkit.github.io/menu/about.html (visited on 07/06/2021).

[32] ——, *cFS*. [Online]. Available: https://cfs.gsfc.nasa.gov/ (visited on 07/06/2021).

[33] ——, *cFS*. [Online]. Available: https://github.com/nasa/cFS (visited on 07/06/2021).

[34] E. Stoneking, *42: A General-Purpose Spacecraft Simulation*, NASA Software Designation GSC-16720-1, 2010–2019. [Online]. Available: https://sourceforge.net/projects/fortytwospacecraftsimulation, %20https://github.com/ericstoneking/42.

[35] *COSMOS Docs*, 24th Oct. 2020. [Online]. Available: https://cosmosc2.com/assets/COSMOS_Docs_10_24_2018.pdf (visited on 07/06/2021).

[36] B. Aerospace, *COSMOS*, 24th Oct. 2020. [Online]. Available: https://github.com/BallAerospace/COSMOS (visited on 07/06/2021).

[37] *Celestia Screenshot Gallery*. [Online]. Available: https://celestia.space/gallery.html (visited on 07/06/2021).

[38] *Celestia*. [Online]. Available: https://github.com/CelestiaProject/Celestia (visited on 07/06/2021).

[39] *Oracle® VM User's Guide*, E18549-08, version 3.0.3, pp. 1–2. [Online]. Available: https://docs.oracle.com/cd/E26996_01/E18549/E18549.pdf (visited on 04/06/2021).

[40] A. Crespo, I. Ripoll, M. Masmano, P. Arberet and M. Jean-Jacques, 'XtratuM: An Open Source Hypervisor for TSP Embedded Systems in Aerospace,' May 2009.

[41] *Software User Manual – XM-ARM*, version 14-035-03.005.sum.04, Jul. 2017.

[42] *XtratuM Hypervisor for LEON3*, version xm-3-usermanual-022c, Feb. 2011.

[43] D. Alcaide Nombela and fentISS, *XtratuM IA32*, 2021. [Online]. Available: https://github.com/danoloan10/xm-ia32.

[44] Docker, *Docker Overview*. [Online]. Available: https://docs.docker.com/get-started/overview/ (visited on 21/06/2021).

[45] ——, *Docker Overview*. [Online]. Available: https://docs.docker.com/engine/reference/builder/ (visited on 21/06/2021).

[46] *Podman*. [Online]. Available: https://podman.io/ (visited on 21/06/2020).

[47] *Sysbox*. [Online]. Available: https://github.com/nestybox/sysbox (visited on 21/06/2020).

[48] Github, *No License*. [Online]. Available: https://choosealicense.com/no-permission/ (visited on 08/06/2021).

[49] *What is Free Software?* [Online]. Available: https://www.gnu.org/philosophy/free-sw.en.html (visited on 10/06/2021).

[50] ——, *No License*. [Online]. Available: https://choosealicense.com/appendix/ (visited on 08/06/2021).

[51] Wikipedia, *Coupling (computer science)*, 1st Jun. 2021. [Online]. Available: https://en.wikipedia.org/wiki/Coupling_(computer_programming) (visited on 20/06/2021).

[52] ——, *Cohesion (computer science)*, 10th Jun. 2021. [Online]. Available: https://en.wikipedia.org/wiki/Cohesion_(computer_science) (visited on 20/06/2021).

[53] D. Alcaide Nombela and fentISS, *XtratuM ARM*, 2021. [Online]. Available: https://github.com/danoloan10/fprime.

[54] Wikipedia, *Coupling (computer science)*, 1st Jun. 2021. [Online]. Available: https://en.wikipedia.org/wiki/V-Model (visited on 20/06/2021).

[55] *About space law*. [Online]. Available: https://www.esa.int/About_Us/ECSL_-_European_Centre_for_Space_Law/About_space_law (visited on 08/06/2021).

[56] *European Centre for Space Law*. [Online]. Available: https://www.esa.int/About_Us/ECSL_-_European_Centre_for_Space_Law/About_ECSL (visited on 08/06/2021).

[57] *Space Law Treaties and Principles*, United Nations – Ofice for Outer Space Affairs, 2021. [Online]. Available: https://www.unoosa.org/oosa/en/ourwork/spacelaw/treaties.html.

[58] D. Alcaide Nombela and fentISS, *XtratuM ARM*, 2021. [Online]. Available: https://github.com/danoloan10/xm-arm.

[59] ——, *XtratuM SPARC*, 2021. [Online]. Available: https://github.com/danoloan10/xm-sparc.

[60] D. Alcaide Nombela, *XtratuM Framework*, 2021. [Online]. Available: https://github.com/danoloan10/xmfw.

[61] ECSS, 'Software,' European Cooperation for Space Standardization, ECSS ECSS-E-ST-40C, 6th Mar. 2009. [Online]. Available: https://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/.

[62] ——, 'Software product assurance,' European Cooperation for Space Standardization, ECSS ECSS-Q-ST-80C, version Rev.1, 15th Feb. 2017. [Online]. Available: https://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/.

[63] *Space industry worldwide - statistics & facts*, 2021. [Online]. Available: https://www.statista.com/topics/5049/space-exploration/.

[64] *Metereological missions*, 2021. [Online]. Available: https://www.esa.int/Applications/Observing_the_Earth/Meteorological_missions.

[65] *Telecommunications Integrated Applications*, 2021. [Online]. Available: https://www.esa.int/Applications/Telecommunications_Integrated_Applications.

[66] *Galileo and EGNOS*, 2021. [Online]. Available: https://www.esa.int/Applications/Navigation/Galileo_and_EGNOS.

[67] *Curiosity*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Curiosity_(rover).

[68] *Perseverence*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Curiosity_(rover).

[69] *INTA — Servicios*, 2021. [Online]. Available: https://www.inta.es/INTA/es/servicios/#.

[70] *Space sustainability rating to shine light on debris problem*, 2021. [Online]. Available: https://www.esa.int/Safety_Security/Space_Debris/Space_sustainability_rating_to_shine_light_on_debris_problem.

A. CPU BENCHMARK

CODE LISTING A.1. CPU benchmark partition

```
#include <stdio.h>
#include <xm.h>
#include <irqs.h>

#define LOG(...) do { \
    xmTime_t hw; \
    XM_get_time(XM_HW_CLOCK, &hw); \
    printf("[P%d] [%d] - ", XM_PARTITION_SELF, hw); \
    printf(__VA_ARGS__); \
    printf("\n"); \
} while (0)

#ifdef CONFIG_MMU
#define N_ITERATIONS 1024
#define SHARED_ADDRESS  0x1C000000

xm_u32_t *sem = (xm_u32_t *) SHARED_ADDRESS;

xm_u32_t it = 0;
xmTime_t hw[N_ITERATIONS];

#else
#error MMU not enabled
#endif


void CycleStartHandler(trapCtxt_t *ctxt)
{
    if (it < N_ITERATIONS) {
        XM_get_time(XM_HW_CLOCK, (hw + it));
        it++;
    } else {

        if (*sem != XM_PARTITION_SELF)
            return;

        LOG("Start times:\n");

        for (; it > 0; it--) {
            printf("\t[ %d:%d ]\n",
                    XM_PARTITION_SELF,
                    *(hw + N_ITERATIONS-it)
                );
        }
        printf("\n");
```

```
46
47          LOG("Halting...\n");
48
49          *sem = *sem + 1;
50
51          XM_halt_partition(XM_PARTITION_SELF);
52      }
53  }
54
55  void PartitionMain(void)
56  {
57  #ifdef CONFIG_ARM
58      InstallIrqHandler(XAL_XMEXT_TRAP(XM_VT_EXT_CYCLIC_SLOT_START),
           CycleStartHandler);
59  #else
60  #   error Architecture not supported
61  #endif
62
63      HwSti();
64
65      XM_clear_irqmask(0, (1<<XM_VT_EXT_CYCLIC_SLOT_START));
66
67  #ifdef FPU_MODE
68      register float i = 2.4356;
69      LOG("FPU mode enabled");
70  #else
71      register int i = 0;
72  #endif
73
74      *sem = 0;
75
76      while(1) {
77
78  #ifdef FPU_MODE
79          i = i * 0.1;
80          i = i / 0.1;
81  #else
82          i++;
83          i--;
84  #endif
85
86      }
87  }
```

CODE LISTING A.2. CPU benchmark configuraton

```
<SystemDescription xmlns="http://www.xtratum.org/xm-arm-2.x" version
   ="1.0.0" name="timers">
    <HwDescription>
        <MemoryLayout>
            <Region type="rom" start="0x0" size="1MB" />
            <Region type="sdram" start="0x00100000" size="1023MB" />
        </MemoryLayout>
        <ProcessorTable>
            <Processor id="0" frequency="667Mhz">
                <CyclicPlanTable>
                    <Plan id="0" majorFrame="1000us">
                      <Slot id="0" start="0000us" duration="0500us"
                         partitionId="0" />
                      <Slot id="1" start="0500us" duration="0500us"
                         partitionId="1" />
                    </Plan>
                </CyclicPlanTable>
            </Processor>
        </ProcessorTable>
        <Devices>
            <Uart id="1" baudRate="115200" name="Uart" />
        </Devices>
    </HwDescription>

    <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB" />
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="looper" flags="boot" console="Uart">
            <PhysicalMemoryAreas>
                <Area start="0x10000000" size="512KB" />
                <Area start="0x1C000000" size="128KB" />
            </PhysicalMemoryAreas>
        </Partition>
        <Partition id="1" name="logger" flags="boot" console="Uart">
            <PhysicalMemoryAreas>
                <Area start="0x14000000" size="512KB" />
                <Area start="0x1C000000" size="128KB" />
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>
</SystemDescription>
```

B. MEMORY BENCHMARK

CODE LISTING B.1. Memory benchmark partition

```
#include <stdio.h>
#include <xm.h>
#include <irqs.h>

#define LOG(...) do { \
    xmTime_t hw; \
    XM_get_time(XM_HW_CLOCK, &hw); \
    printf("[P%d] [%d] - ", XM_PARTITION_SELF, hw); \
    printf(__VA_ARGS__); \
    printf("\n"); \
} while (0)

#ifdef CONFIG_MMU
#define N_ITERATIONS 1024
#define SHARED_ADDRESS  0x1C000000

xm_u32_t *sem = (xm_u32_t *)  SHARED_ADDRESS;
xm_u32_t *pno = (xm_u32_t *) (SHARED_ADDRESS + 0x100);

xm_u32_t it = 0;
xmTime_t hw[N_ITERATIONS];
xm_u32_t pr[N_ITERATIONS];

#else
#error MMU not enabled
#endif

void CycleStartHandler(trapCtxt_t *ctxt)
{
    if (it < N_ITERATIONS) {
        pr[it] = *pno;
        XM_get_time(XM_HW_CLOCK, (hw + it));
        it++;
    } else {

        if (*sem != XM_PARTITION_SELF)
            return;

        LOG("Start times:\n");

        for (; it > 0; it--) {
            printf("\t[ %d:%d, %d ]\n",
                    XM_PARTITION_SELF,
                    hw[N_ITERATIONS-it],
                    pr[N_ITERATIONS-it]
```

```
                );
        }
        printf("\n");

        LOG("Halting...\n");

        *sem = *sem + 1;

        XM_halt_partition(XM_PARTITION_SELF);
    }
}

void PartitionMain(void)
{
#ifdef CONFIG_ARM
    InstallIrqHandler(XAL_XMEXT_TRAP(XM_VT_EXT_CYCLIC_SLOT_START),
       CycleStartHandler);
#else
#   error Architecture not supported
#endif

    HwSti();

    XM_clear_irqmask(0, (1<<XM_VT_EXT_CYCLIC_SLOT_START));

    *sem = 0;

    while(1) {
        *pno = XM_PARTITION_SELF;
    }
}
```

CODE LISTING B.2. Memory benchmark configuraton

```
<SystemDescription xmlns="http://www.xtratum.org/xm-arm-2.x" version
   ="1.0.0" name="timers">
    <HwDescription>
        <MemoryLayout>
            <Region type="rom" start="0x0" size="1MB" />
            <Region type="sdram" start="0x00100000" size="1023MB" />
        </MemoryLayout>
        <ProcessorTable>
            <Processor id="0" frequency="667Mhz">
                <CyclicPlanTable>
                    <Plan id="0" majorFrame="1000us">
                      <Slot id="0" start="0000us" duration="0500us"
                         partitionId="0" />
                      <Slot id="1" start="0500us" duration="0500us"
                         partitionId="1" />
                    </Plan>
                </CyclicPlanTable>
            </Processor>
```

```
        </ProcessorTable>
        <Devices>
            <Uart id="1" baudRate="115200" name="Uart" />
        </Devices>
    </HwDescription>

    <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB" />
    </XMHypervisor>

    <PartitionTable>
        <Partition id="0" name="looper" flags="boot" console="Uart">
            <PhysicalMemoryAreas>
                <Area start="0x10000000" size="512KB" />
                <Area start="0x1C000000" size="128KB" />
            </PhysicalMemoryAreas>
        </Partition>
        <Partition id="1" name="logger" flags="boot" console="Uart">
            <PhysicalMemoryAreas>
                <Area start="0x14000000" size="512KB" />
                <Area start="0x1C000000" size="128KB" />
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>
</SystemDescription>
```

C. CMAKE FILES FOR F'

CODE LISTING C.1. Toolchain configuration

```
####
# XtratuM for Xilinx Zynq7000
####

set(CMAKE_SYSTEM_NAME "Xtratum")

set(CMAKE_C_COMPILER "/opt/toolchain/gnu/arm/lin/bin/arm-xilinx-eabi-gcc")
set(CMAKE_CXX_COMPILER "/opt/toolchain/gnu/arm/lin/bin/arm-xilinx-eabi-g++")

set(CMAKE_LINKER "/opt/toolchain/gnu/arm/lin/bin/arm-xilinx-eabi-ld")

# Order matters here
set(CMAKE_C_LINK_EXECUTABLE
    "<CMAKE_LINKER> -o <TARGET> <OBJECTS> <LINK_LIBRARIES>   <CMAKE_C_LINK_FLAGS> <LINK_FLAGS>"
)
set(CMAKE_CXX_LINK_EXECUTABLE
    "<CMAKE_LINKER> -o <TARGET> <OBJECTS> <LINK_LIBRARIES> <CMAKE_CXX_LINK_FLAGS> <LINK_FLAGS>"
)

execute_process(COMMAND
    ${CMAKE_C_COMPILER} -print-libgcc-file-name -march=armv7-a -mcpu=cortex-a9 -mfpu=vfpv3
    OUTPUT_STRIP_TRAILING_WHITESPACE
    OUTPUT_VARIABLE libgcc
)

include_directories(BEFORE
    "/opt/xm/xal/include"
    "/opt/xm/xal/include/c++"
    "/opt/xm/xm/include"
)

add_compile_options(
    "SHELL:--include xm_inc/config.h"
    "SHELL:--include xm_inc/arch/arch_types.h"
)

link_directories("/opt/xm/xal/lib" "/opt/xm/xm/lib")
add_link_options("SHELL:-u start" "SHELL:-u xmImageHdr")
add_link_options(-T/opt/xm/xal/lib/loader.lds)
```

```
add_link_options("SHELL:--start-group ${libgcc} -lxm -lxal --end-group")

# Start address of the partition
# TODO integrate with the system with xpathstart
add_link_options(-Ttext=0x10000000)

# this disables tests that check whether PartitionMain was linked
# TODO change for actual tests
set(CMAKE_C_COMPILER_FORCED TRUE)
set(CMAKE_CXX_COMPILER_FORCED TRUE)

set(FPRIME_DISABLE_DEFAULT_LOGGER TRUE)
set(FPRIME_USE_BAREMETAL TRUE)

# DO NOT EDIT: F prime searches the host for programs, not the cross
# compile toolchain
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
# DO NOT EDIT: F prime searches for libs, includes, and packages in the
# toolchain when cross-compiling.
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

CODE LISTING C.2. Platform configuration

```
####
# XtratuM platform
###

add_definitions(-DTGT_OS_TYPE_XTRATUM)

# Disable everything. Based on XAL examples
set(CMAKE_C_FLAGS
  "${CMAKE_C_FLAGS}   -Wall -O2 -nostdlib -nostdinc -ffreestanding -Darm -fno-strict-aliasing -fomit-frame-pointer -fno-builtin -march=armv7-a -mcpu=cortex-a9 -mfpu=vfpv3"
)
# Disable everything AND exceptions and RTTI
set(CMAKE_CXX_FLAGS
  "${CMAKE_CXX_FLAGS} -Wall -O2 -nostdlib -nostdinc -ffreestanding -Darm -fno-strict-aliasing -fomit-frame-pointer -fno-builtin -march=armv7-a -mcpu=cortex-a9 -mfpu=vfpv3 -fno-exceptions -fno-rtti"
)

include_directories(SYSTEM "${FPRIME_FRAMEWORK_PATH}/Fw/Types/Xilinx")
```

D. DOCKERFILES

CODE LISTING D.1. XtratuM base image

```
FROM ubuntu:14.04

RUN apt-get update && apt-get install --yes \
    git \
    gcc-multilib \
    binutils \
    make \
    makeself \
    ncurses-dev \
    libxml2-dev \
    libxml2-utils

COPY . /

CMD [ "sh" ]
```

CODE LISTING D.2. F' system base image

```
FROM ubuntu:20.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -qq --yes python3 python3-venv
    python3-pip cmake gcc-multilib git

RUN git clone https://github.com/danoloan10/fprime.git /opt/fprime

RUN python3 -m venv /opt/fprime/venv && . /opt/fprime/venv/bin/
   activate && pip install --upgrade fprime-tools fprime-gds

CMD [ "bash", "-c", "cd /opt/fprime && exec bash" ]
```

CODE LISTING D.3. `2.0.5-arm` build image

```
FROM danoloan/xmfw:base

ADD https://github.com/Danoloan10/xm-arm/archive/refs/tags/2.0.5.tar
   .gz /opt/xtratum.tar.gz

RUN tar -C /opt -zxf /opt/xtratum.tar.gz && rm /opt/xtratum.tar.gz
   && mv /opt/* /opt/xtratum

COPY . /

CMD [ "sh" ]
```

CODE LISTING D.4. `2.6.0-ia32` build image

```
FROM danoloan/xmfw:base

ADD https://github.com/Danoloan10/xm-ia32/archive/refs/tags/2.6.0.tar.gz /opt/xtratum.tar.gz

RUN dpkg --add-architecture i386 && apt-get update && apt-get install --yes libxml2-dev:i386 && tar -C /opt -zxf /opt/xtratum.tar.gz && rm /opt/xtratum.tar.gz && mv /opt/* /opt/xtratum

COPY . /

CMD [ "sh" ]
```

CODE LISTING D.5. `1.0.8-sparc` build image

```
FROM danoloan/xmfw:base

ADD https://github.com/Danoloan10/xm-sparc/archive/refs/tags/1.0.8.tar.gz /opt/xtratum.tar.gz

RUN tar -C /opt -zxf /opt/xtratum.tar.gz && rm /opt/xtratum.tar.gz && mv /opt/* /opt/xtratum

COPY . /

CMD [ "sh" ]
```

CODE LISTING D.6. `2.0.5-arm-xalcxx` build image

```
FROM danoloan/xmfw:base

ADD https://github.com/Danoloan10/xm-arm/archive/refs/tags/2.0.5-xalc++.tar.gz /opt/xtratum.tar.gz

RUN tar -C /opt -zxf /opt/xtratum.tar.gz && rm /opt/xtratum.tar.gz && mv /opt/* /opt/xtratum

COPY . /

CMD [ "sh" ]
```

E. MEMORY MANAGEMENT

CODE LISTING E.1. `stdlib.c` memory management library implementation

```c
#include <stddef.h>
#include <stdlib.h>
#include <stdbool.h>
#include <xm.h>

/*
 * ... other functions
*/

struct __buddy_entry {
    bool free;          // u8 - 1B
    xm_u8_t order;      // u8 - 1B
};

/*
 * Auxiliary functions
 * These functions are just for convenience and understanding
 */

// N = M / ( 2^k + T )
static inline size_t __buddy_list_size(const struct xmPhysicalMemMap
    *xmPM)
{
    return (size_t)
        (xmPM->size / (BLOCK_BYTE_SIZE + sizeof(struct __buddy_entry
           )));
}

// H = M - N * T
static inline size_t __heap_size(const struct xmPhysicalMemMap *xmPM
   )
{
    return (size_t) xmPM->size
        - (__buddy_list_size(xmPM) * sizeof(struct __buddy_entry));
}

static inline struct __buddy_entry *__buddy_list_start
    (
        const struct xmPhysicalMemMap *xmPM
    )
{
    return (struct __buddy_entry *) ADDR2PTR(
             xmPM->startAddr + __heap_size(xmPM)
        );
}
```

```
43
44  static inline xmAddress_t __heap_start(const struct xmPhysicalMemMap
        *xmPM)
45  {
46      return xmPM->startAddr;
47  }
48
49  static inline xm_u8_t __min_greater_order(size_t size)
50  {
51      xm_u8_t order = ((size & (size - 1)) == 0) ? 0 : 1;
52      while (size >>= 1) order++;
53
54      if (order > BLOCK_BYTE_ORDER) {
55          return order - BLOCK_BYTE_ORDER;
56      } else {
57          return 0;
58      }
59  }
60
61  // MEMORY MANAGEMENT
62
63  struct __buddy_entry *buddy_list = NULL;
64  size_t buddy_list_size = 0;
65
66  struct xmPhysicalMemMap *xmPM = NULL;
67
68  void *malloc(size_t size)
69  {
70      xm_u8_t order, target_order;
71      xm_u32_t buddy, block;
72
73      void *mem = NULL;
74
75      if (xmPM == NULL) {
76          // get the memory map flagged to be used as heap
77          struct xmPhysicalMemMap *iter = XM_get_partition_mmap();
78
79          int i;
80          for(i = 0; (i < XM_params_get_PCT()->noPhysicalMemAreas) &&
                (xmPM == NULL); i++)
81          {
82              if (iter->flags & XM_MEM_AREA_FLAG0) {
83                  xmPM = iter;
84              } else iter++;
85          }
86
87          if (xmPM == NULL)
88              return NULL;
89      }
90
91      if (buddy_list == NULL) {
92          // initialize buddy list
```

```
93          buddy_list      = __buddy_list_start(xmPM);
94          buddy_list_size = __buddy_list_size (xmPM);
95          buddy_list[0].free  = true;
96          buddy_list[0].order =  __min_greater_order(__heap_size(xmPM)
                );
97      }
98
99      target_order = __min_greater_order(size);
100
101     for (block = 0; (mem == NULL) && (block < buddy_list_size);
            block += (1 << target_order)) {
102         if (buddy_list[block].free) {
103             order = buddy_list[block].order;
104
105             // split free blocks into buddies until the target order
                    has been reached
106             while (order > target_order) {
107                 order--;
108
109                 buddy = block ^ order;
110
111                 buddy_list[block].order = order;
112
113                 buddy_list[buddy].free  = true;
114                 buddy_list[buddy].order = order;
115             }
116
117             if (order == target_order) {
118                 buddy_list[block].free  = false;
119                 mem = ADDR2PTR(__heap_start(xmPM) + (1 <<
                        BLOCK_BYTE_ORDER) * block);
120             }
121             // if the order of the free block is less than the
                    target, we have not found anything
122         }
123     }
124
125     return mem;
126 }
127
128 void free(void *ptr)
129 {
130     xm_u32_t block, buddy;
131     xm_u8_t order;
132     xmAddress_t address;
133
134     if ((ptr == NULL) || (buddy_list == NULL))
135         return;
136
137     address = PTR2ADDR(ptr);
138
139     block = (address - __heap_start(xmPM)) >> BLOCK_BYTE_ORDER;
```

```
140       buddy_list[block].free = true;
141
142       order = buddy_list[block].order;
143       buddy = block ^ (1 << order);
144
145       // merge free buddies
146       while (buddy < buddy_list_size
147               &&  buddy_list[buddy].free
148               && (buddy_list[buddy].order >= order))
149       {
150           order++;
151
152           block >>= order;
153           block <<= order;
154
155           buddy_list[block].order = order;
156
157           buddy = block ^ (1 << order);
158       }
159   }
```

CODE LISTING E.2. `malloc` example (`partition.c`)

```
/*
 * $FILE: partition.c
 *
 * Fent Innovative Software Solutions
 *
 * $LICENSE:
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <xm.h>
#include <irqs.h>
#include <assert.h>

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

#define TIME(__expr) do { \
    xmTime_t t1, t0; \
    assert(XM_get_time(XM_HW_CLOCK, &t0) == XM_OK); \
    __expr; \
    assert(XM_get_time(XM_HW_CLOCK, &t1) == XM_OK); \
    PRINT("time: %d\n", (int)(t1-t0)); \
} while (0)

// Heap size 128MB
// For the heap to be this size, the memory region must be at least
//     128MB + 2^(27-k)*2
// where k is the BLOCK_BYTE_ORDER parameter configured in XtratuM
// The default value for k is 6 -> 132MB of mem are required
#define HEAP_SIZE (1 << 27)

```

```
void PartitionMain(void)
{
    int i;
    const char *hola = "hola", *adios = "adios";
    char *str = NULL, *other = NULL;

    // basic usage
    TIME(
        str = (char *) malloc(5 * sizeof(char));
    );
    assert(str != NULL);
    PRINT("hola?\n");
    strcpy(str, hola);
    PRINT("%s\n", str);
    TIME(
        free(str);
    );

    // check that free memory is used again
    other = (char *) malloc(1450 * sizeof(char));
    assert(str == other);

    // more than one object at a time
    str = (char *) malloc(6 * sizeof(char));
    assert(str != other);
    strcpy(str, adios);

    PRINT("hola?\n");
    PRINT("%s\n", other);
    PRINT("%s\n", str);

    free(str);
    free(other);

    // malloc more memory than available
    TIME(
        str = (char *) malloc(HEAP_SIZE + 1);
    );
    assert(str == NULL);

    // try to malloc more memory than available in two times
    other = (char *) malloc(1);
    assert(other != NULL);
    TIME(
        str = (char *) malloc(HEAP_SIZE);
    );
    assert(str == NULL);
    free(other);

    // now there is enough memory
    TIME(
        str = (char *) malloc(HEAP_SIZE);
```

```
    );
    assert(str != NULL);

    // check that memory can be used
    memset(str, 'o', HEAP_SIZE);
    for(i = 0; i < HEAP_SIZE; i++) {
        assert(str[i] == 'o');
    }

    PRINT("SUCCESS!\n");

    XM_halt_partition(XM_PARTITION_SELF);
}
```

CODE LISTING E.3. `malloc` example configuraton

```
<SystemDescription xmlns="http://www.xtratum.org/xm-arm-2.x" version
   ="1.0.0" name="hello_world">
    <HwDescription>
        <MemoryLayout>
            <Region type="rom" start="0x0" size="1MB" />
            <Region type="sdram" start="0x00100000" size="1023MB" />
        </MemoryLayout>
        <ProcessorTable>
            <Processor id="0" frequency="400Mhz">
                <CyclicPlanTable>
                    <Plan id="0" majorFrame="200ms">
                        <Slot id="0" start="0ms" duration="200ms"
                           partitionId="0" />
                    </Plan>
                </CyclicPlanTable>
            </Processor>
        </ProcessorTable>
        <Devices>
            <Uart id="1" baudRate="115200" name="Uart" />
        </Devices>
    </HwDescription>
    <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB" />
    </XMHypervisor>
    <PartitionTable>
        <Partition id="0" name="Partition0" flags="boot fp" console=
           "Uart">
            <PhysicalMemoryAreas>
                <Area name="main" start="0x10000000" size="256KB" />
                <Area name="heap" flags="flag0" start="0x10040000"
                   size="132MB" />
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>
</SystemDescription>
```

CODE LISTING E.4. c++ example (partition.cpp)

```
/*
 * $FILE: partition.c
 *
 * Fent Innovative Software Solutions
 *
 * $LICENSE:
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <xm.h>
#include <irqs.h>
#include <assert.h>
#include <xal.h>

#define PRINT(...) do { \
        printf("[P%d] ", XM_PARTITION_SELF); \
        printf(__VA_ARGS__); \
} while (0)

#define TIME(__expr) do { \
    xmTime_t t1, t0; \
    assert(XM_get_time(XM_HW_CLOCK, &t0) == XM_OK); \
    __expr; \
    assert(XM_get_time(XM_HW_CLOCK, &t1) == XM_OK); \
    PRINT("time: %d\n", (int)(t1-t0)); \
} while (0)

// Heap size 128MB
// For the heap to be this size, the memory region must be at least
//     128MB + 2^(27-k)*2
// where k is the BLOCK_BYTE_ORDER parameter configured in XtratuM
#define HEAP_SIZE (1 << 27)

```

```
class A {
    private:
        int a;
    public:
        int geta() {
            return a;
        }
        int geta(int &v) {
            v = a;
        }
        void sum(int b) {
            a += b;
        }
        A(int _a):a(_a) {
            printf("hola: %d\n", a);
        }
};

A global(10);

void PartitionMain(void)
{
    A a(3), *b;

    // local constructors work
    printf("%d\n", a.geta());
    a.sum(4 % 3);
    printf("%d\n", a.geta());

    // global constructors do not work
    printf("%d\n", global.geta());
    global.sum(4);
    printf("%d\n", global.geta());

    // test new and delete operators
    int bint;
    b = new A(5);
    b->geta(bint);
    printf("%d\n", bint);
    b->sum(4);
    printf("%d\n", b->geta());
    delete(b);

    XM_halt_partition(XM_PARTITION_SELF);
}
```

CODE LISTING E.5. c++ example configuraton

```
<SystemDescription xmlns="http://www.xtratum.org/xm-arm-2.x" version
   ="1.0.0" name="hello_world">
    <HwDescription>
        <MemoryLayout>
```

```
            <Region type="rom" start="0x0" size="1MB" />
            <Region type="sdram" start="0x00100000" size="1023MB" />
        </MemoryLayout>
        <ProcessorTable>
            <Processor id="0" frequency="400Mhz">
                <CyclicPlanTable>
                    <Plan id="0" majorFrame="200ms">
                        <Slot id="0" start="0ms" duration="200ms"
                           partitionId="0" />
                    </Plan>
                </CyclicPlanTable>
            </Processor>
        </ProcessorTable>
        <Devices>
            <Uart id="1" baudRate="115200" name="Uart" />
        </Devices>
    </HwDescription>
    <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB" />
    </XMHypervisor>
    <PartitionTable>
        <Partition id="0" name="Partition0" flags="boot fp" console=
           "Uart">
            <PhysicalMemoryAreas>
                <Area name="main" start="0x10000000" size="256KB" />
                <Area name="heap" flags="flag0" start="0x10040000"
                   size="132MB" />
            </PhysicalMemoryAreas>
        </Partition>
    </PartitionTable>
</SystemDescription>
```

CODE LISTING E.6. c++ example Makefile

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=../..

CONTAINER = container.bin
RSW = resident_sw

all: $(CONTAINER) $(RSW)

include $(XAL_PATH)/common/rules.mk

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SRCS := $(sort $(wildcard *.cpp))

CRTI_OBJ:=$(shell $(TARGET_CXX) $(TARGET_CXXFLAGS) -print-file-name=
   crti.o)
```

```
18 CRTBEGIN_OBJ:=$(shell $(TARGET_CXX) $(TARGET_CXXFLAGS) -print-file-
      name=crtbegin.o)
19 CRTEND_OBJ:=$(shell $(TARGET_CXX) $(TARGET_CXXFLAGS) -print-file-
      name=crtend.o)
20 CRTN_OBJ:=$(shell $(TARGET_CXX) $(TARGET_CXXFLAGS) -print-file-name=
      crtn.o)
21
22 #OBJS := $(CRTI_OBJ) $(CRTBEGIN_OBJ) $(patsubst %.cpp,%.o, $(SRCS))
      $(patsubst %.S,%.o, $(ASRCS)) $(CRTEND_OBJ) $(CRTN_OBJ)
23 OBJS :=  $(patsubst %.cpp,%.o, $(SRCS))
24
25 PARTITIONS=partition0.xef #partition1.xef
26
27 partition0: $(OBJS) $(XMLCF)
28     $(TARGET_LD) -o $@ $(OBJS) $(TARGET_LDFLAGS) -u init_array -
          Ttext=$(shell $(XPATHSTART) 0 $(XMLCF))
29
30 PACK_ARGS=-h $(XMCORE):xm_cf.xef.xmc \
31     -p 0:partition0.xef# \
32     -p 1:partition1.xef
33
34 $(CONTAINER): $(PARTITIONS) xm_cf.xef.xmc
35     $(XMPACK) check xm_cf.xef.xmc $(PACK_ARGS)
36     $(XMPACK) build $(PACK_ARGS) $@
```