Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Ivan Raul Lopez Guadarrama

Virtual Platform for the ARM Cortex-M0 Processor

Master's Thesis

Espoo, February 20, 2015

Supervisors:	Professor Heikki Saikkonen, Aalto University School of Science Professor Reinder J. Bril, Eindhoven University of Technology
Instructors:	Vesa Hirvisalo, D.Sc. (Tech.). Pekka Nikander, Ph.D.

In memory of María del Carmen

Aalto University School of Science Degree Programme in Computer Science and Engineering	ABSTRACT OF THE MASTER'S THESIS	
Author: Ivan Raul Lopez Guadarrama		
Title: Virtual Platform for the ARM Cortex-M0 Processor		
Number of pages: vii + 68	Date: 20/Feb/2015	Language: English
Professorship: Software Systems	Code: T-106	
Supervisors: Professor Heikki Saikkonen, Professor Reinder J. Bril		
Instructors: Vesa Hirvisalo (D.Sc. Tech.), Pekka Nikander (Ph.D.)		

Abstract:

Modern microcontrollers provide a 32 bit core, a rich set of peripherals and on chip memories. However, due to the recent slowing down of the exponential increase in RAM performance, memory has become the main cost factor of low-end MCUs. For enabling the IoT evolution, and until a technological breakthrough improves memory performance, inexpensive sensing and actuation nodes will be heavily memory constrained.

Ell-i Open Source Co-operative developed an Open Source Hardware prototype PoE enabled IoT node, which targets to use the smallest ARM Cortex-M0 MCU, with a maximum of 4kB of RAM. Although multiple Open Source Real-Time Operating Systems were available, none of them satisfied the requirements. These included fitting into the memory footprint without minimal configuration, properly handling of the hardware interrupt controller, or adequate alignment with the company's business plan.

This work provides a memory constrained scheduler that rivals in performance and memory footprint the evaluated Open Source RTOSs alternatives. Additionally, it provides safety features not present in other systems, while providing the necessary alignment to Ell-i Co-Operative. A comprehensive evaluation of popular RTOSs for the Cortex-M0 architecture is included, as it allows the benchmarking of the developed system.

Keywords: ARM, Cortex-M0, Real Time, voluntary preemption, context switch, semaphore, RTOS, PoE, IoT, Open Source, Open Source Hardware, Business Models

Acknowledgements

I would like to be thankful with my instructors for their contributions to this project; Vesa Hirvisalo for his guidance in academic methods as well as writing, and Pekka Nikander for providing theoretical and technical assistance, suggested literature, and even personal help in times of need. Additionally, I would like to offer my gratitude to Heikki Saikkonen for taking the time to supervise this work, and to Reinder J. Bril for providing the initial ideas that seeded the topic selection.

Furthermore, I want to thank all the members of Ell-I Co-Operative, for giving me time to work on my thesis, even when there were more important things to do. Special thanks are directed to Teemu, Eero, Lari, Otso, and Asif.

Last, but not least, I want to thank my family and girlfriend for providing the support I needed, as well as times of peace and joy that allowed me to refocus and continue writing. This includes, but is not limited to: Paola, Rebeca, Raul and Katya. Finally, I want to send special thanks to my friends Simon and Vlad, for their resistance to my everyday complaints and unconditional support.

Espoo, February 20, 2015.

Ivan Raul Lopez Guadarrama

Contents

Acknowledgements.......... iii

Contents.......... v

Acronyms.......... vii

Chapter 1 - Introduction.......... 1

1.1 Problem.......... 1

1.2 Contributions.......... 2

1.3 Thesis Structure.......... 2

Chapter 2 - An Open Source Hardware Ecosystem.......... 4

2.1 Context of Open Source Projects.......... 4

Open Source Software.......... 4

Open Source Hardware.......... 4

Open Source Hardware Business Models.......... 6

2.2 Ell-i business model.......... 8

Ecosystem creation.......... 9

Chapter 3 - Scheduling in Real Time Systems.......... 11

3.1 Beginnings and Development.......... 12

Limited Preemption.......... 14

Context Switch Improvement.......... 15

3.2 Synchronization in Real Time Systems.......... 16

Priority Inheritance.......... 16

Priority Ceiling and Stack Resource Policy.......... 17

Chapter 4 – Ell-i Development Environment.......... 18

4.1 Arduino Ecosystem.......... 18

Background.......... 18

Programming Model.......... 19

4.2 Ell-duino Development Board.......... 22

ARM Cortex-M0 Processor.......... 22

PoE and CoAP Support.......... 23

Arduino Compatibility.......... 24

4.3 Provided Runtime.......... 25

Peripheral Initialisation.......... 25

Context Switch Prototype.......... 26

4.4 Other (Open Source) RTOS.......... 28

Previously Published Benchmarks.......... 29

Chapter 5 - Memory Constrained Scheduler 31
5.1 System Architecture 31
System Exceptions and Priorities 31
Initialization 31
API 33
5.2 Scheduler 34
Data Structures 34
Context Switch State Space 35
5.3 Semaphores 37
Data Structures 38
Blocking Pre-Check 38
5.4 Time Management 40
Data Structures 41
Interaction with Scheduler 42
5.5 Memory Management 43
Linker Interaction 43
Thread Initialisation Records 43
Decompression Process 44
Chapter 6 – Evaluation 46
6.1 Setting 46
6.2 Procedure and Metrics 47
Software Testing Framework 47
Hardware Testing Framework 48
6.3 Workload 50
6.4 Results 52
6.5 Discussion 55
Timing 55
Memory 57
Chapter 7 - Conclusions and Future Work 59
References 61
Appendix A - Measurement of Hardware Exception Handling Times 68
Direct Measurements 68
Indirect Measurements 68

Acronyms

CAD	Computer Aided Design
FPDS	Fixed Priority Scheduling with Deferred Preemption
FPNS	Fixed Priority Non-Preemptive Scheduling
FPPS	Fixed Priority Preemptive Scheduling
IoT	Internet of Things
IP	Intellectual Property
M2M	Machine to Machine
MCU	Microcontroller Unit
MPS	Maintain Power Signature (PoE standard specific term)
OSHW	Open Source Hardware
OSS	Open Source Software
PCB	Printed Circuit Board
PCP	Priority Ceiling Protocol
PD	Powered Device (PoE standard specific term)
PIP	Priority Inheritance Protocol
PoE	Power over Ethernet
RTOS	Real Time Operating System
SoC	System on Chip
SRP	Stack Resource Policy
TCB	Task Control Block

Chapter 1 - Introduction

The advance in semiconductor technologies has allowed an exponential improvement of the processing power, memory capacity and other metrics of computing hardware during the last fifty years. This continuous evolution, predicted by Gordon Moore in 1965, is the reason why modern microcontrollers can fit a 32bit processor, a rich set of peripherals, and integrated memories in a very small and cheap package. Additionally, the widespread use of processing cores designed by ARM Holdings in mobile and embedded applications has led to a commoditisation of the low-end MCUs.

This commoditisation process generates an extensive offer of cheap, yet powerful MCUs. Thus, it has been one of the main driving forces behind the Internet of Things (IoT) evolution or the Open Source Hardware movement. However, in recent years the pace of the Moore's law progression has started to slow down. This trend is particularly affecting the memory component of computing systems, making it a bottleneck to the performance of comparatively faster processors, and the main cost factor of microcontrollers and mobile systems.

In order to enable the IoT evolution, the development of inexpensive embedded systems capable of sensing, actuating and communicating in real time is essential. Therefore, as memory is the main pricing element, the software stack targeted for those devices should limit as much as possible the memory usage, while still providing comprehensive networking and sensing capabilities.

Moreover, this thesis is based on the internship work at Ell-i Open Source Co-operative. Ell-i is an Open Source Hardware organisation which targets to provide Power over Ethernet (PoE) enabled nodes as part of the IoT ecosystem. This prototype nodes are planned to be relatively inexpensive by the inclusion of the smallest MCUs from the Cortex-M0 family, which are very limited in RAM memory.

1.1 Problem

Real Time Operating Systems (RTOSs) for microcontrollers are not new, neither are their Open Source implementations. In fact, Open Source RTOSs have widespread usage due to the performance they provide at zero cost. However, even if there exist a variety of options available for the ARM powered MCUs, this variety diminishes drastically towards the low-end devices. Consequently, none of them fulfilled the specific requirements for driving the Ell-i PoE node prototype, which targets to use the cheapest Cortex-M0 processors, with very limited RAM in the range of 2 - 4 KB. In particular there are three key aspects that none of the available Open Source RTOSs managed to fulfil. Two are technical and the third one is related to the Ell-i business strategy.

The first technical problem is that the majority of systems available are targeted to more powerful processors, and require a 'minimal' configuration to fit on the low-end MCUs. Although some systems have ports to architectures part of the Cortex-M family, only a few are available for the Cortex-M0 processor. Some of the issues with 'minimal' configurations include that the system still provides more functionality than needed, removing functionality is not efficient, or the reduced configuration does not translate into memory savings or performance increase. As a consequence, designers either select a device closer to the high-end, or prefer solutions which obviate threading requirements, if possible.

The second problem is that most Open Source RTOSs are designed to be as generic and platform independent as possible. Although this is generally a benefit, it becomes a problem when some architectures have peculiarities that challenge the soundness of the software design. In particular, the interrupt system of the ARM Cortex-M provides very low latency at the cost of possible complex hardware events interleaving. This complexity not entirely addressed by the available RTOSs, and generates a latent risk for systems used in hard real-time applications and other mission-critical situations.

The third problem is closely related to the alignment between the scheduling software and the business model of Ell-i. In concrete, two specific requirements were not fulfilled by other Open Source RTOSs. The most critical is the necessity to collectively own the copyright of the software, in order to license it to companies as part of the business strategy. The second requirement is that, as an Open Source Hardware organisation, it is ideal to provide software specially tailored to the designed hardware.

1.2 Contributions

The main contributions of this thesis are related to the design and implementation of a memory constrained scheduler tailored for the Cortex-M0 architecture, with focus on its hardware interrupt controller. The objective is to enable basic threading applications on the low-end MCUs with a maximum 4KB of RAM and very limited Flash. There are three key benefits the developed system provides in comparison with available Open Source RTOS.

First, the implementation provides a very small footprint for both, RAM and Flash. Moreover, it follows a similar strategy as the data-driven peripheral initialisation concept developed at Ell-i, generating a complete solution. Second, the system provides low latency for synchronisation primitives and context switches. This leads to fast performance for Real Time applications which require very limited interference from system interrupts. A key benefit is that there is no ‘minimal’ configuration, leading to the efficient usage of all the involved resources. The third benefit is the safety derived from the careful analysis of the hardware interrupt system of the Cortex-M0 architecture and the static allocation of all the system level resources at compile time.

An additional contribution regards to the business side of Ell-i: the ownership of the copyright of the developed scheduler. This element is a key factor for the licencing business of Open Source material. Moreover, this thesis represents a detailed documentation of the implemented system, which serves as a reference for further development. It is also important for the strategy to provide Open Source Software as a bundle specifically crafted to highlight the possible use cases of Open Source Hardware.

1.3 Thesis Structure

The rest of this thesis is organised as follows. In the two following chapters, the background of the thesis is discussed. Chapter 2 deals with the business background, locating the current development in the framework of an Open Source Hardware ecosystem. Chapter 3 deepens into the technical details of Real Time Systems as a basis to analyse RTOSs implementations.

Chapter 4 provides insight on the material that served as a basis for the development of this project. This includes a description of the hardware platform used as a target for the software stack, the previously developed peripheral initialisation system from Ell-i, and a discussion of various RTOSs which codebase is available (including Open Source and Proprietary systems).

Chapter 5 offers a detailed description of the implemented scheduler, as well as the design rationale behind the selection of particular data structures and interfaces. In particular, section 5.2 offers details of the core scheduling system and serves as a basis for the other sections of this chapter.

Chapter 6 describes the evaluation methodology, provides the results, and offers a discussion based on them. Specifically, the resulting data is presented in section 6.4, while its discussion is located in section 6.5. Finally, Chapter 7 concludes the thesis by revising its contributions, attempts to relate them to a broader problem space, and describes possible directions of future developments.

Chapter 2 - An Open Source Hardware Ecosystem

The purpose of this chapter is to review the literature regarding Open Source Software (OSS) and Open Source Hardware (OSHW). It begins by discussing the history and current state of Open Source projects, both OSS and OSHW. Then, a detailed description of business models for OSHW is presented. Finally, the specific business model of Ell-i is discussed, with particular focus on the ecosystem creation.

2.1 Context of Open Source Projects

Open Source Software

The idea of free software is not new. During the beginning of the computer evolution, computers were regarded as research tools and only owned by universities and corporations. At that time, software was shared freely among educational institutions, as the act of programming was paid, not the program itself. When the costs of computers decreased and they became widespread available, programmers started to restrict the rights to their software and charging fees for each copy [1].

The origins of the current use of the "Free Software" can be tracked to 1984 when it was used as a political argument by Richard Stallman [1]. This movement was also the beginning of the Free Software Foundation, the GNU Project, and the GNU General Public License (GPL). Similarly, Eric Raymond popularized the idea of Open Source in his classic article "The Cathedral and the Bazaar" [2]. The importance of the "source code" is particularly relevant, as the openness of the source implies not only free distribution, but the possibility to modify and build upon the provided software.

The Open Source Software (OSS) has proven to benefit multiple components of the IT ecosystems. For the developer group it reduces the costs of the development process and enables a virtually no cost distribution strategy. For business users, it enables the deployment of servers without the OS license fees. In addition, Ubuntu, a variation of the Linux Open Source OS, was regarded as the most secure end user operating system for end users by the UK government [3].

Nevertheless, the OSS trend has not only become politically important, but also has proven to be a profitable business by its own. Red Hat, one of the companies that sells support for their own Linux Open Source OS, reported a revenue of 1.33 billion USD for the 2013 fiscal year [4], and employs more than six thousand people worldwide [5].

Open Source Hardware

The idea of Open Source Hardware (OSHW) can be historically related to the open interfaces in the computer industry. When IBM competed with Apple at the personal computer market, they decided to use off-the-shelf parts instead of internally designed components. The decision was necessary in order to reduce internal costs and handle the very short time to market. As a result, they enabled the first open interface computer platform [6]. In short time, other companies like Dell and Gateway started producing completely compatible IBM PC's.

The availability of the internal details of hardware was at some point necessary for enabling adequate support. An example is found in the Tektronix Oscilloscopes form the 1950's. Some of the devices were very expensive, around half a year salary of an engineer, and had vacuum tubes with very short lifespan [7]. The necessity for constant changing required the product to be sold together with very detailed service manuals. The documents not only described how to operate the device, but detailed the entire internal functionality, with a bill of materials and schematic diagrams, in order to allow the user to replace parts as needed.

Although the OSHW is based on very similar philosophical roots as OSS, it was not until the reduction of prices of components and the widespread use of Internet services that the first practical projects started to materialize. Nevertheless, the differences with OSS have posed new challenges, some of which still require advances in technology and legislation. The main practical differences between OSHW and OSS have already been drafted by the academic literature. The differences and problems detailed below are based on the analysis of Locke [7], Acosta [8], and Rubow [9].

One practical difference regards distribution. Software is a non-tangible asset that can be distributed at practically no cost through Internet. In contrast, hardware has inherent costs of distribution and storage. Another problem is updatability. While software can be automatically updated and with no cost, hardware requires the substitution of sections, parts, or even the whole unit, which is not automatic and definitively not free.

An additional constraint, which is projected to drop rapidly, is the cost of manufacturing and testing equipment. While software requires a relatively inexpensive personal computer, the infrastructure required to manufacturing and testing of electrical and electronic hardware can be very expensive. This factor limits certain processes to be solely available to large companies, excluding in practice the participation of amateur and semi-professional developers. Nevertheless, the prices of this kind of equipment are dropping very rapidly, partially as a consequence of the open hardware trend, which also affects manufacturing and testing hardware.

An additional problem related to the physical existence of OSHW, which is also expected to decline at some point in the future, is the reliance on suppliers. Although many of the passive and simple components are available from multiple suppliers, the more advanced ones, like MCU's and specific peripherals are only provided by one specific manufacturer. As the configuration of the hardware is designed to work with that particular component, substitution requires extensive modification of hardware designs and related software. However, the commoditization led by ARM Holdings has partially diminished the costs of changing MCU manufacturer, at least from the software side, as the common architecture allows partial reuse of existing code and related tools with changes only required on the proprietary peripherals.

While software is essentially the output of text (source code) passed through a compiler, it was not until the maturation of the GCC compiler (first widespread Open Source compiler) that the Open Source trend growth rapidly. This compiler was one of the main enabling factors for the Open Source success. In the same way, hardware is designed used specialized Computer Aided Design (CAD) tools. Currently, the majority of these tools is proprietary and the open source options are not mature enough. Furthermore, the proprietary and most used ones are from competing companies, decreasing considerably compatibility. Although it is possible to "export" the design to open formats, it is not possible to modify the design and returning it to the proprietary format.

Another problem generated by the use of current proprietary CAD tools is the impossibility of automatic merging. While there exist extensive sets of tools for merging and version control for software (Source Code Management), they are all based upon the fact that source code is actually plain text. However, that trend is also in decrease, and merging of hardware models is expected to be more automated, as the collaboration of teams through internet services is becoming the most frequently and cost efficient solution. Unfortunately, in the hardware design nowadays, merging requires human effort, making the process more costly.

Furthermore, copyright is the legal framework that protects software. Concretely, it can be used to prevent deviations of the terms of the license that would fall into copyright infringement. Unfortunately, the majority of the components of OSHW, such as particular circuit designs or the selection of specific components, is not covered by the protection of copyright [7]. A possible alternative is to protect hardware designs by patent laws. However, it is not as simple as copyright: apart from being expensive and very time consuming, it requires the invention novelty. The problem is that patents were designed for protecting a different kind of intellectual output. In summary, the protection of OSHW is still an open discussion, with multiple prototype license types already present but with no general acceptance.

Open Source Hardware Business Models

The purpose of this section is to review the most prominent business models available for OSHW. Although some of the models described below were initially designed for the software, they are carefully selected as they can also provide channels of revenue for the hardware counterpart. Nevertheless, there are also models which origins and validity are closely related to the specific tangible characteristics of hardware.

Support Sellers

First, it is necessary to provide a brief reference to the usage of OSS. Although the Open Source characteristic implies that the software can be modified at will, the reality is that just a few individuals and organizations outside the main developer group actually touch the code and build upon it. In fact, the majority of users only "use" the code, as the technical skills and background required to successfully modify the source code to add functionalities are relatively unique and specific [10].

As a consequence, when an organization which requires a specific change or simply support for the OSS commonly seeks for assistance from the main developers. The main advantage of is a zero cost advertisement and distribution campaign, while support is the main revenue channel. This model has been widely referenced in literature related to Open Source [7, 11, 12], and is probably one of the first successful business strategies built around an open source ecosystem.

The custom development and support for hardware also requires expertise on the fields that the project of the customer encompasses. In fact, the set of abilities required is even more unique, as the core development team should have proficiency in both, hardware and software. The two main kinds of support for open source projects [10] will be discussed in the sections below.

Professional Support

The first kind of strategy that proved profitable consisted of selling professional services, such as customer support and maintenance, for an open source codebase. However, as the workforce behind the service providing corporation is limited, only particular versions (revisions) of the software are supported. Additionally, the software is only supported with the constraint that the user does not modify to source code. Such modifications commonly lead to violations of the service agreement [10].

Custom Development and Support

One special case of the support business model is to provide support to the modifications that the client makes to the original code. As discussed earlier, support sellers only provide service to the unmodified and updated versions. However, it is also possible to provide services and support the modifications the client makes to the original codebase. Two notable examples for OSS are Gluecode, later acquired by IBM [13], and Specifix [10].

In the case of hardware, the same kind of personalization can be offered. Instead only supporting the reference designs, tailored services can be provided to generate hardware and software solutions that fit the particular needs of the client. The particularity that hardware requires software to run over it provides an additional layer of customization. Additionally to the proficiency in software and hardware (which is specific and unique), expertise on the field of the client's project is also necessary.

Brand Licensing

The branding model is based on the ownership of a trademark, and the free distribution of the open source outcomes [11]. In particular, only the products, either software or hardware, generated by the development group can hold the brand name. This brand represents some added value to the customer, either in the form of additional testing phases to ensure quality, increased confidence in the support and extended periods of sustained maintenance, or simply by the expertise represented by the core developer team [12].

However, the revenue generation for an open source company following this strategy is not based solely on the possession of the trademark, but in the selling of this brand qualifier in a fashion similar to franchising. This strategy is particularly effective in the commercialization of OSHW, which manufacturing is relatively expensive and leads to a non-free distribution cost. A branded scheme benefits the manufacturer that buys the brand, as it can charge an additional price to the product, and compete on basis of quality and other attributes.

Additionally, three benefits are presented to the core developer team. First, it obtains revenue for the trademark commercialization. Second, the quality and other attributes of the product can be ensured by the agreement. Third and finally, it is not necessary to produce large quantities of the hardware product in order to generate revenue for the manufacturer, which is particularly important when the company is at its start-up stage as the demand is very low compared to fully established products.

Core Open (Loss Leaders)

The loss leader business strategy is commonly used by established companies that release a piece of software as open source in order to increase their market share where they have low participation [11, 12]. The classic example of this strategy is the Android OS developed by Google. The target of the development was the revenues that an application market associated with the platform can generate. At the time of development, the market share was completely dominated by Apple and their iOS operating system for smartphones. Nevertheless, the open source distribution strategy proved extremely successful, as Android has more than 80% of the smartphone OS market share by the first quarter of 2014 [14]. The application for hardware follows the same strategy. In particular, large MCU manufacturing companies could provide OSHW designs that use their own devices.

Dual License Model

This model is based on the fact that *reciprocal licenses* (which enforce any modification to be distributed with the same license) do not allow additional modifications to be released as closed products. This limitation reduces the possibilities of commercialization, as organizations usually prefer to keep their changes private. Nevertheless, the utilization of OSS and OSHW as a component of closed, private products is possible by the implementation of *academic licenses*, which allow the distribution of modifications in either open or closed approach [10, 15].

However, in order to enable the use of an Open Source project in a closed fashion, the development organization should own the copyright of the material. In order to achieve this, the organization either employs all the developers, or agrees the transfer of rights with the outside developers that modify to the core distribution. As this requires additional effort, it can directly be monetized. In other words, the client pays for the right to choose what license is more adequate for their purpose [10].

Bulk Discount Model

The bulk discount model is based on the reduced costs that economies of scale provide when a manufacturing process is increased in numbers. As a consequence, this model only valid to OSHW. However, during the initial phase of most OSHW projects, the quantities required for the testing prototypes or initial demands are too small in comparison with the ones that benefit from economies of scale. Nevertheless, this effect can be reduced by the addition of an exclusivity or branding model. Consequently, manufacturers can afford some initial loss in exchange of possible future profits and lack of competition. This model is mainly used by the designers of the Raspberry Pi, which have an exclusive distribution agreement with RS and Farnell [16].

2.2 Ell-i business model

The strategy from Ell-i focuses on designing inexpensive Power over Ethernet (PoE) development boards compatible with the Arduino ecosystem. The use of ARM 32 bit architecture in conjunction with simple Ethernet controllers and carefully crafted PoE components allows the use of only one MCU device per board.

The computing power of the Cortex-M0 architecture allows the integration of the network stack into the main MCU, which reduces the additional cost of Ethernet controllers with integrated network stack, or the use of an additional MCU to handle the networking services. Additionally, the integration of PoE to the board eliminates the need of external components, such as the proprietary devices used for the PoE support in the Arduino platform [17, 18].

Ell-i proposes the co-operative organizational model and the egalitarian ownership as a form of innovation in the electronics industry and the OSHW community. The co-operative concept provides benefits from both, external and internal perspectives. From the external side, it is a strategy to cope with the negative forces that the world economic policies can exert over companies [19]. From the internal side, it is a channel to distribute responsibility more evenly across members, compared to other organizational models.

In particular, the application of egalitarian ownership increases further the uniformity of responsibility distribution, as employees own equal parts of the company. This leads to important benefits, such as decisions being taken in a democratic fashion, increase in the involvement on strategic planning, and a very flat hierarchical relationship among members [20]. This kind of ownership is also positive to the external ecosystem, as responsibility can extend beyond the boundaries of the organization and spread among the community, which is ideal for an open source project.

Ecosystem creation

As discussed earlier, the co-operative model with egalitarian ownership is also innovative for the Open Source ecosystem. The hierarchical structure among OSS teams can be quite challenging, in particular for new developers. Although the hierarchy is more dynamic when compared to proprietary projects [7], it still requires considerable effort. As an example, new developers for the Linux kernel are expected to work at least two years before receiving committer status [10]. As a consequence, Ell-i organizational structure is very attractive for new developers, which benefit from a less steep start and are able to participate in organizational decisions as soon as they become members. For the co-operative point of view, a fast community growth in comparison to more hierarchical projects is a very important differentiator, mainly during the start-up phase.

As in any other Open Source project, the identification of the reasons why developers participate without economic remuneration is of high importance. In general, the revised literature groups the motivating factors between internal and external. From the internal side, the most cited motivators are altruism and satisfaction of psychological needs. From the external ones, the most commonly referred are: reputation or prestige, access to the knowledge base and building of skills, and satisfaction of specific personal needs that the Open Source project solves.

Altruism is regarded as the most obvious internal motivation [21, 22, 23]. It is simply translated in the gratification of sharing a solution for a complex problem to anyone who might need it. In addition, the satisfaction of diverse psychological needs (equivalent to the upper Maslow needs [24]), is referred in multiple forms, such as intellectual gratification and creativity [7], intrinsic motivation and community identification [22], or simply as psychological needs [23]. In particular, Open Source projects provide a level of liberty that is hardly available for developers working inside corporations with hard structures.

Reputation is the most important external factor, as it provides present and future opportunities for personal career of the developers. It is referred as reputation [7, 21, 23], prestige [7], or peer recognition and self-marketing [22]. The access to the knowledge base and the acquisition of new skills are another important motivations for developers. It is referred as access to knowledge [21] or development of human capital [22]. In particular, Open Source projects give opportunities to students to participate in real life projects, which would be very complicated in a normal organization. Finally, the solution for a personal problem of the developer can be a strong motivation, mainly for the first movers of new projects [22].

The interaction between these motivating factors is also relevant for the general performance of the participating developers. In general, external motivations have more weight on the decision of participating or continuing in an open source project. However, students and hobbyists are a special case, mainly driven by internal motivations [22]. Regarding internal motivations, the effort intensity of developers is commonly strengthened by the satisfaction of psychological needs, mainly autonomy, and decreased by the altruistic behaviour. Apparently, the enjoyment of sharing knowledge distracts the participant from reaching specific goals and providing significant results [23].

Chapter 3 - Scheduling in Real Time Systems

As the literature regarding real time systems scheduling is extensive, it would be out of the scope of this literature analysis to target all its extent. For this reason, this literature review will be limited to the fixed priority preemptive scheduling (FPPS) upon a single processor, its evolution and selected topics of current development. However, it is relevant to briefly describe the most important topics that real time scheduling covers nowadays. As multiple variables can be selected to divide and classify real time scheduling algorithms and applications, the most relevant ones are described below.

First, the algorithms can be classified based on the stage at which calculations of the scheduling decisions are done. In that sense, algorithms can be classified as either static or dynamic. From one hand, static algorithms set the scheduling decisions in advance, require previous knowledge of the characteristics of the tasks, and result in little runtime overhead. Additionally, if the calculations of the scheduling sequence are done before the running the system, the static algorithm is also classified as offline [25]. From the other hand, dynamic algorithms take decisions during runtime, leading to more adaptable systems able to handle unexpected levels of activity. As a consequence, they are suitable for systems that include with soft deadlines, or situations where a processing upper bound cannot be calculated.

Second, scheduling algorithms can be classified by the restrictions to preemption that they enforce. From one side, preepmtive algorithms allow tasks to be halted at any point in time, and continue at a later moment without affecting the behaviour and correctness of their operation. The only difference is that the total execution time is extended. From the other side, non-preemptive algorithms treat tasks as continuous processing sections that cannot be stopped once they are started. This kind of behaviour offers a solution to mutual exclusive operations in concurrency situations, and is closely related to blocking.

An additional type of algorithms, referred as hybrid, or of limited preemption [25], present a behaviour between the fully preemptive and the non-preemptive schemes. This strategy allows tasks to execute for a short extra time before they are suspended. The systems that implement limited preemption use preemptive behaviour as a general policy, but treat special sections of execution as non-preemptable. In case a task is requested to be halted at this sections, the preemption is deferred until the atomic section is completed. For this reason, the algorithms are also called of deferred preemption.

Third, scheduling mechanisms are classified by the level of interaction between tasks they allow. The simplest case treats all tasks as independent entities, without precedence relationships or resource sharing constraints. Unfortunately, this case is greatly simplified to be useful for practical system. The opposite case, closer to reality, treats tasks as highly interrelated entities. The extended analysis of the interaction between tasks by synchronization primitives will be detailed on section 3.2 Synchronization in Real Time Systems

Fourth, the set of tasks can contain either only periodic processes, or a mixture of periodic and aperiodic requirements. In general, aperiodic or non-periodic are terms used to define any process that do not match with the fixed period, bounded computation time, hard deadline scheme. The analysis of the strictly periodic case was the seminal work of the fixed priority scheduling theory. However, it has been extended by subsequent work in to encompass more general cases like soft deadlines, sporadic task arrival, or highly variable execution time tasks.

Fifth, algorithms can be further classified by their ability to schedule tasks to more than one processor. In general, the extension of algorithms from the uniprocessor case to a multiprocessor architecture leads to counter intuitive, suboptimal results. In fact, the problem of optimally scheduling on multiple processor is NP-hard, as meeting low latencies and balanced utilization exceedingly challenging [25]. As a consequence, the multiprocessor case usually needs to be simplified in order to enable the use of uniprocessor techniques, which leads to suboptimal results.

3.1 Beginnings and Development

1. All tasks are periodic
2. All tasks have a deadline equal to their period
3. All tasks are independent (no shared resources or precedence relationships)
4. All tasks have fixed computation time, or at least, a fixed upper bound
5. A task cannot suspend itself
6. All tasks are released at the beginning of their period
7. All overheads are ignored, and assumed to be zero
8. A critical instant exists
9. All tasks are fully preemptive
10. Just one processor is available

Table 1. Constraints of the Liu and Layland model

The seminal work for the evolution of FPPS theory was the 1973 article from Liu and Layland that elaborated a polynomial feasibility test for the rate-monotonic scheduling algorithm [26]. From that point onwards, the constraints posed by this work (listed in Table 1) were steadily removed by subsequent research. Consequently, new directions and applications emerged from the field of real time scheduling. The most relevant trends of evolution are described below.

The first trend is related with the feasibility analysis. In general, new methods were devised to allow relaxation in the constraints imposed by the model of Liu and Layland [26]. Initial approaches, during the beginnings of the 80's, were based on the calculation of the Least Common Multiple (LCM) of the periods of tasks. However, this approach became inefficient even for small sets, preventing extension and applicability [27]. As a result, response time analysis became the preferred tool for the analysis of rate-monotonic scheduling.

One consequence of advancing the response time theory was that the priorities were no longer restricted to the rate-monotonic scheme to allow feasibility analysis. The only limitation was that the priorities of tasks should remain constant during runtime (static). Another consequence was the removal of the deadlines equal to period constraint, which resulted in a framework that included valid analysis for deadlines less than or greater than periods [26, 27]. This extension provided the foundations for the deadline-monotonic scheduling algorithm, which is closely related to the rate-monotonic one. The third result was the partial relaxation of the critical instant requirement, allowing arbitrary phasing. In general, the critical instant analysis leads to pessimistic results when applied to a set with arbitrary offsets.

Although the response time analysis is useful for the offline analysis of systems, computational efficient tests are still preferred for the evaluation during runtime. One of the recent advances is the Hyperbolic Bound test [28] for the rate-monotonic priority assignment. This test "improves" the feasibility of the original Liu and Layland test, while keeping polynomial complexity. However, both tests are necessary but not sufficient, as some task sets are still schedulable even if they fail to satisfy the schedulability bounds.

The second trend is related with the interaction between tasks and the subsequent relaxation of the independence constraint. However, these topics will be treated in detail in section 3.2 Synchronization in Real Time Systems

The third trend deals with the extensions to the model to allow the inclusion of aperiodic tasks. As briefly discussed earlier, aperiodic tasks do not fit the classical periodic model, including cases with soft deadlines, significantly varying inter arrival time, or significantly varying computation time [29]. The initial and simplest approach is to allocate these tasks with lower priority with respect to the hard deadline set, effectively relying them to the background. However this strategy greatly increases latency. As a solution, more advanced approaches were devised to improve response time (quality of service) of soft deadline tasks without affecting the hard time constraints.

One of the first approaches was to use a Polling server, which behaves like a periodic task and is usually located at the highest priority level. This design approach consisted on extending the capacity of this server to the maximum possible, such that the set is still schedulable. However, multiple drawbacks emerged, such as wasting high priority capacity (in the case that no soft processes were available at the time), and long response times when capacity was depleted previous to an aperiodic request.

The drawbacks of the polling server were addressed by a set of algorithms known as "bandwidth preserving", which include the Priority Exchange server, the Deferrable server, and the Sporadic server. They allow the conservation of processing time even when no aperiodic task workload is available. However, they easily degrade and provide the same performance as the Polling server. As a consequence, another family of algorithms was designed to recover the unused capacity left by the hard deadline tasks.

The most prominent of this family is the Slack Stealing algorithm. It is optimal in the sense that it minimizes the response time of soft tasks amongst all algorithms that that meet the required hard deadlines [26, 27]. However, the limitations that preclude it widespread application include the calculation of the LCM of the task set. This only allows the use of periodic tasks without jitter, as well as only being feasible in practice for sets with small LCM.

In general, an open question remains of finding ways of improving system utility by using the spare capacity released by multiple factors, which include:

- Tasks completing in less than their worst case execution time
- Sporadic tasks not arriving at their maximum rate
- Periodic tasks not arriving at their worst case, critical instant, phasing

The fourth trend is related to the handling of transient overloads and adaptability to unexpected conditions. The main reason for this situations is that, in some cases, the calculation of the worst case execution time might be inaccurate or simply impossible [29]. Just as an example, an overload of a system based on the rate-monotonic algorithm will cause tasks with longer periods to miss their deadlines. However, this behaviour might be problematic, as the importance of a tasks could be not directly related to its priority. In general, the feasibility analysis with pessimistic assumptions, or the assignment of high criticality to certain set of tasks are valid alternatives to handle transient overloads.

The fifth trend discusses the simplifications and related considerations applied porting real time systems to either software or hardware. In general, the constraint of zero overhead is not valid on implemented systems, as the kernel operations need to be accounted. These operations, such as time progress handling (in either event driven or tick driven systems) or context switch overheads impose additional delays which deviate from the ideal operation. Further details on strategies to reduce this overheads are presented in the Context Switch Improvement section.

This section has briefly described the last decades of evolution of the fixed priority scheduling theory. As some of the topics are closely related to this dissertation, a more detailed and updated discussion will be presented in further sections. In concrete, two particular topics will be analysed in the next sections, as they are tightly linked to the software development of the main contribution of this dissertation.

Limited Preemption

As briefly explained earlier, limited preemption systems present a behaviour in the middle of the space stretched by fixed priority preemptive scheduling (FPPS) and fixed priority non-preemptive scheduling (FPNS). This kind of systems, and their analysis, are important as implementations of synchronization primitives and other resource sharing concepts rely on executing of procedures without the interference from other tasks. This behaviour is commonly referred as atomic. From multiple strategies available in the literature, this section focuses on two different approaches to the problem of providing a trade-off between FPPS and FPNS.

The first related strategy is based on a dual priority system that schedules tasks based on their classic priority, and later elevates their priority to the “priority threshold” upon execution. This strategy for assigning priorities allows to control the “preemptability” of tasks, as only procedures with higher priority than the priority threshold are able to suspend the running task [30]. This model is equivalent to the normal FPPS when the priority threshold and the classic priority are the same. Equivalently, when the priority threshold is set to the highest priority level, this model behaves as a system based on FPNS.

The schedulability analysis for this model is an extension of the framework available for FPPS and FPNS, which were briefly discussed in previous sections. This model is proven optimal in the sense that, if a set of tasks is schedulable by either FPPS or FPNS, it will also be schedulable by the priority threshold algorithm. In fact, it improves schedulability, as certain sets of tasks are only schedulable by this method [30]. Additionally, the model is fully extended to allow the integration of polling and sporadic servers to the analysis without constraints in the validity of the results.

However, finding of the optimal “priority threshold” values based on a given set of fixed priorities is non-trivial. Additionally, the size of the search space for obtaining the optimal set of fixed priorities and preemption thresholds from a set of tasks requires heuristic approaches, which preclude the usage for online scheduling algorithms. Nevertheless, if the priorities are calculated offline, no runtime overhead is added to the system. Furthermore, this schedulability system leads to less than or equal number of context switches when compared to FPPS [30]. This property effectively improves performance and decreases temporal interference from the kernel to the user tasks.

Another approach to the problem of limited preemption is to represent each task as a series of sub jobs that are non-preemptable. In other words, preemption for a particular task is only possible at certain well defined points. This kind of scheduling, formally known as fixed priority scheduling with deferred preemption (FPDS), provides multiple advantages. Such benefits include the reduction of the cost associated with arbitrary preemption, or the obviation of resource access protocols [31].

Although this kind of algorithms were proposed since the 90's, the maturity of the response time analysis was not sufficient to provide a correct worst case scheduling results. The correct solution arises from the use of a continuous scheduling model rather than a discrete one, as well as the distinction that the worst case response time is a supremum rather than a maximum for all tasks except the lowest priority one [31]. This novel analysis proves all precedent analysis to be either pessimistic or optimistic. In fact, the analysis for FPDS can be used for FPNS (which is considered a subcase), leading to situations where the worst caser response time is not present on the first instance of the task (which was a common assumption in the analysis of real time networks, such as CAN [32]).

Returning briefly to the subject of costs related to preemption, is important to distinguish between *arbitrary* and *voluntary* preemption. The term *voluntary preemption* was first used during the beginnings of the 90's to describe the points at which a program scheduled under FPDS is allowed to be halted [33]. In particular, this technique allows the reduction of the overhead cost related to context switching in comparison with *arbitrary preemption*. As preemption places can be selected by the user, they are chosen where the context to be saved is minimum. In practice, these points can be locations that avoid pipeline flushes, prevent the interruption of multicycle instructions, or reduce the number of registers that need to be saved (for example, obviating the necessity of saving the state of any coprocessor in architectures that include one).

Context Switch Improvement

As described on the previous section, the overhead generated by the context switch can be reduced by carefully selecting the processor conditions and memory elements that need to be saved. As cache memory is one of the main strategies to improve average latencies of modern processors, it is common that context switch improving techniques target it. In particular, they focus on strategies that prevent cache misses and their side effects, such as pipeline flushing [34]. However, as the target processor architecture for this work is not cache-enabled, these strategies are considered to be out of the scope of this review.

One effective strategy is to reduce the set of registers that need to be saved by locating instructions where the registers that need to be saved is minimum. These places, called *fast context switch points*, emerge from an analysis of register properties such as liveness [35]. In general, registers in a processor can be distinguished between user allocable registers and system registers. The latter always need to be saved, as they contain data related to the internal processor state, such as the stack pointer, the program counter, or the link register. In turn, the user allocable registers only require to be saved in case they are *live* at the time of preemption. A register is considered to be *live* in case it contains data that will be used in the future, and is considered to be *dead* in any other case.

The process of calling functions within the code provides additional insight about the liveness of registers. In general, the hybrid caller convention is used to distinguish the program section that has the responsibility of saving registers, either the caller function or the callee function. From one hand, caller-save registers are scratch registers that should be saved and restored if they are live across a function call. From the other hand, callee-save registers are non-scratch registers that are saved and restored if they are used within the function. As non-scratch registers not used within a function need interprocedural analysis to determine if they are live, a *fast context switch point* is defined as a program section where no scratch registers are live [35]. Additionally, the ARM processor used in this work follows the same convention for caller / callee saved registers [36].

The back-end of a compiler can be optimized to generate more locations for fast context switch in a method called register remapping. This technique reorganizes the use of registers without decreasing the overall performance, by mapping scratch registers to non-scratch registers that are not used within the live range of the scratch register [35]. A similar approach optimizes a compiler to minimize the context size, and then propagates this information back to a statically generated OS. This OS then generates special code that is able to context switch these sets of variable number of registers [37].

3.2 Synchronization in Real Time Systems

The roots of the synchronization theory can be attributed to Edsger W. Dijkstra. During the 60's, he conceived the idea of the semaphore and made the special distinction for the mutex, which is a special binary semaphore targeted to solve the mutual exclusion problem [38]. Since then, the semaphore concept has found extended applicability in the design of OS and the implementation of real time systems. However, the usage of semaphores without special cautions frequently leads to problematic situations, such as deadlocks when accessing multiple resources, or cases of priority inversion which violate the basic premises of any priority based system.

As a consequence, synchronization protocols have been designed to prevent the catastrophic consequences that hard real time systems failures can lead to. These protocols are a set of implementation guidelines as well as extensions to the schedulability analyses. Three of the most widely used protocols will be described in the text below.

Priority Inheritance

The Priority Inheritance Protocol (PIP) consist of dynamically adjusting the priority of a task holding a resource to the highest priority of any other task waiting for that resource, as long as this priority is higher than the original one. Upon release of the resource, the priority of the task is restored to its original value [39]. As a consequence, this protocol solves the problem of the priority inversion. Additionally, it is important to highlight that the priority elevation occurs when the resource is accessed by the higher priority thread.

The Priority Inheritance scheme is widely used in RTOS's, as its implementation is transparent and straightforward. However, this protocol has multiple drawbacks. First, it does not prevents deadlock. Second, chained blocking is still possible, although it is limited. Third, the calculation of the maximum blocking time for schedulability analysis is relatively complex, which prevents its usage for online scheduling decisions.

Due to the transparency of this protocol, it can be extended to decrease the occurrences of context switches and their related cost. In particular, embedded applications can be analysed during compile time to find the specific order locks will be acquired. This is possible as real time embedded systems are commonly constituted of cyclic threads that acquire resources always in the same order. With that information, the scheduler can prevent the activation (and related context switches) of threads that will block on resources which are not available [40]. This extension does not decrease the original schedulability or response time of the threads compared with simple PIP. However, in case the locking order is decided at runtime, this extension is not implementable.

Priority Ceiling and Stack Resource Policy

The Priority Ceiling Protocol (PCP) was designed to overcome the problems of the PIP. In particular, it addresses the problems of deadlock formation and chained blocking [39]. This strategy ensures that a task can only execute its critical section when its priority is higher than all the other preempted critical sections. If this is not the case, the task is blocked and inherits its priority to the task that caused the blocking. In order to achieve this behaviour, each resource (mutex) has a value referred as priority ceiling, which is the highest priority of all the tasks that could possibly lock that resource.

A runtime system designed to implement this protocol is aware all the time of the allocated resource (r^*) that has with the highest priority ceiling on the system (p^*). A task is only allowed to enter its critical section in case its priority is higher than the priority p^*. In any other case, the task is blocked and inherits the priority of the task that holds r^* (not the priority p^*). In case a task generates blocking to multiple tasks without releasing the resource, the priority of those tasks is transitively inherited.

Apart from avoiding deadlock and chained blocking, PCP ensures that a task can be blocked at most by one critical section of a lower priority task. However, this protocol has some disadvantages, such as the necessity of calculating the priority ceiling for each resource (mutex) beforehand, or the generation of *ceiling blocking* [39]. From one hand, the priority ceiling calculation is frequently done during the design phase, which requires constant maintenance and analysis of system schedulability, which prevents its online, dynamic calculation. From the other hand, *ceiling blocking* is a kind of blocking not present on other schemes. However, the usage of the protocol reduces dramatically the worst case blocking time of tasks.

In the same way PIP can be extended for reducing the number context switches, a strategy known as *priority ceiling preemption protocol* extends PCP for the same purpose [41]. In general, the threads need to be analysed, either at compile time or at runtime, for detecting the next resource to be acquired. The scheduler activates tasks not only based on the global ceiling priority, but also on the parameters of the resource that will be immediately locked by the new thread. Additionally, the schedulability is not only maintained, but improved. In particular, the worst case response time of certain tasks can be improved. However, the analysis required at runtime decreases slightly the overall improvement of this protocol extension.

The Stack Resource Policy (SRP) scheme is almost identical to PCP, with the only difference that the blocking is enforced at the time the task is activated and selected to execute (preemption), in comparison with blocking on resource access. As such, this property makes SRP adequate not only for FPPS, but also for algorithms with non-preemptable sections, such as FPDS or FPNP. In general, offers the same advantages of. An additional advantage of SRP is that tasks can share the same stack, as it was designed to enable reduction and control of RAM for constrained automotive applications [42].

Chapter 4 – Ell-i Development Environment

This part of the thesis discusses the material that served as a basis for the development of the scheduling system. First, a brief summary of the Arduino ecosystem is provided, as it was the basis for the development of the Ell-i prototype PoE node. The next part focuses on this node, the Ell-duino development board. The following sections discuss the software basis, which includes the Arduino compatible Ell-i Runtime, and the publicly available code from Open Source and proprietary RTOSs.

4.1 Arduino Ecosystem

Background

The Arduino project focuses on providing to users without technical expertise related to electronics (mainly hobbyists, students, designers and artists) a platform of inexpensive devices capable of interacting with their surroundings [43]. It started as a project for providing students with cheap and accessible developing boards at the Interaction Design Institute Ivrea, in Italy [44]. Although it was originally based on 8 bit AVR architecture MCU's, the steady decrease in price of 32 bit ARM MCU's leaded to new versions based on the Atmel implementation of the ARM Cortex-M core.

The Arduino boards are supported by an Opens Source, cross-platform, Integrated Development Environment (IDE) written in Java [45]. This IDE is based on the one used by the Processing project [46], which was also targeted to hobbyists and artists. The Arduino IDE also integrates a special library targeted to simplify the access and control of input / output peripherals. This library is based on the Wiring project [47], which was also initiated at the Ivrea Institute, and is also partially based on the Processing project. It is important to highlight the idea behind Processing, which was to allow artists to "sketch" ideas through code [46] (Actually, the idea behind Wiring is just slightly changed to "sketching with hardware" [47])

The Arduino project uses a reciprocal license for the OSHW boards, the Creative Commons Attribution-ShareAlike [48], which enforces that every modification is released with the same license, and gives credit to the original developer. From the IDE and OSS side that interacts with the hardware, the GPL reciprocal license is used for the Java environment and a slightly different LGPL reciprocal license for the MCU libraries [49]. The LGPL library is special as it only enforces reciprocity (release with the same license) on changes made to the open part. Any other component can be licensed according to the requirements of the developer, as long as there is a clear distinction between the open modules and the modules with different license attributes [50].

However, the main revenue channel for this company is based on the commercialization of their trademark. This is the most prominent example of the application of the branding model discussed in the Open Source Hardware Business Models section. Apart from ensuring the quality of the produced boards, which has been a strategic advantage against competition [51], the business model has been very profitable, generating a revenue of more than one million USD by 2010 [52, 53]. In detail, they charge to manufacturers a general license fee, and an additional royalty charge of 10% over the wholesale price [54].

Unfortunately, the evolution of the electronics industry has posed transition problems for the Arduino team. Apart from the fact that ARM owns the largest share in the embedded 32 bit market since the

2000's [55], the general trend points toward the eventual adoption of energy efficient 32 bit processors in the place of 8bit and 16 bit counterparts [56]. In fact, ARM launched the Cortex-M0+ architecture targeting the 8 bit market, commonly driven by low cost and very low power consumption [57].

This trend is particularly problematic for the Arduino project lifetime, since the majority of their sold products and professional expertise is based on the 8 bit AVR architecture. Although they produced the first board based on ARM 32bit architecture (Arduino Due, Cortex-M3) in 2012 [58, 59], and their second (Arduino Zero, Cortex-M0+) in 2014 [60], the transition is far from being complete. Apart from maximum voltage differences that rendered the majority of external shields incompatible with newer boards, the software adaptation is still a work in progress, for both, the Arduino team, and all the third party providers.

The requirements for internet connectivity for IoT applications has generated additional pressure for the Arduino project. Although they have provided extensions (in the form of shields) to their boards to enable wired [17] or wireless [61] Internet connectivity, the overall cost of a board plus a shield is far from optimal. They proposed a board with integrated wired Internet access [18], although the price of this board is almost equivalent to price of the board plus shield packet as of 2014 [1] [62, 63].

As discussed earlier, the lack of expertise on 32 bit architectures is forcing the Arduino team to take design decisions which optimize the time to market, at the cost of a steep increase in the prices for 32bit based devices. Such devices include boards based on newer MCU's, network enabled systems, and Power over Ethernet (PoE) applications. In particular, the choice of Ethernet controllers [17, 18] and WiFi modules [61, 64] with integrated network stacks, or the integration of closed source components for PoE support [17, 18], make evident that the market trends are moving the team out of their locus of expertise.

Programming Model

The Arduino project uses a simplified program template called "sketch". In its most basic form, a sketch is composed of two C-like functions with no arguments or return values: the "setup" function, which is run only once when the board is powered up (or after a reset), and the "loop" function, which is cyclically run until the board is powered off (or until a reset is generated). Additional functions from the Arduino libraries, standard C libraries, libraries provided by 3rd parties and user defined functions, can be used to generate the desired behaviour. As the program defined as "sketch" is not fully C or C++ compliant, it requires additional steps before being provided to a standard C/C++ build system.

This modification target mainly three objectives. The first is to generate prototypes for the user defined functions. However this process is not perfect, as it is unable to generate declarations for functions defined within particular namespaces or classes, or with default argument values [65]. The

[1] For a fair comparison, the *Uno* and the *Ethernet* board use the same AVR Atmega328 MCU at 16Mh.
Without PoE support:

- Arduino Uno Board (€20) + Ethernet Shield (€30) =	€50
- Arduino Ethernet Board (€40) =	€40

With PoE support:

- Arduino Uno Board (€20) + Ethernet Shield with PoE (€45) =	€65
- Arduino Ethernet Board with PoE (€55) =	€55

second objective is to add all the necessary library references. This step adds a reference for the "Arduino.h" library (or to the "WProgram.h" library for legacy code based on the original Wiring project). The third and last step consists on appending a "main" function that matches the target board. The specific options that this code transformation requires are defined in the XML files included with the Arduino IDE distribution. For clarity, Snippet 1 describes a simplified output of the code transformation for a sample sketch.

Original "Sketch"

```
int led = 13;

// the setup routine runs once when you press reset:
void setup() {
  // initialize the digital pin as an output.
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  digitalWrite(led, HIGH);   // turn the LED on
  delay(1000);               // wait for a second
  digitalWrite(led, LOW);    // turn the LED off
  delay(1000);               // wait for a second
}

// user defined function
void my_function() {
  //do something
}
```

Transformed "Sketch" (correct C code)

```
#include "Arduino.h"

int main(void);

void setup(void);
void loop(void);

void my_function(void);

int led = 13;

void setup() {
  pinMode(led, OUTPUT);
}

void loop() {
  digitalWrite(led, HIGH);
  delay(1000);
  digitalWrite(led, LOW);
  delay(1000);
}

void my_function() {
}

int main() {
  setup();
  for (;;)
    loop();
  return 0;
}
```

Snippet 1. Arduino Sketch Pre-Processing

After the transformation, the sketch code is a fully compliant C/C++ program. Then, it is compiled and linked by a standard build process. As the Arduino project initiated with boards based on the 8 bit AVR architectures, the code has been always compiled and linked with the latest AVR GCC toolchain distribution. As new 32 bit board models also use the ARM core processors manufactured by AVR, the same toolchain is used for both, the 8 bit boards and the new 32 bit boards. As a consequence, additional functions supported by the AVR GCC libraries can be used within the Arduino IDE.

The main advantage of the Arduino software system is the availability of function libraries that effectively simplify the interaction with the underlying hardware. This property enables its use by any person without knowledge of the electronics field. In other words, the Arduino system provides a layer of abstraction that reduce complex operations to single statements. As a generality, the libraries are focused on some kind of input / output process, such as network or serial communication, display handling or motor control.

A useful example of this simplification is the abstraction of the configuration of a PWM output. Depending on the architecture, a pin which outputs a PWM pattern usually reflects the interaction of at least two peripherals: a GPIO controller and a Timer. If the MCU supports some sort of clock gate control, the initial step is to enable the gate to the required peripherals. Then, the timer is configured for PWM operation, which commonly involves frequency, counting thresholds, and output settings. Finally, the GPIO is configured to be controlled by the Timer and output the desired pattern.

In contrast, the Arduino system silently enables and configures the desired peripherals before the code from the programmer is executed. Then, only two actions are required from the user: one that redirects the output from the Timer to the desired GPIO, and another that selects the desired threshold for controlling the PWM duty cycle. This operations are elegantly wrapped by functions that only request from the user the relevant parameters, and realize all the calculations and register accesses on behalf of the programmer.

As an important reference, the summary of the functions provided by the Arduino IDE is presented in the tables below (Table 2 and Table 3). It is important to note that there is a subset of relatively advanced features only available for the more powerful, ARM 32 bit based boards (and some boards with USB support).

Only Core Distribution, Available on all Boards	
Functions (Based on [66])	***Libraries*** (Based on [67])
I/O • Digital I/O (3) • Analog I/O (3) • Advanced I/O (5) Time • Time (4) Data • Conversion (6) • Bits and Bytes (7) Math • Math (7) • Trigonometry (3) • Random Numbers (2) Interrupts • External Interrupts (2) • Interrupt Control (2)	Base Classes (1) • Stream Communication (5) • Firmata • Serial • SPI • SoftwareSerial • Wire Networking (3) • Ethernet • GSM • WiFi Memory (2) • EEPPROM • SD Display (2) • LiquidCrystal • TFT Motor (2) • Servo • Stepper

Table 2. Arduino Functions, All Boards

Only Core Distribution, Available for Due	
Functions (Based on [66])	***Libraries*** (Based on [67])
I/O • Analog I/O (1)	Available for Due (and other selected boards) • Audio • Keyboard • Mouse • USBHost • Scheduler

Table 3. Arduino Functions, Available for Due

4.2 Ell-duino Development Board

The first starting point for the development of the scheduler presented on this document was the Ell-duino board. It is a development board pin compatible with the Arduino Due and with off-the-shelf support for Power over Ethernet, and Ethernet communication in general. It features a Cortex-M0 based MCU from STMicroelectronics, the STM32F051R8T6. This MCU is targeted to low cost applications, and features a limited amount of Flash and SRAM memory as the principal cost reduction constraint.

ARM Cortex-M0 Processor

The ARM Cortex-M0 is the low end core from the ARM Cortex-M family (which is oriented to be used in the fabrication of microcontrollers). It is targeted to be implemented in a small die area, in order to enable low price and low power MCUs and SoC applications. It features a 3 stage pipeline, without any kind of cache memory or branch prediction. This core implements the Von Neumann ARMv6-M Architecture [68], which is a subset of the more powerful Modified Harvard ARMv7-M Architecture [69] used for the Cortex-M3 and Cortex-M4 counterparts (the architecture implemented in the M4 core sometimes called ARMv7E-M, which only denotes small extensions from the ARMv7-M).

Among the differences between the v6-M and the v7-M architectures, the most relevant for the implementation of an RTOS are the limit of priority levels of the interrupt controller (NVIC in ARM terms), and the lack of instructions for counting leading or trailing zeroes. From the NVIC side, only 2 MSB bits are used for the priority register bytes, which leads to only 4 priority levels for the v6M architecture in comparison with the 256 available for the of the v7-M architecture (apart from the system interrupt priority levels). Additionally, the absence of hardware support for counting leading zeroes, CLZ instruction, is a direct consequence of the very limited support for the Thumb-2 Instruction set (in contrast, the Thumb instruction set is almost completely covered).

For other kind of OS applications, which require memory or privilege management, the Cortex-M0 core might pose certain limitations. From the memory management side, the M0 core does not support the Memory Protection Unit (MPU), which is an optional component for the M3 and M4 cores. The underlying reason is that the v6-M architecture does not implement any kind of privileged operations, or different execution privilege states. As a consequence, any privilege management or protection would require a software implementation. However, the Cortex-M0 core does have the possibility of having a Bit Banding system, which enables atomic operations for setting and clearing specific bits of words in that region, instead of the standard read-modify-write operation, which can be problematic for concurrency.

In particular, the Ell-duino board uses the STM32F051, one of the STMicroelectronics implementations of the Cortex-M0 core. The specific MCU is the STM32F051R8T6, which operates at the range of 2.0 to 3.6 V and has 8 Kbytes of RAM and 64 Kbytes of FLASH [70], apart from a wide range of peripherals. However, as this board is for prototyping and not for final production, the memory available in the production MCU would be more limited, as well as the set of available peripherals. The maximum operating frequency of the system is 48 MHz, what leads to an instruction cycle time of 20.833 ns when operating at maximum speed. However, some peripherals are clocked at subdivisions of the main frequency.

In particular, the STM32F0 family has a Von Neumann architecture, as the same system bus is used to access both program memory (Flash) and data memory (SRAM). A general scheme of the architecture is shown in Figure 1 (based on Figure 1 (MS32128V2) from [71]). It is important to highlight that the bus network of the chip is administered by the Busmatrix, which has two masters: the Cortex-M0 core and the DMA Controller. Four slaves buses are connected, one for each memory (Flash and SRAM), and two AHB Buses. The fact that the AHB2 bus directly connects the GPIO modules with the Busmatrix effectively decreases I/O latency. In comparison, all the other peripherals are connected through the APB bus to the AHB1 by a bridge controller.

One important advantage of the Von Neumann architecture implemented on the Cortex-M0 core is that program memory can be accessed as only readable data memory. This characteristic enables the possibility of storing constant values in the Flash, and access them during runtime without the necessity of copying them to RAM beforehand. This particular property is used by the initialization routines further described in section Peripheral Initialisation. Another important property is that all the peripheral registers are memory mapped, so they can be accessed by normal read and write operations from the standard C language.

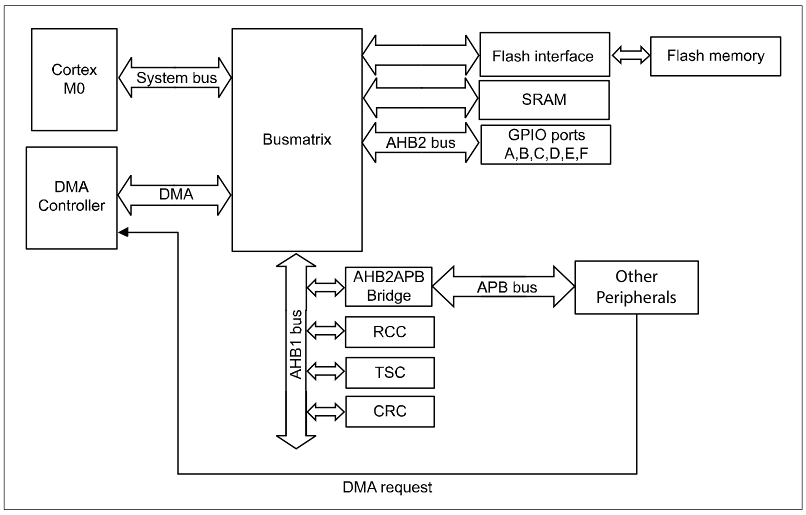

Figure 1. General STM32F0 Architecture

PoE and CoAP Support

The main differentiating characteristic of the Ell-duino board is the off-the-shelf support for Power over Ethernet and normal Ethernet communication. In particular, the board includes all the power electronics and passive components for extracting energy from the PoE link to power itself and the devices connected to it. Additionally, the board can be powered from an external DC power source or take power from the serial connection. Internally, there is a power management circuit that enables multiple power sources concurrently without interference (known as Power Domains).

An intrinsic advantage for a system within a PoE network is that it is not energy constrained [72]. In fact, the latest standard for regulating the PoE operation [73] specifically requires that a Powered Device (PD) provides a Maintain Power Signature (MPS) for the power supplier in order to prevent disconnection. The MPS is defined as the drawing of at least 10 mA of current for a period of at least 75 ms followed by an optional cease of current demand of maximum of 250 ms (effectively meaning drawing current for 75 ms every 325 ms). Additionally the device should keep an impedance within the range of maximum 26.3 kΩ and minimum 50 nF.

As a consequence, if a PD keeps constantly the MPS minimal state (without intermittent 250 ms dropout) would consume a minimum of 425 mW for a type 2 PD (minimum 42.5 V), or 370 mW for a type 1 PD (minimum 37.0 V). As a comparison, the MCU that composes the Ell-duino board has a typical power consumption at its maximum frequency of (48Mhz) of 79 mW. For comparison, the maximum power dissipation for the STM32F051R8T6 is of 444 mW (when operated at its maximum temperature limit of 85 °C) [70]. In conclusion, the Ell-duino board software and hardware design is not power constrained, as the minimal power provided by the PoE link is one order of magnitude higher than the average consumption of the main components.

Regardless of the connection to a PoE link or to normal Ethernet, the board filters the data signals to an ENC28J60 Ethernet Controller directly connected through SPI to the STM32F0 MCU. As an strategy for enabling ample network applications and keeping the cost of the board at a low level, the Ethernet Controller does not contain an integrated IP stack (in contrast to the more costly controllers with IP stack included present on all Arduino network enabled boards). This controller is manufactured by Microchip and supports up to the 10 Mbit/s speeds [74]. It implements a 10BASE-T physical layer and a MAC layer, which features a programmable packet filtering system.

As the Ell-duino board is intended for Machine to Machine applications (M2M), the integrated software stack is planned to support the Constrained Application Protocol (CoAP). The CoAP is an application layer protocol primary targeted to enable the REST architectural style on constrained networks and constrained nodes (with limited program and data memory) [75]. In concrete, it was designed with the primary objective of having very small packet overheads and specifically to work on top of UDP for preventing packet fragmentation as much as possible (due to the intended use in slow and lossy networks like low power WPANs).

CoAP implements a 16 bit Message ID for packet identification and four kinds of message types: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK), and Reset (RST). The general idea is that CON messages are sent and resent with an exponential back-off delay until an ACK message with the same Message ID is received. On the contrary, NON messages are sent without expecting any response, except when the receiver sends an RST message with the same Message ID that indicates no more messages should be sent until further notice.

For overhead reduction, a binary 32 bits header is used. This header contains five fields, being the Message Type and Message ID the most relevant for the communication handling. In particular, the Message ID is not only used for tracking message responses but also to prevent any duplication. As a consequence, the minimum size of a CoAP message is 4 bytes, and the maximum is directly related to the maximum payload size of the UDP frame. As fragmentation is a primary concern, in the cases where it is not possible to confirm the maximum payload size, the standard defines some default maximum sizes that should be enforced.

Arduino Compatibility

The Ell-duino board is targeted to be hardware and software compatible with the Arduino project. In concrete, it is pin and voltage compatible with the Arduino Due (which features an AVR manufactured

Cortex-M3 based MCU). In general, the only limiting characteristic for the hardware compatibility is that the selected pin can be connected to a peripheral that provides the same functionality, such as timers PWM output or ADC analog input. This characteristic provides ample liberty on processors that can internally multiplex different functions for a particular external pin. For the STM32 family, the GPIO module can connect some pins of Port A to a maximum of 8 different functions, or some of Port B to a maximum of 4 functions [70].

However, software compatibility is more complex in case the MCU is used to support concurrently any additional processing load apart from the Arduino sketch flow of execution. The main problem is that, historically, the Arduino sketches have had full control of the processor. However, that constraint can be relaxed owing to two reasons. First, only a few Arduino APIs have real time constrains, or equivalently, can be used to provide real time behaviour. As a matter of fact, only a limited subset of activities effectively require precise timings: mainly communication with other devices connected through protocols with strict timings, like SPI. Second, traditionally the MCUs for the Arduino boards are low cost 8 bit AVR with frequencies in the range of 16 MHz (with the exception of the Arduino Due) [76]. This processors have effective processing power of only a fraction of what the Cortex-M0 can deliver, as a result of higher frequencies and the 32 bit architecture.

As a consequence, software compatibility is defined as the ability to run a fair number of real-time Arduino sketches, with the Internet connectivity in the background. In practice, a limited number of functions (mainly the time functions with microsecond precision [66]) and sample sketches will require modification in order to function properly on a multithreading environment. Additionally, other activities, such as motor control, can be executed concurrently, as the Ell-I Runtime multithreading support will be mandatory for handling the network connectivity and the Arduino sketches at the same time. In practice, the maximum number of threads defined by the Ell-i Cooperative for the Ell-duino board is eight.

4.3 Provided Runtime

The second starting point for the present work was the Runtime code targeted for the Ell-duino board. In concrete, the code is targeted to the Cortex-M0 core and the peripherals present on the STM32F0 family of MCUs. The main differentiation point of this runtime, which inspired part of the scheduler implementation, is the strategy for initializing peripherals based on constant data structures stored in the program memory. Additionally, the code for realizing a context switch assisted by hardware was also present in the given runtime as part of a work-in-progress round robin scheduler.

Peripheral Initialisation

As memory is among the most important pricing factors for MCUs (due to the slowing of Moore's law), the Ell-I Runtime presents a novel strategy for initializing peripherals with a static approach, inspired on the previous work realized by Nikander et. al. [77]. This strategy represents the final desired state of peripheral registers as data, instead of steps of changes instructed by the program. It saves 30% to 40% of program memory compared to the classic imperative approach [78].

The concrete implementation takes advantage of the features of the GNU linker that enable locating program objects in specific sections of memory. Then, due to the Von Neumann architecture, this sections of the program memory are treated as only-readable data, and iterated to obtain the desired final peripheral register states without first copying to RAM. To assist the iteration, the limits of this memory location are transferred from the linker back to the C code as part of the linking process. The details are shown in Snippet 2 and Snippet 3.

```c
typedef struct {
    const enum system_init_r_type   init_record_type;    // Type of SystemInitRecords in the union
    const uint8_t                   init_record_number;  // Number of SystemInitRecords
    const union {
        const int32_t               init_record_offset;  // Offset to be added to the addresses
                                                         // in the SystemInitRecords
        volatile preg16_t *const    init_record_address16; // Base register address for 16_only
                                                           // or 16_with_offset
        volatile preg32_t *const    init_record_address32; // Base register address for 32_no_address
    };
    const union {
        const SystemInitRecordAddrAndOnes *        init_records_addr_and_ones;
        const SystemInitRecordAddrOnesAndZeroes *  init_records_addr_ones_and_zeroes;
        const SystemInitRecordData16Only *         init_records_data16_only;
        const SystemInitRecordData32Only *         init_records_data32_only;
    };
    const union {
        const SystemInitRecordRegisterOffset *     init_records_register_offsets;
    };
} SystemInitRecordArray;
```

Snippet 2. Peripheral Initialisation Data Structures

C Initialisation Code

```c
extern const SystemInitRecordArray __peripheral_start[];
extern const SystemInitRecordArray __peripheral_end[];

void SystemInitPeripherals(void) {
    for (register const SystemInitRecordArray *ir = __peripheral_start;
         ir < __peripheral_end;
         ir++) {
        SystemInitFunctions[ir->init_record_type](ir);
    }
}
```

Linker Script

```
. = ALIGN(4);
  __peripheral_start = .;
  KEEP (*(.peripheral.RCC*))      /* Read-only RCC initialisation data */
  KEEP (*(SORT(.peripheral.*)))  /* Read-only peripheral initialisation data */
  __peripheral_end = .;
```

Snippet 3. Peripheral Initialisation Code and Linker Script

Context Switch Prototype

The ARM Cortex-M cores save part of the stack frame on exception entry and restore it in exception return. In particular, the processor saves the scratch registers together with the processor state related registers. The previous strategy partly allows the usage of standard C functions as exception handlers [36]. The hardware partial context save process is more efficient than its software equivalent, taking only 15 cycles for the exception entry and 16 for the exception return in the Cortex-M0 STM32F0 processor (for more details on interrupt handling latencies, please consult Appendix A - Measurement of Hardware Exception Handling Times). Consequently, it is advisable to assist the context switch by taking advantage of the hardware saved registers and simply complete the missing registers for the full stack frame.

The Cortex-M processor implements the ARMv6-M architecture, which has a relatively advanced interrupt controller (NVIC) [68]. It allows multiple interrupt priorities, pre-emption and nesting. Therefore, there are multiple possibilities to arrange the priorities of the system interrupts. The scheme suggested by ARM [79] for assisting context switch consists of leaving the system asynchronous interrupt, PendSV, at the lowest possible priority level. As a result, it can execute after all the other interrupts and exceptions have been served. The advantage is threefold: it does not generate interference to higher priority interrupts, it can assist the context switch after a higher priority interrupt has generated some synchronization mechanism that requires the change of current running thread, and it benefits from the tail-chaining mechanism.

The initial prototype of context switch provided by the Ell-i Co-Operative used the suggested approach of setting PendSV at the lowest interrupt priority. Three steps are required in order to execute the context switch from the PendSV context. First, the part of the context not saved by hardware is pushed to the stack. Second, the stack pointer of the switched out thread is saved and the one from the new thread is loaded. The third step consist of popping the software saved registers. Finally, the exception return is requested, loading the rest of the saved context efficiently by hardware.

The context switch prototype also takes advantage of the two stack pointers available for the ARMv6-M architecture. The two stack pointers, main stack pointer (MSP) and process stack pointer (PSP), are banked and only one is visible at a time [68]. Although there is no restriction for using either MSP or PSP on Thread mode, the system will automatically switch to MSP when it enters Handler mode. Therefore, selecting the PSP to be used only by Thread mode and reserving MSP only for exceptions provides two advantages. First, the stack space can be isolated for each thread and the exceptions, allowing more detailed control and analysis of the usage and reducing considerably the stack space required by each individual thread. Second, when realizing the context switch from Handler mode, the PSP will remain intact and it will be possible to realize operations on it without affecting the current execution that depends on the MSP.

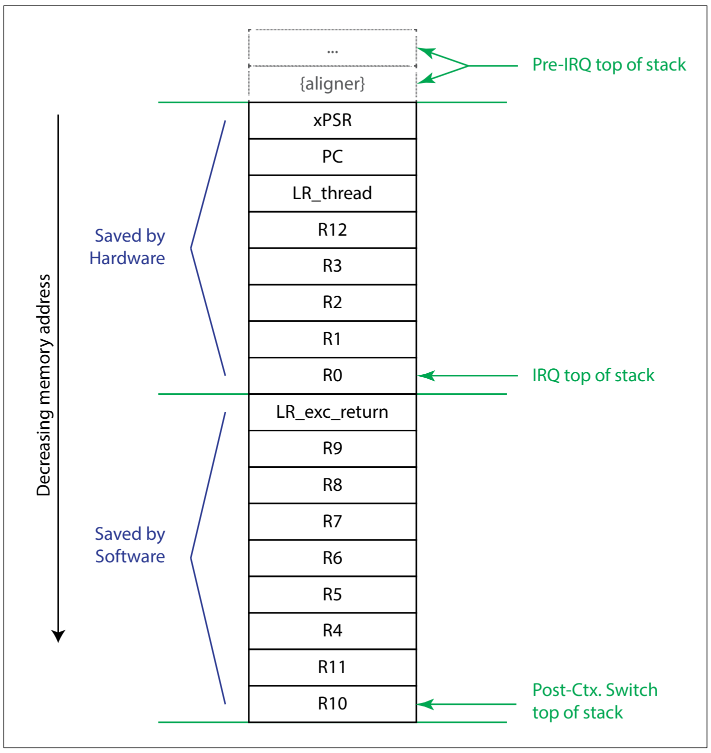

Figure 2. Context Switch Prototype Context Frame

4.4 Other (Open Source) RTOS

The third starting point was the existing implementations of RTOSes available for the Cortex-M0. Both, Open Source and Proprietary systems were considered. From the Open Source side, ChibiOS/RT [80] was suggested by the Ell-i Co-Operative as the preferred candidate for its coding style, reduced memory footprint, and speed. FreeRTOS [81] was also briefly analysed for implementation details. From the previous experience of the author, the proprietary Micrium uC/OS-II [82] and uC/OS-III [83] were also considered as a starting point as their source code is available for academic purposes, and it is accompanied by very detailed documentation in the form of books.

The first significant difference that was found among the implementations was the computational complexity of the scheduling operation. While the RTOSs from Micrium perform the scheduling in constant time[2], the analysed open source counterparts, ChibiOS/RT and FreeRTOS provide linear complexity. Nevertheless, although the constant scheduling time provides scalability, the specific implementation for the Cortex-M0 is not efficient for a small number of tasks when compared to the linear alternatives. The main problem lies in the lack of the Count Leading Zeroes (CLZ) instruction in the ARMv6-M architecture. As a result, the look-up tables used for providing equivalent functionality increase the memory footprint and related execution time.

The second relevant distinction between the reviewed implementations was the execution space where the context switch is performed. FreeRTOS and the systems from Micrium execute the context switch from the PendSV exception at the lowest priority level. In contrast, ChibiOS/RT performs the context switch in the program space that is actually executing (either thread mode or in handler mode), and only calls for the Non Maskable Interrupt (or PendSV as an alternative) for completing a small part of the context switch when executing from handler mode. As the provided context switch prototype was optimized for using the PendSV handler, that alternative was selected for the scheduler implementation.

	ChibiOS/RT	***FreeRTOS***	***uC/OS-II***	***uC/OS-III***
Scheduling Complexity	O(n)	O(n)	O(1)	O(1)* *Case with limited threads shown for fair comparison. With unlimited threads, O(n).
Data Structure	Queue	Queue	Bitmap	Bitmap
Context Switch Space	From Thread mode: -Thread Mode From Handler Mode: -Handler + NMI -Handler + PendSV* *optional, preferred NMI	PendSV	PendSV	PendSV

Table 4. Scheduler Details for the Initial Surveyed Implementations

[2] In practice, Micrium uC/OS-III implements a two-step search process (first step with linear complexity, second with constant) because of its ability for handling unlimited priorities. For an equivalent comparison, if the number of priorities was fixed, the two-step search process could be optimized for constant complexity, in the same way it is done in Micrium uC/OS-II.

Previously Published Benchmarks

In order to have a general overview of the performance of Open Source RTOSs, previously published benchmarks are also analysed for related results. The study realized by Otava [84] selects three systems: CooCox CoOS [85], uc/OS-III and FreeRTOS. The selection was made with the basis if comparable set of features. The main analysis is for OS time overhead and was performed on a Stellaris LM3S8962 from Texas Instruments that feature a Cortex-M3 core at 40 MHz. The code from each system was compiled with GCC 4.6.2 for Thumb 2 and each RTOS was scaled to its minimal configuration. In concrete, two tests were performed: one for measuring context switch overhead and other to analyse the software timer jitter.

From the overhead side, the particular measurement comprises the context switch together with an object synchronization primitive. The author considers that this kind of operation is the most often executed for the implementation of systems on top of RTOSs. In particular, two synchronization objects are tested, mutexes and queues. They were measured by a hardware timer in the MCU clocked by an external source. However, as in this thesis there is no use of queues, the results for that primitive are considered out of the scope.

Two tasks were used for the testing, one with higher priority than the other. The measurement encompasses the overhead of the higher priority task blocking on an object owned by the lower priority task, and the subsequent context switch to that task. It is important to note that the authors do not measure the opposite event, when the lower priority thread switches back to the higher priority thread. Nevertheless, the results are considered valid and serve as a base for future comparison. They are presented in Table 5.

For the software timer jitter, the period is measured via a hardware timer clocked by an external reference generator. It is concluded that the implementation of the software timer is identical in the three implementations, and the deviation is caused by inaccuracies of the crystal oscillator.

	FreeRTOS	*CoOS*	*uC/OS-III*
Mutex Waiting (us)	11.59±0.001	15.51±0.010	14.13±0.005
Code Size (Bytes)	6928	7236	9864

Table 5. Previously published benchmarks for RTOSs performance

Another performance analysis for RTOSs is presented in the work from Ugurel and Bazlamacci [86]. They compare between the Xilkernel and uC/OS-II RTOSs sing a Xilinx Spartan 3AN FPGA with MicroBlaze v8 soft core running on top. Unfortunately, they are unable of measure precise timings for each processor. Instead, they measure number of switches during a fixed period of 30 seconds. Although the results are not comparable (as the system runs on top of a soft processor instead of a silicon MCU), the thread model strategy for comparison is the same as [84].

In detail, two tasks are present, one with higher priority than the other. Cyclically, the higher priority thread blocks on a semaphore owned by the lower priority, which immediately releases the semaphore for the higher priority thread. The results from this study conclude that Xilkernel is almost ten times faster than uC/OS-II. For completeness of results, they give detailed footprint information, divided by sections (text, data, bss, etc.). They also disable all non-related features (to bring the RTOSs to their minimal configuration), and consider that the loop that generates the context switches has negligible overhead.

Chapter 5 - Memory Constrained Scheduler

The chapter that follows moves on to describe the design and implementation details of the scheduler. First, the overall architecture of the system is presented. Then, the next three sections present the respective implementation details of the scheduler, semaphore, and time management subsystems. In particular, the scheduler section deepens on the context switch state space, which deeply depends on the hardware interrupt controller. The last section of the chapter deals with the memory management policies of the system, with focus on the integration to the existing runtime.

5.1 System Architecture

There are three relatively independent functional modules that compose the kernel: scheduler, semaphores and time management. The scheduler is the base component that enables the operation of the two other modules. There is also initialization code that configures the hardware and prepares the data structures for the rest of the modules, although it is not considered a functional module as such. The code for the modules is distributed among the various layers of the kernel. The kernel is structured as a three layered system: with the exception handlers at the bottom, the internal functions in the middle and the API for the user at the top (See Figure 3 and Figure 4). The API functions are the only ones supposed to be used by the application.

System Exceptions and Priorities

For the selection of priorities of the system interrupts, is important to highlight that The Cortex-M0 system only implements the two most significant bits of the priority registers. As a consequence, there are only four priority levels available for the interrupts and configurable priority system exceptions. Another relevant characteristic is that multiple exceptions can be registered at the same priority, although preemption is only possible between different priority levels. In other words, an exception is unable preempt another one in the same priority level [68].

As a result of the previous behaviour, the exceptions located at the highest priority level, band [0, 63] in Table 6, execute their code without interference of other exceptions. The effect is equivalent of disabling and enabling interrupts in thread mode or handler mode of lower priority exceptions. Although Reset, NMI and Hard Fault have higher priority that the band [0, 63], it is also not possible to disable them. Consequently, the behaviour is equivalent and the closest to an atomic operation available in the Cortex-M0 processor.

As the SysTick and SVCall Handlers modify sensitive data structures from the scheduler, locating them at the highest priority level makes the code executed from them inherently atomic. There is no problem of locating both of them at the same level, as it is not possible that they preempt each other. Altogether with the PendSV at the lowest priority band, the system effectively provide two priority levels completely available for user handlers.

Initialization

The initialization procedure, although not a functional block, is vital for the operation of the kernel. In particular, the initialization procedure not only configures hardware as in other RTOSs. In order to keep the Flash memory footprint as low as possible, the core data structures and stack spaces are created and initialized at runtime. Table 7 provides a concise summary of the executed operations

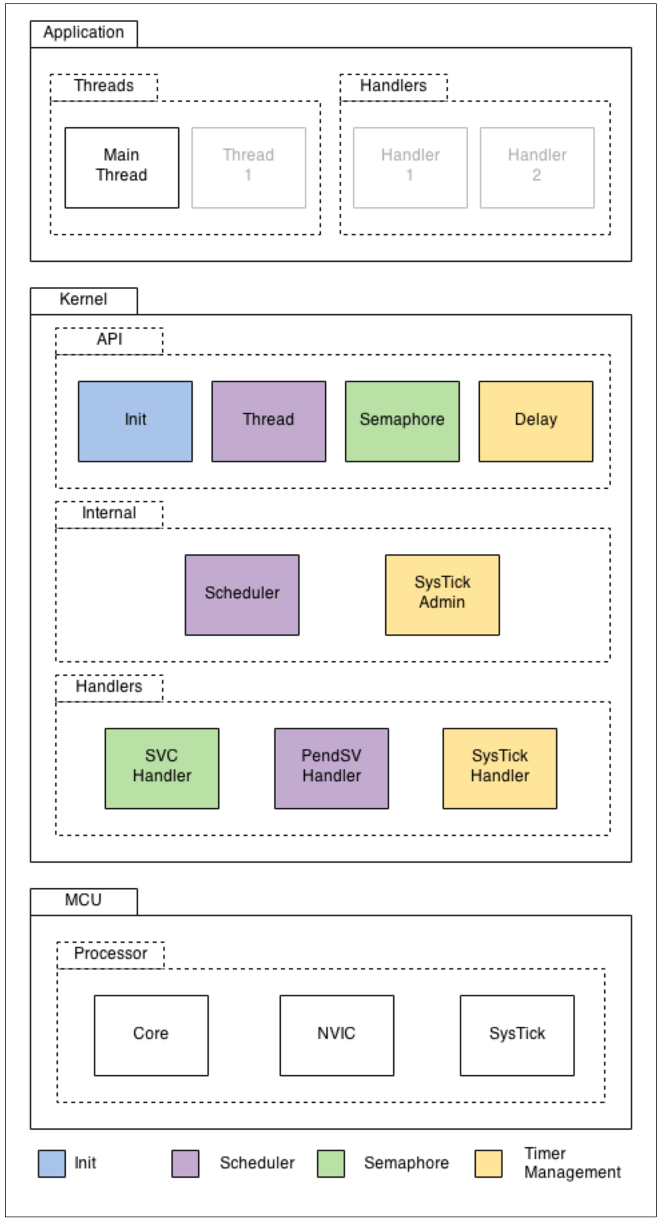

Figure 3. General Architecture

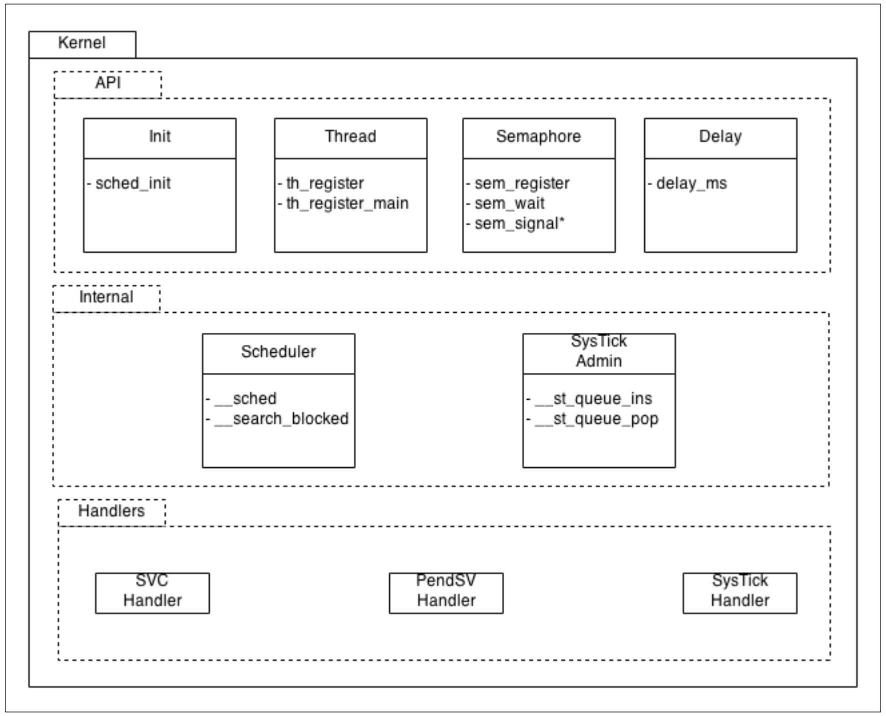

Figure 4. Kernel Detailed Architecture

** function callable from handler mode.*

during initialization. In particular, the decompression procedure of initialization records is different from all the other RTOSs analysed. Further details of this procedure are presented in Decompression Process.

Priority	Exception
-3	Reset (1) [async]
-2	NMI (2) [async]
-1	Hard Fault (3) [sync]
[0, 63]	SVCall (11) [sync], SysTick (15) [async]
[64, 127]	
[128, 191]	
[191, 255]	PendSV (14) [async]

Table 6. System Exceptions Priorities Arrangement

SP	- switch SP to PSP - point MSP to the interrupt stack
Thread	- decompress TCBs from initialization records - prepare stack space: - calculate limits - write sentinel values - prepare initial contexts - prepare pointers and flags
SysTick Admin	- prepare timeout queue
NVIC	- set exception priorities (SVCall, PendSV, SysTick)
SysTick	- configure frequency and enable - enable interrupt

Table 7. Summary of initialization operations

API

From the set of API functions, the functions that register threads and semaphores (th_register, th_register_main and sem_register) to the system are actually function-like preprocessor macros. The reason behind is the requirement of declaring and defining multiple variables in specially defined program sections at once. For example, for registering a thread it is necessary to declare and define with specific values the thread initialization register in the special “thread_init” section. Furthermore, it is also required to declare the placeholders for stack, TCB and timeout, each one in a particularly defined section (more details of the specific details can be found in the “Memory Management section”. The semaphore case is relatively similar: it is necessary to create a variable with specific values and a pointer to that variable at once.

Another important characteristic of the API, in particular of the semaphore waiting and signalling functions, is the ability to recognize if they are being called from thread or from handler context. This feature is added to prevent potential problems in the case the user mistakenly tries to wait for a semaphore from interrupt context. In that particular case, the request is simply discarded by the system. However, if the user tries to signal from interrupt context, what is purposely allowed for synchronization, the system handles the request accordingly. In particular, signalling a semaphore from handler mode leads to a complex set of alternative actions (analysed in section “state space”), when compared to the thread mode counterpart.

5.2 Scheduler

The scheduler provides the core functionality of the kernel, and is the base for the other modules (semaphore and time management). It is designed to provide the necessary functionality with simple data structures and reduced memory (RAM) footprint. The system provides a linear scheduler in the sense that calculating the highest priority ready thread has a worst case execution time proportional to the number of registered threads at the system. Although the implementation would not scale adequately with large number of threads, for the target of 8 threads, the worst case execution time is still comparable to other alternatives, with the advantage of reduced footprint.

In summary, the scheduler locates the highest priority ready thread by iterating through the TCB array and checking for the first thread that is not blocked by either a semaphore or a timeout. In particular, the NIL pointer is used as the value of the semaphore pointer that represents no blocking. The iteration starts with the highest priority thread, and continues with decreasing priority until the lowest priority thread is reached. For each step, the semaphore pointer of the TCB structure is tested against the NIL pointer. In case all the threads specified by the user (including main thread) are blocked, the idle thread is selected to execute. The idle thread is a special thread with the lowest priority (lower than the lowest priority user thread) that is never blocked, and works as a fail-safe in case of special situations. The current implementation of the idle thread has no functionality, although debugging and security features can be implemented in the future.

It is also important to highlight that for the kernel there are only two possible states for a thread, either ready or blocked. There is no practical distinction between a thread blocked by a semaphore and one blocked by a delay. In other implementations [ref], it is usual to analyse differently threads which are waiting in a “suspended” or “sleep” state. The distinction is important when different data structures handle threads depending on their status, which is commonly chosen for efficiency purposes. Nevertheless, as the scheduler implementation is designed to handle a small number of threads and reduced memory footprint, it is only important if a thread is ready for execution or not.

An example of the operation is depicted in Figure 5. In that case, there are three user threads (which one of them is the main thread). The highest priority thread is blocked by a semaphore. The lowest priority thread is blocked waiting for a delay to timeout. In that case, the iteration will start with the highest priority thread, continue because it is blocked by a semaphore, and stop in the next thread (middle priority in this case), as it has a NIL pointer. In case the second thread was also blocked (all user threads blocked), the iteration will stop at the idle thread, which is always in a ready state.

Data Structures

The main data structure of the scheduler is an array of priority ordered TCBs. In turn, the TCB data structure provides the basic functionality and is prepared for future enhancements of the scheduler that require extra fields. In particular, the *dynamic_priority* field is intended for future use when the priority of threads can be modified dynamically by some resource access protocols like priority inheritance of priority ceiling. The *stack_top* field is intended for stack overflow checking and protection. Together with the sentinel value written at the top of each thread stack (cross reference with lower section), it is possible in a future extension to detect and take some particular action in case of stack overflow.

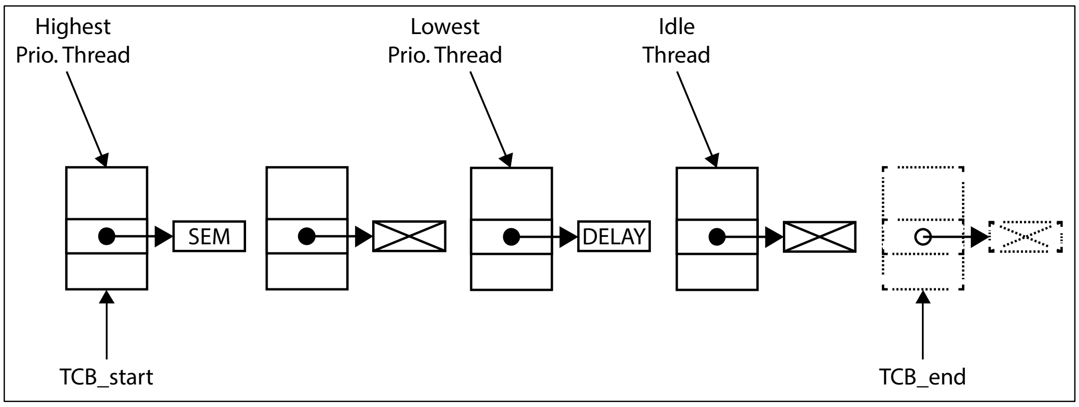

Figure 5. TCB array example for 3 user threads + idle thread

```
/* 4 words */
typedef struct {
    volatile uint32_t * volatile current_top; // Top of current stack
    uint32_t *stack_top;                      // Low memory address
    volatile semaphore_t * volatile semaphore; // Semaphore stopped at
    uint16_t  stack_size;                     // Stack Size in 64 bit words
    uint8_t   base_priority;                  // Static base priority
    volatile uint8_t   dynamic_priority;      // Dynamically assigned priority
} thread_t;
```

Figure 6. TCB data structure

(notice the use of volatile qualifier to prevent caching)

Context Switch State Space

Since the context switch is effectively executed at the lowest priority available, it is possible that a higher priority interrupt (either SysTick system exception or any user defined interrupt) requests scheduling services and consequently modifies the target thread that is supposed to be switched in. In particular, switching a task with itself is a forbidden operation. The reason is that the current state of the thread is discarded and switched to a previous state that is corrupted and no longer valid.

The ARM-v6M architecture only provides information about the pending status of the PendSV exception, but is unable to accurately report if the PendSV exception is inactive, active or preempted. Nevertheless, with an extra software flag to know if the critical section of the context switch has been executed or not, the necessity to know the status of the PendSV exception is obviated. Furthermore, it is possible to have full control of the context switch outcome at any moment regardless of the stage at what the context switch process is interrupted.

The “switch pending” flag is enabled from the very moment the context switch is requested and the PendSV exception is set to pending. It will only be disable when either: the context switch is cancelled before the PendSV activates for the first time (the status is still pending), or the context switch critical section has been executed. That determinism provides consistency to the state space and allows to

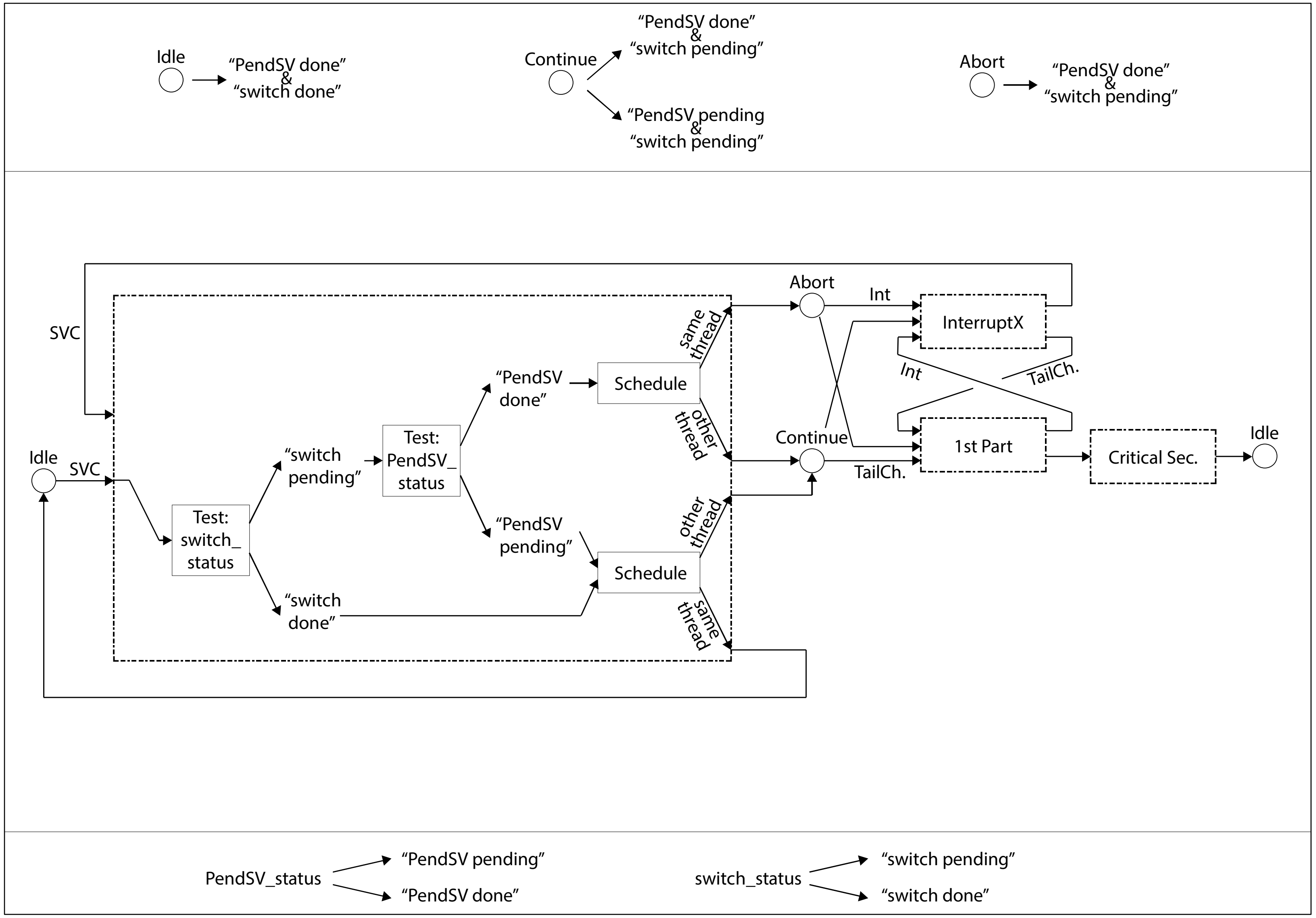

Figure 7. Context Switch State Space

decide what actions can be performed if the newly required switch in task is different form the one intended at the initial context switch request.

As long as the context switch critical section has been passed, the PendSV can be set again independently if the exception has already completed, or it is still in progress of being completed. That is possible because the PendSV exception can have actually four possible states: inactive and not pending, inactive and pending, active and not pending, and active and pending. For the analysis purposes, being pending, regardless of being active or inactive, is the same effect for the program execution and the state space analysis.

In case the PendSV has already started executing and an interrupt that preempted it during the first part requests that the thread that is being switched out is switched in again, a state known as "abort" is reached. In that state, it is necessary to cancel the context switch, but it is not possible as the PendSV exception is already running and some registers have already been pushed to the stack. In that particular case, an extra flag, the "abort_flag" is used to indicate during the critical section that the SP should not be changed. Consequently, when the critical section has passed, the rest of the PendSV simply pops the registers that pushed previously, leading to a system state identical to the one when the exception started, effectively cancelling the context switch.

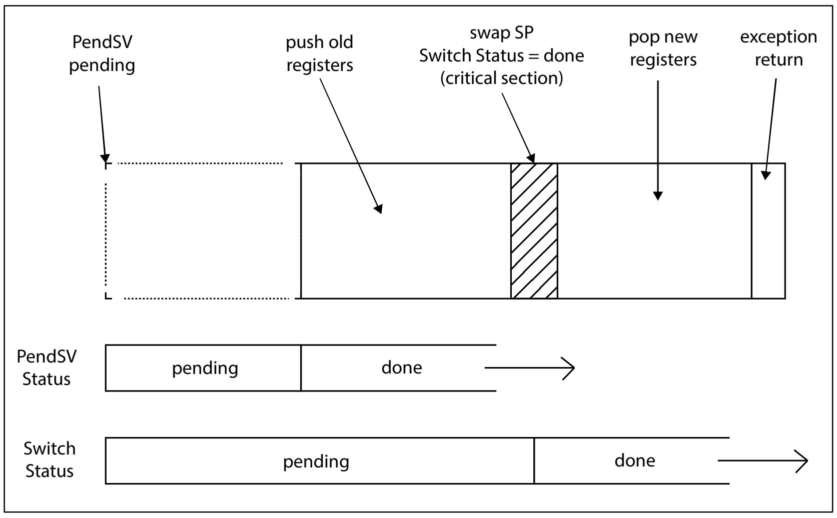

Figure 8. PendSV normal progress, with status of flags

5.3 Semaphores

The semaphore block enables the functionality of resource sharing, mutual exclusion and synchronization. Although it is not a base component, it is one of the two mechanisms made available by the kernel to block and switch thread execution (the other being the time management block). The semaphores are declared statically, with the initial value being the only creation parameter. There is no distinction between counting semaphores, binary semaphores and mutexes.

For the purpose of the present analysis, a mutex is distinguished from a binary semaphore with two characteristics: it handles the concept of ownership and it is initialized as "not taken". The ownership represent the awareness of the system of what thread currently has the mutex "taken". The ownership, together with the property of being initialized as "not taken", leads to the possibility of resource access protocols like priority inheritance or priority ceiling.

However, the use of resource access protocols leads to dynamic priority assignment. With the current scheduler implementation, the change of priority was not possible, as they are statically designed. Nevertheless, the data structure for the semaphore was prepared with the “ceil_priority” parameter for the future implementation of the priority ceiling protocol. The priority ceiling protocol was chosen because it prevents deadlocks, in contrast with the priority inheritance counterpart.

Data Structures

The data structure is designed to be compact and enable future extensions at the same time. As previously described, the “ceil_priority” is designed to future implementation of the priority ceiling protocol. If the concept of ownership is required in the future, a pointer to the TCB of the owner thread can be appended at the end, changing the size from one to two words. As long as the number of bytes in the structure is kept a power of two, the generated code for handling the data is relatively efficient.

```c
/* 1 word */
typedef struct {
    volatile uint16_t value;
    const    uint16_t ceil_priority;
} semaphore_t;
```

Snippet 4. Semaphore Data Structure

Blocking Pre-Check

In order to decrease the execution time of the wait operation, a test is made before the actual atomic operation is executed. In case the wait operation will most probably block, the atomic operation is realized through the SVCall handler. The underlying reason is that a switch to handler mode will most probably be necessary, as a context switch will eventually be required. As the tail-chaining mechanism allows fast entry from the SVCall handler to the lower priority PendSV handler, the execution time is only slightly increased in comparison with only executing the PendSV handler. However, the main advantage is that the PendSV will always be called from tail-chaining a higher priority interrupt, what simplifies the analysis of the context switch state space.

Nevertheless, there is a slight chance that the semaphore does not actually block the current thread, even if the previous test suggested it, because an interleaving event made the semaphore available between the non-atomic test and the actual execution of the atomic SVCall taking operation. In that case, the SVCall handler simply returns to the calling thread, without requesting any context switch. The described behaviour is depicted in Figure 8

In the case that the pre-check test outputs that the semaphore is available, the system call creates an atomic operation in handler mode by disabling the interrupts. Immediately after, the semaphore wait operation is executed. In case it was successful, the interrupts are enabled again and the function call simply returns. However, there is the slight chance that the semaphore was taken between the pre-check and the start of the atomic section. In that case, the interrupts are enabled again and the SVCall handler is requested in the same way as if the test output the semaphore would probably block the current thread.

The situation just depicted leads to the worst case execution time of the wait operation: the test outputs that the semaphore is available, then it is unsuccessfully acquired it in thread mode and the

SVCall handler is required to block the current thread. Nevertheless, the pre-test code was carefully crafted to only increase the worst case execution time slightly compared to the whole wait operation.

In contrast, when the test accurately predicts that the wait operation will not block, the execution time is reduced considerably, as there are no exception entry, tail-chaining or exception-return latencies added to the semaphore wait operation. In summary, the pre-check system is implemented to reduce dramatically the execution time in the case the semaphore is available, while slightly increasing latency of the operation when a context switch is required.

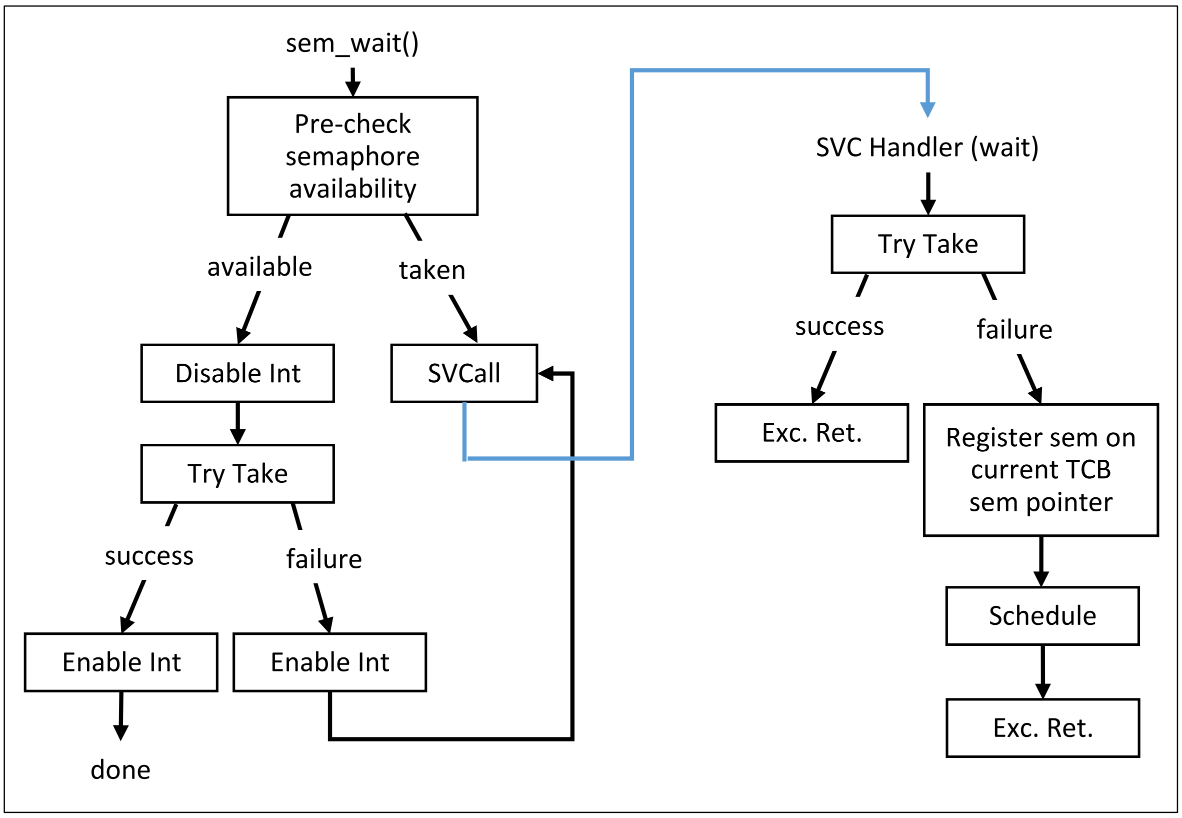

Figure 9. Semaphore Wait Operation

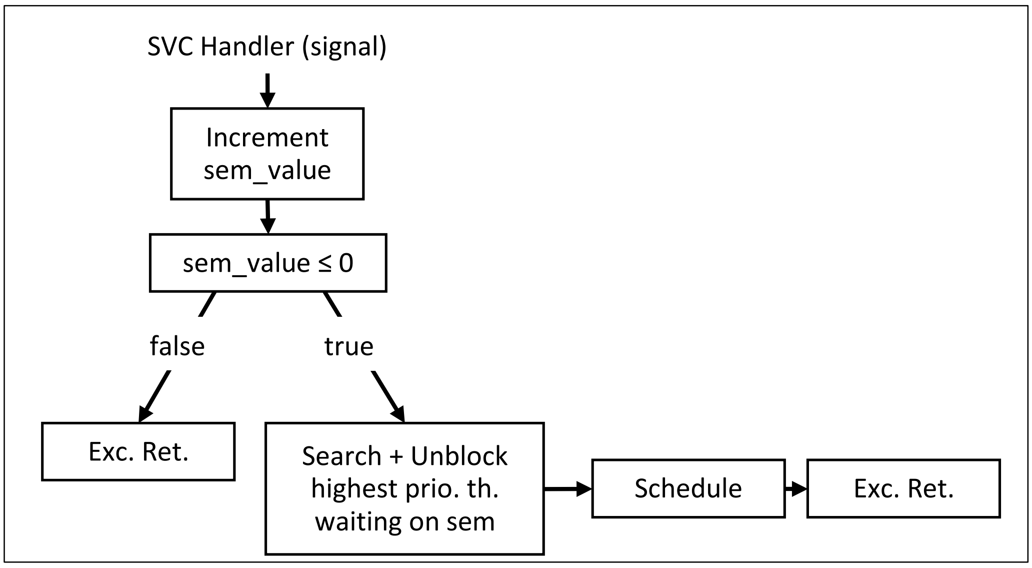

Figure 10. Semaphore Signal Operation

5.4 Time Management

The time management module allows to pause the execution of threads for the requested amount of time. As in other RTOSs, the module can be enabled or disabled at compile time, depending on the needs of the user (in contrast with the semaphore system which is always enabled). The choice of the time granularity for the system is highly relevant, as it is a tradeoff between functionality and kernel performance: high granularity allows high precision but generates temporal interference to executing threads as they are constantly preempted by a kernel exception that effectively consumes processor time.

In general, there are two possible implementations for a time management module. The most common is to interrupt the normal flow of execution at regular intervals. The system code executed at periodic intervals handles events that depend on the pass of time, like updating the system time, or accounting for processor usage in systems which schedule based on time slices or round-robin strategies. The described interrupt is commonly known system tick, and is standard feature in both, OSs and RTOSs.

The other approach relies on the fact that usually it is known beforehand when the next system event will occur. In that case, it is possible to program a hardware timer to interrupt the system when the next system event will be executed. The previous approach is regarded as “tickless” operation, as it lacks of a periodic interrupt that measures the pass of time. In fact, although the system tick code of regular OS executes periodically, only a few of those interrupts execute useful code. In general, the system tick simply modifies counters, and only performs relevant actions when those timers reach a particular value.

The “tickless” implementation is particularly useful in energy constrained systems, as it allows to enable the deep power-saving modes of modern processors during long periods. In contrast, systems based on periodical interrupts need to reactivate the system on regular intervals, what is generally inefficient, as “waking up” the system is particularly energy consuming. Another advantage is the absence of periodic system code interrupting the user code flow of execution (particularly important for processors with cache memories). Consequently, the tradeoff between temporal precision and interference to thread execution is removed, and the granularity can be set as high as the hardware timer is able to provide.

For the present implementation, the system tick approach was chosen because of two reasons. First, the STM32F0 Cortex-M0 implements the SysTick module, which is a tightly integrated timer specifically designed to provide system tick functionality. The two main benefits of the SysTick module are the implementation independency (compared to peripheral timers provided by each manufacturer) and the specialized interface that reduces configuration code. Furthermore, even when the SysTick module is an optional feature in the ARMv6-M architecture, it is a standard component for the ARMv7-M architecture. As the future plans of the Ell-i Co-Operative target also the Cortex-M4 processor (which implements the ARMv7-M architecture), the design choice provides scalability and manufacturer independence.

The other reason for selecting the system tick approach is that the target system is not energy constrained. In fact, the target platform requires a minimal power consumption in order to comply with the PoE node standard behavior [73]. Furthermore, from the energy perspective, the “tickless” operation is only beneficial when the system is in the idle state. If the system is intended to rarely reach the idle state, as is the case of the projected target system, the periods that allow power-saving modes are scarce, and the difference in energy consumption between a tick and a “tickless” system is negligible.

As the kernel is designed with the system tick interrupt approach, the selection of the interrupt period is of high importance. The selected tick frequency is 1000 Hz, which provides a time granularity of 1ms. The decision is backed by the fact that the Arduino environment has a delay function with milliseconds as argument and the average interference to the user threads with the current implementation is below 0.125%[3]. Although a higher granularity decreases the error between the requested and real elapsed time, it is not considered necessary, as the projected application has no stringent timing requirements, and the jitter generated is acceptable.

As a result of the selection of 1ms interrupt period, the delay difference between the requested value and the real delay is in the range of [-1ms, 0ms] non-inclusive. The maximum difference occurs when the delay is registered just before the next system tick interrupt. In contrast, the minimum difference is present when the delay is registered just after the previous system tick. In summary, the delay will always be less than the expected value, with the limit of slightly less than 1ms. The described behaviour is depicted in Figure 10.

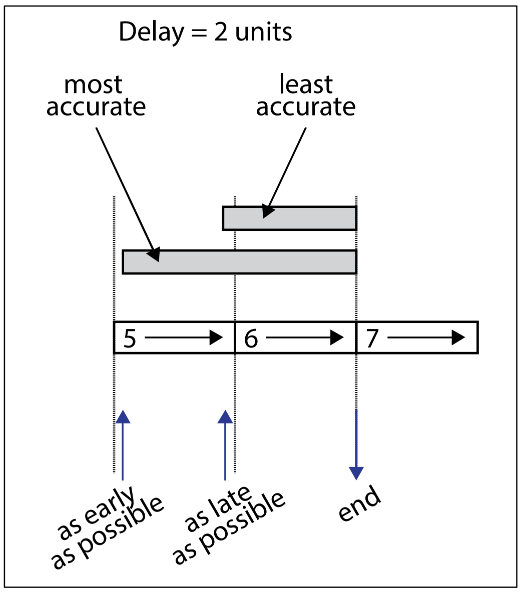

Figure 11. Worst and Best Case Timing Errors

Data Structures

In the case of using a periodic system interrupt, the minimization of the execution time is of vital importance, as it directly affects the time performance of any kernel implementation. In the particular case of the implemented scheduler, the function of the SysTick exception is limited to handle the

[3] Considering an exception entry of 15 cycles, an exception return of approximately 15 cycles (real value 16 cycles), and an interrupt execution of 30 cycles (the current implementation is faster) leads to 60 effective cycles lost per interrupt. With the target operating frequency set to 48.000.000 cycles per second, using 60.000 cycles for the SysTick exception (1000 interrupts per seconds) leads to 0.125% of processor time used for handling the system tick interrupt.

timeout of blocked threads. For an efficient implementation, the time complexity should be preferably constant, and with a constant as small as possible. As a result, the queue is the data structure is selected because of its constant access time to the initial node.

If the timeout events are ordered by incremental delay and only the difference from the previous event is stored, it is only necessary to decrease the time counter of the head of the queue to represent the pass of time. Because accessing the head of the queue and decreasing a variable within it do not depend on the number of events, the time complexity remains constant. Furthermore, al the selected open source RTOSs revised initially utilize the same “queue of incremental delays” concept.

From the possible implementations, the single linked list alternative is preferred as it offers constant time complexity for the head insertion (leading to constant time for the access of the head element) as well as linear time for the insertion of an ordered element. Although the doubly linked list was also considered, the property of having constant insertion time at the tail of the queue brings no effective improvement for the application. As the queue is not required to be traversed backwards, neither to have elements added at the end directly, the increased memory (RAM) footprint and execution overhead of a doubly linked list are not desirable.

Nevertheless, the possibility of constant complexity for the ordered insertion could improve the execution time and scalability of the new timeout registration operation. The use of search enabled structures like binary search trees or self-balancing binary search trees in conjunction with the queue was contemplated, as they provide constant insertion time for ordered elements. However, the memory and execution time overhead related to a search tree implementation is not an acceptable tradeoff, as the number of threads is relatively small and the linear time option outperforms the scalable alternative.

```c
/* 4 words */
typedef struct timeout_struct timeout_t;
struct timeout_struct {
    volatile uint32_t count;
    volatile semaphore_t *volatile *sem;
    uint32_t flags;
    timeout_t *next;
};
```

Snippet 5. Timer Data Structure

Interaction with Scheduler

The interaction with the scheduler uses the same scheme of the semaphore module. In order to block a thread during the desired delay time, the TCB blocking pointer is set to a predefined value different from the NIL pointer. As described in the previous sections, for the scheduler there is no difference between a thread blocked by a semaphore and one blocked by a timeout when searching for the highest priority ready thread. The value set in the blocking pointer is fixed, and designed to never collide with a semaphore or any other relevant data. As a future extension, an alternative is to store the pointer to the timeout event in the queue, which could allow to “wake up” threads before the completion of the desired delay (functionality not present in the current implementation).

When the head of the queue reaches the end of its timeout delay, the blocking pointer of the blocked thread TCB is set to NIL, what changes the state to ready and enables possible scheduling. In case more than one timeout event reaches the end of their delay at the same time (the time difference of the nodes with the current head is zero), all the threads blocked by those timeout events are unblocked at once. Nevertheless, the operation of the scheduler is not affected and the ready thread with the highest priority is selected to continue. In other words, there is no special ordering for threads that are waken up simultaneously than the predefined priority.

5.5 Memory Management

Although the designed scheduler does not have a memory management system in the traditional sense, the allocation of static memory and relevant data structures for the correct operation follows a methodically defined process. The design is based on the peripheral initialisation implemented for the Ell-I Runtime [78] (for details, refer to section Peripheral Initialisation). The overall strategy is focused on compile-time calculation of required space, with particular focus on RAM availability and compactness of Flash stored data structures.

Linker Interaction

Following the idea of having control of the linking process from the original Ell-i Runtime, a tailored linker script is used to provide a very particular ordering of objects among the RAM and Flash memory. In concrete, the "section" attribute extension for the C language by GNU GCC is used to pass arguments to the linker, which optimizes the process of collecting objects from multiple source files. Additionally, the ability of using variables provided by the linker in the C files is extensively used to offload the calculation of addresses and space requirements to the compile time.

In particular, two special kind of objects were specified for this scheduler: placeholders and thread initialisers. The placeholders are objects used for the calculation of RAM availability. They are finally discarded by the linker, as they do not contain useful data and are only used for setting the address limits of their particular sections. The initialisers are data structures on the program memory (Flash) that completely describe the abstraction of a thread for the scheduler. They follow the same idea of the peripheral initialization records already implemented in the provided Ell-i Runtime.

There are three kinds of placeholders for the scheduler: Stack, TCB and Timeout. Each time a thread is registered to the scheduler (at compile time), a placeholder of each kind is created. The Stack placeholders have the additional requirement of being 8 bytes aligned, which is a consequence of the architectural requirement for the stack pointer to always be 8 bytes aligned [36, 68]. At link time, the objects are grouped in their respective categories to enable the calculation of the start and end addresses of each section. After that step is completed, the objects are discarded, as the only important information is the limits of each section. Those variables will be propagated by the linker to their respective C variables.

Thread Initialisation Records

The Thread Initialisation Records contain all the information necessary for initializing the TCB, as well as parameters regarding the stack space and the data necessary for initializing the stack space for the correct initialisation during the first context switch to that particular thread. The corresponding data structure is presented in Snippet 6. Apart from the "stack_size" and "priority" fields, the rest of the parameters are used to provide the initial context to the stack space of each thread.

Each initialisation record is created as part of the thread registration at compile time (the other objects created are the three placeholders: Stack, TCB and Timeout). A very important step is to name the section the objects are assign based on the provided priority, as the linker orders the collected initialisation records objects based on this property. This characteristic provides two advantages. First, it offloads the processing load from ordering the TCBs and stacks by priority at runtime. Second, it allows the registration of non-consecutive priorities without any consequence on the linking process or runtime execution.

```c
/* 8 words */
typedef struct {
    const uint16_t stack_size;     // In 64 bit words
    const uint16_t priority;       // Base priority
    const void (*function)( uint32_t arg0,
                            uint32_t arg1,
                            uint32_t arg2,
                            uint32_t arg3);
    const void (*exit_function)();
    const uint32_t arg0;
    const uint32_t arg1;
    const uint32_t arg2;
    const uint32_t arg3;
    const uint32_t flags;
} thread_init_t;
```

Snippet 6. Thread Initialisation Data Structure

Decompression Process

The decompression process consist on iterating through the initialisation registers in priority order. As the priority order is ensured by the linker, this process is simplified to a consecutive-order iteration implemented by a C “for” statement. The first step when accessing an initialisation record is to test if it is the record of the main thread, as the initialization process is slightly different.

In case of all threads except main, the Stack space for the corresponding thread is written with the initial context on top of the stack (based on the function parameters and default values), and a “sentinel” value at the bottom of the stack to track possible overflows. Optionally, the rest of the stack space can be cleared if required. Finally, the TCB is initialised with the parameters regarding the stack limits, as well as the priority provided by the linker. A very important step during the TCB creation is to initialise the semaphore pointer to NIL. In other words, no thread initially blocked by any semaphore or timeout.

The main thread follows a different initialisation procedure. First of all, it does not require the stack space to be initialised or to write an initial context on top of it stack, as it is already running. Second, the stack limits do not need to be assigned at runtime, as they are already assigned and are valid when this decompression takes place. In fact, the TCB state of the main thread is initialised as a thread that is currently running: the parameters are not valid until a context switch stores the correct values for future re-entrancy.

At the end of the initialisation process, the RAM memory is allocated with the map shown in Figure 11. The two most important sections for the scheduler are the pre-allocated, “thread_section” and “timeout_section”. They are pre-allocated in the sense that they do not contain any valid data until the scheduler is initialised. It is important to note that the stack of the main thread is already in use when the scheduler is initialised. As a consequence, all the other stacks are allocated after it. The interrupt stack is located at the very end just before a possibly empty area. The objective is to prevent any problem in case the interrupt stack overflow, as it is not possible to precisely predict the number of nested interrupts or the functions user defined ISR may call during execution.

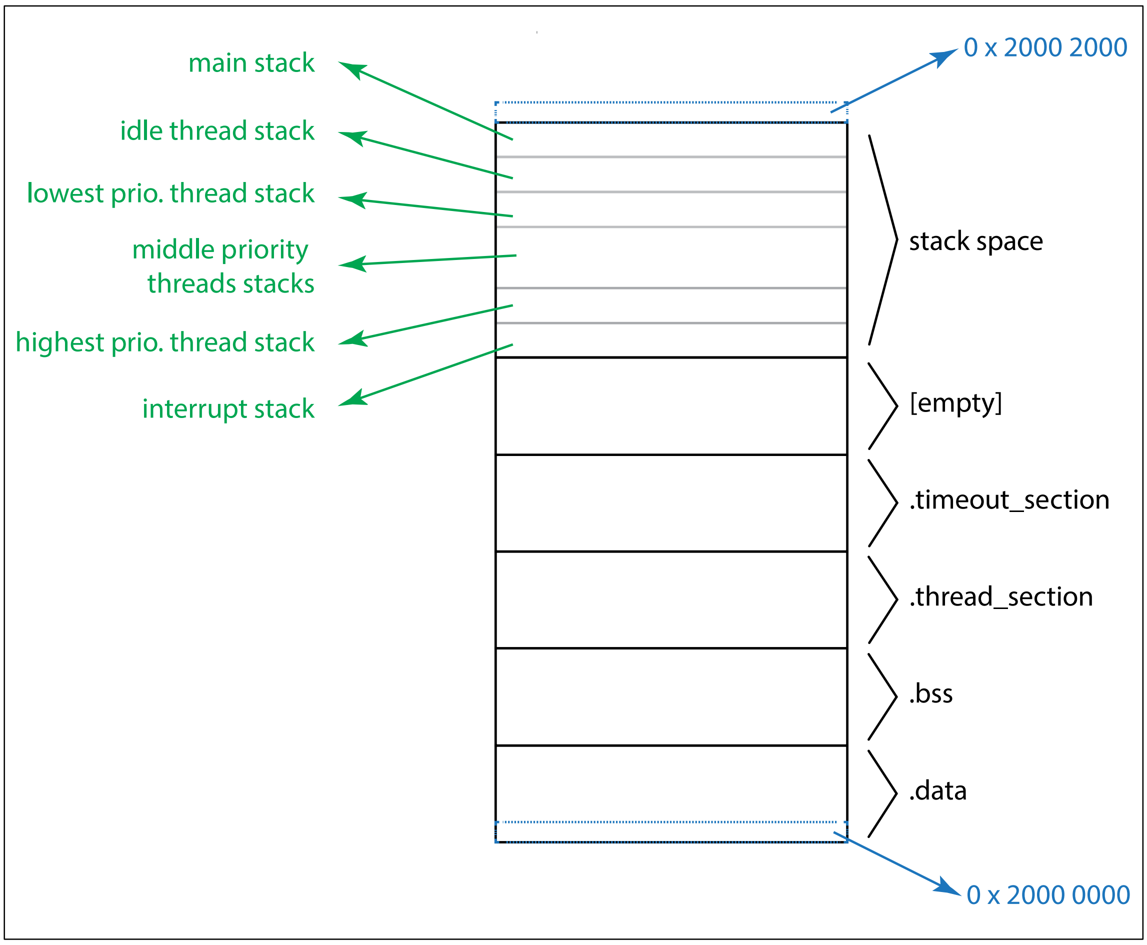

Figure 12. RAM Map

Chapter 6 – Evaluation

This chapter describes and discusses the methods used for testing and evaluating the scheduler and other Open Source RTOSs, as well as the results obtained from this process. The first section presents the selection procedure for comparable RTOSs. The second section describes the testing framework, which comprises a hardware setup as well as a common codebase for software support. The third section details the workload used evaluate the memory and timing performance of the selected RTOSs and the Ell-i scheduler. The next section presents the resulting data, which is evaluated and discussed in the last section of this chapter.

6.1 Setting

In order to evaluate the overall performance of the developed system, the chosen method of is to compare certain features with similar pieces of software also distributed as Open Source. As enabling threading in a very memory constrained system was the objective of the designed scheduler, systems with similar set of features were selected. In particular, the selected systems were also targeted toward small to medium MCUs, and had the option of disabling or enabling features during compile time. This particularity allowed a more fair comparison, as all the elements that provide functionality beyond the scope of the current analysis were deactivated.

The selection of similar systems for comparison was achieved in a two stage process. The first step identified RTOS that support the MCU used, which has relatively high memory, speed, and instruction set constraints. As the MCU used is on the low end of the ARM Cortex-M family, only a subset of the existing (commercial and open source) RTOSs have ports to that particular architecture. In contrast, if it was a high end MCU, like the Cortex-M4, both RTOS and normal OS (such as Linux) have been ported and are available for off-the-shelf usage.

The second step consisted on filtering the existing software according to the following criteria:

1. License
2. Popularity and Active Development
3. Memory Footprint (Flash and RAM)
4. Similar Features (Scheduler, Semaphores and Timer)
5. Documentation and Adaptability to the Testing Framework

From the licensing side, every software distributed with only a commercial licensing was discarded. After this, a quick analysis of popularity and how recently the software have been actively developed was performed. The result of this analysis left only four alternatives for further study (presented in Table 8). However, a special clause form the licensing terms on FreeRTOS prevents using the software for comparison purposes. Consequently, it was removed from the set for comparison, leaving only three systems (ChibiOS/RT, CoOS, and eCos).

ChibiOS/RT	***FreeRTOS***	***CoOS***	***eCos***
Multiple [87]: • Pure GPL3	Multiple [88]: • GPL2 with linking exception *	Single [89, 90]: • 3 clause BSD	Single [91]:

• GPL3 with linking exception • Commercial *Demos, test code and low level drivers under Apache 2.0	• Commercial *Another exception is that the software cannot be used for any kind of comparison.		• GPL3 with linking exception

Table 8. License Analysis for selected RTOSs

After the previous step, the remaining RTOS were studied at the source code level to evaluate the compatibility with the designed testing framework. In general, the modularity and limited usage of inline assembly allowed a straightforward understanding of the internal operation in all but one case: eCos. The main problem was that eCos uses an additional process before generating the actual source code and providing it to the compiler. This additional process is intended to generate the C code from general templates, after a configuration procedure. However, the base templates are not easy to understand, as they are not C or C++ code, but the input for a configuration tool. As a consequence, the absence of C source files and the necessity of using a specific set of tools (compiler and linker), prevented the adaptation of eCos to the testing framework.

At the end, only two systems were left for comparison: ChibiOS/RT and CoOS. It is important to highlight that FreeRTOS had all the properties for being adapted to the testing framework. Nevertheless, the restrictions for comparison in public documents prevented its usage in this evaluation. The selected systems were ported to the testing framework for posterior analysis. The testing framework and evaluation procedures will be discussed in the next section.

6.2 Procedure and Metrics

Software Testing Framework

A common set of compiler, compiler parameters, and helper files was stablished in order to remove the possible differences in space and execution time that different compilation process would lead to. From one side, the common compiler and respective configuration flags enabled measuring the efficiency of the scheduler implementations. Not using a common building process would lead to interference due to the efficiency and optimization capabilities of a specific compiler. In particular, as the objective is to deliver scheduling functionality to a memory constrained platform, the compiler was configured to optimize for space, and not for speed.

Following the Open Source trend, the compiler tools used were the port from GCC for the ARM architectures [92](supported by ARM Holdings). It is important to highlight that the building process was purposely divided in a separate compilation and linking steps, in order to use a custom made linker script that allows further control of the specific location of objects. The details of the compiler and flags are presented below.

- **Compiler**: GNU Tools for ARM Embedded Processors 4.8- Q1 2014 (Bare metal EABI pre-built binaries for running on a Linux host)
- **Relevant Compiler Flags**: -Os --param max-inline-insns-single=500 -ffunction-sections -fdata-sections -fno-common -nostdlib -mcpu=cortex-m0 –mthumb
- **Relevant Linker Flags**: --gc-sections

As the target is memory constrained, the compiler is indicated to generate separate sections for each object, and subsequently the linker discards (garbage collects) all unused sections. With this configuration only used elements will be kept. As a result, all the files that compose the ported RTOS can be included in the compilation process, and only the elements that provide the required functionality will be kept. This strategy simplifies considerably the adaptation process of any RTOS to the framework.

From the other side, a common set of initialization and library functions were used for all the ported RTOSs. From the initialization files, the most basic configuration needs two elements: a vector table, and a reset vector handler. In turn, the reset vector handler realizes at least three things: initializes variables by copying data from Flash to RAM, configures the clock and basic power systems, and calls the main function. From the library functions, the ported RTOS only require the division subroutines. As GCC determines which library to compile against based on the architecture, all systems used the same library.

However, as some RTOS require different subroutines, the size of the objects taken from the libraries might vary. In addition, the user application might also vary from system to system, as data structures that support threads and synchronization primitives are different in each system. The size of the interrupt handlers for specific system services is variable as well. As a result, even if the basic initialization system is common for all ported RTOSs, detailed size analysis is needed. A size analysis script in Python was written to accurately provide the sizes of objects and particular linker sections (e.g. the size of the space between two identifiers), and effectively isolate the different parts of the system: mainly the user application and the scheduler functionality.

Hardware Testing Framework

A framework for measuring the execution time of selected primitives of the studied RTOSs was also implemented. In essence, the system consisted of a logic analyser connected to a low latency output pin. This configuration allowed measurements close to individual cycle accuracy. The main reason for this construction is the absence of internal channels able to collect information about the core operation with the required precision. In particular, some hardware factors that directly affect the metrics are in the range of ten to twenty cycles.

Although other possible solutions include the use an available timer peripherals, various factors prevented their use. First of all, the precision required (in the range of tens of nanoseconds, or single cycle) urged for knowledge of the precise latencies of hardware events, such as timer latencies or exception entry times. As this information is not publicly available, the selected framework allowed the achievement of a second purpose: the accurate measurements of hardware events. These events include exception entry, exception return, tail-chaining, and timer latencies with cycle accuracy.

In order to calculate approximately the accumulated errors when measuring events at the core from the outside, it is necessary analyse in sequential order the locations at which latencies are added to the measurement. This analysis is detailed in the section below.

- First, the event is generated at the core.
- The signal is sent toward the GPIO peripheral through the System Bus. However, it is not directly received by the GPIO, first is arbitrated by the Bus Matrix.

- The Bus Matrix uses a round-robin algorithm to arbitrate between the two masters: the core and the DMA driver [71]. Unfortunately, there is no accurate information about this latency. Nevertheless, the DMA was deactivated during all the tests performed.
- Then, the Bus Matrix directs the signal to the GPIO through the AHB2 Bus.
- The GPIO receives the signal and starts the activation or deactivation of the desired pin. Although there is absence of details about the individual latencies between the core and the GPIO peripheral, it is stated that the maximum frequency for toggling an output pin is 2 cycles. As a result, the latency for a signal is between 0 cycles and 1 cycle.
- After the GPIO receives the signal to change the output state of a pin, the latency for the output to reach the desired level is mainly dependent on the voltage input and the capacitance of the load connected to the pin (assuming the current demand is within the accepted limits) [93]. As the pins were directly connected to the logic analyser probe, and it is designed have an equivalent capacitance of 7 pF [94], the fastest response time is expected form the GPIO pin: 5 ns.
- The Logic Analyser is able to read the signal from the output pin. The maximum time resolution that the analyser can handle is 10 ns when operation at its maximum frequency (set to measure at 100 MHz). However, another factor affects the operation, as the voltage levels that the Logic Analyser identifies as logic values might differ from the ones determined by the MCU. As a consequence, the manufacturers of the Logic Analyser indicates that the practical maximum bandwidth is 25 MHz (when operating at 100 MHz) [94]. In conclusion, the error that the analyser generates in practice is 20 ns.
- Finally, the signal is received and stored by a Personal Computer. As the software that drives the Logic Analyser and stores the data warns if there is any error in the data transfer, no latency or error is provided for the connection between the Analyser and the PC.

In summary, there are three latencies that accumulate for the final error calculation. As the processor is set to operate at its maximum frequency for the evaluation, the delay for the path between the core and the GPIO is equivalent to approximately 21 ns. Taking into consideration the other two latencies, 5 ns for the output pin state, and 20 ns for the Logic Analyser, the total error that for the final measurement is of, approximately, 46 ns. The details of the multiple hardware layers that a signal travels from the Core to the PC is illustrated in Figure 12.

As the time each iteration of the experiment takes is variable, it is necessary to calculate the average of multiple cycles to obtain a more accurate and valid figure of the behaviour of the studied RTOS. This variability is caused by two main factors: the measuring error, and the possible variations due to the implementation of the RTOS. As the Logic Analyser simply outputs a series of transitions in the voltage measured by the probe, an additional step was necessary to extract useful data. A script was written in Python, which allowed to obtain statistical values from the raw data generated by the Logic Analyser, mainly the average, maximum, and minimum values.

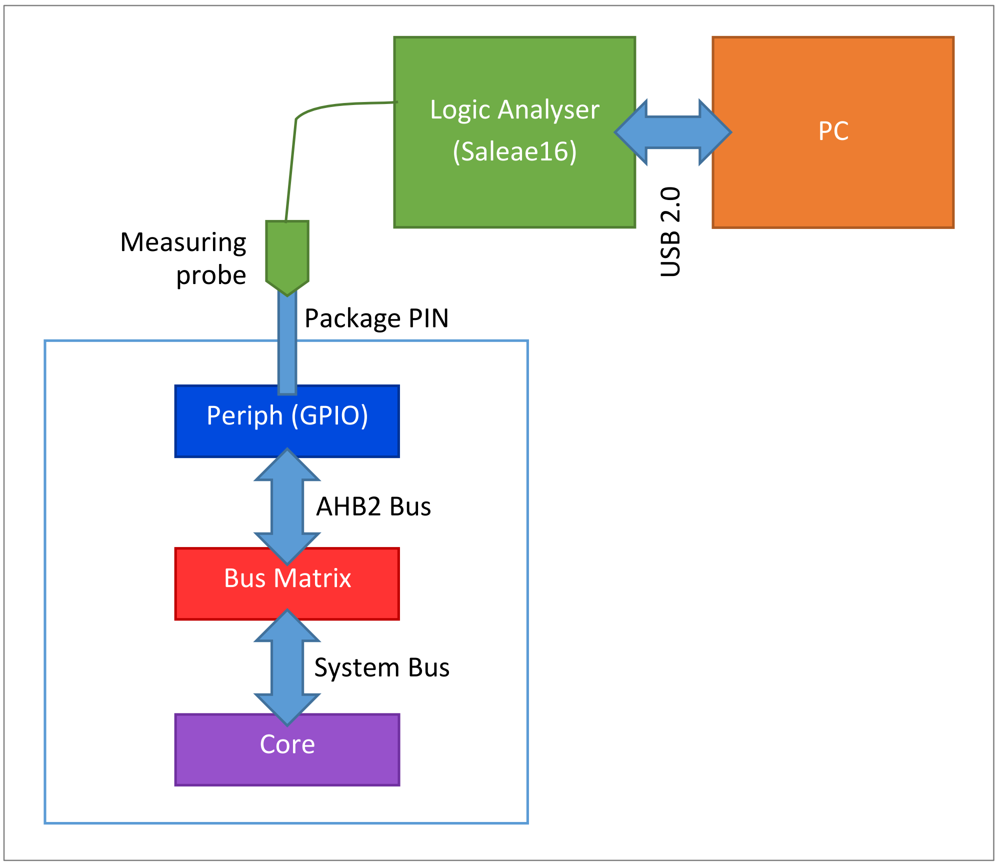

Figure 13. Hardware Testing Framework

6.3 Workload

The objective of the workload is to enable measuring the context switch latency and the related time costs of synchronization primitives. It is important to highlight that in all the studied systems, including the scheduler designed for this work, it is not possible to only measure the context switch latency. In concrete, the context switch never happens by itself; it is always associated with an event that requires a thread different from the current one to execute. As a consequence, in the present work and in related literature [84, 86], the measured time is the compound of a synchronization event together with the subsequent context switch that it generates.

For this evaluation, two events were measured in the studied systems. The first event consists of a semaphore signal (V operation) from a low priority thread that activates a high priority thread and, subsequently, generates a context switch. The second event is a semaphore wait (P operation) from a high priority thread that blocks, and allows a low priority thread to activate, generating a context switch. As both events require only two treads with different priorities, the evaluation is performed with only two user threads. As system created threads, such as the idle thread or some monitoring routine, are not under the control of the user, they are not considered within this analysis.

It is also important to highlight what kind of semaphores were selected for this test. As the scheduler designed for this thesis has no resource access protocol bind to the semaphores, similar primitives with the minimal functionality were also selected in the studied RTOSs. For the case of ChibiOS/RT, the binary semaphores offer the minimal functionality, as they can only have two states, are not associated with any resource access protocol (Priority Inheritance in this case), and can be signalled from ISRs [95]. However, ChibiOS/RT offers the possibility of ordering the binary semaphores queue based not only on priority, but on a FIFO scheme. For the sake of comparison, the binary semaphores were configured at compile time to operate based only on priority, providing the equivalent functionality to the designed system.

For the case of CoOS, semaphores were selected among other synchronization primitives like mutexes or flags [96]. The selection is based on the fact that semaphores are not bind to any resource access protocol (in comparison with mutexes, which use Priority Inheritance), the minimal implementation, and the ability to be signalled from ISRs. As in CoOS there is no distinction between binary and counting semaphores, it is not necessary to choose among them and simply “semaphores” are used.

The system load is designed to make it react by itself. In other words, there are no inputs that trigger any change during measurements, only outputs. As it was stated above, it is not possible to isolate and measure only the context switch time. Therefore, for reducing the interference of the scheduling system and semaphore management, only two threads were created.

This minimal configuration is the same strategy used in the work of Otava [84] and Ugurel and Bazlamacci [86]. In brief, a high priority thread blocks on a synchronizatin primitive owned by a low priority thread. After the context switch, the low priority thread immediately releases the primitive, generating a second context switch that returns to the high priority thread. This sequence is repeated indefinitively to enable multiple samples for statistical analysis. The general pseudocode for the operation is described in Snippet 7.

The particular approach for enabling external measuring is to generate a GPIO pin toggle exactly before the start of the synchronization operation and another when the new thread starts executing. Another GPIO pin is also toggled for measuring the next context switch, just before the primitive is released and again when the high priority thread continues execution. As the time granularity of the hardware testing framework allows individual measurements (in comparison with number of switches in a predetermined time lapse), an additional busy loop is added to the main thread for providing a hint that one cycle has executed. Additionally, this busy loop allows to measure the interference from the GPIO toggling operation when comparing the output graph of both GPIO pins. A detailed scheme of the output waveforms and matching events is presented in Figure 13.

High Priority Thread

```
High() {
    busy delay loop
    set_pin0
    sem_wait()
    reset_pin1
}
```

Low Priority Thread

```
Low() {
    reset_pin0
    set_pin1
    sem_signal()
}
```

Snippet 7. High and Low Priority Threads

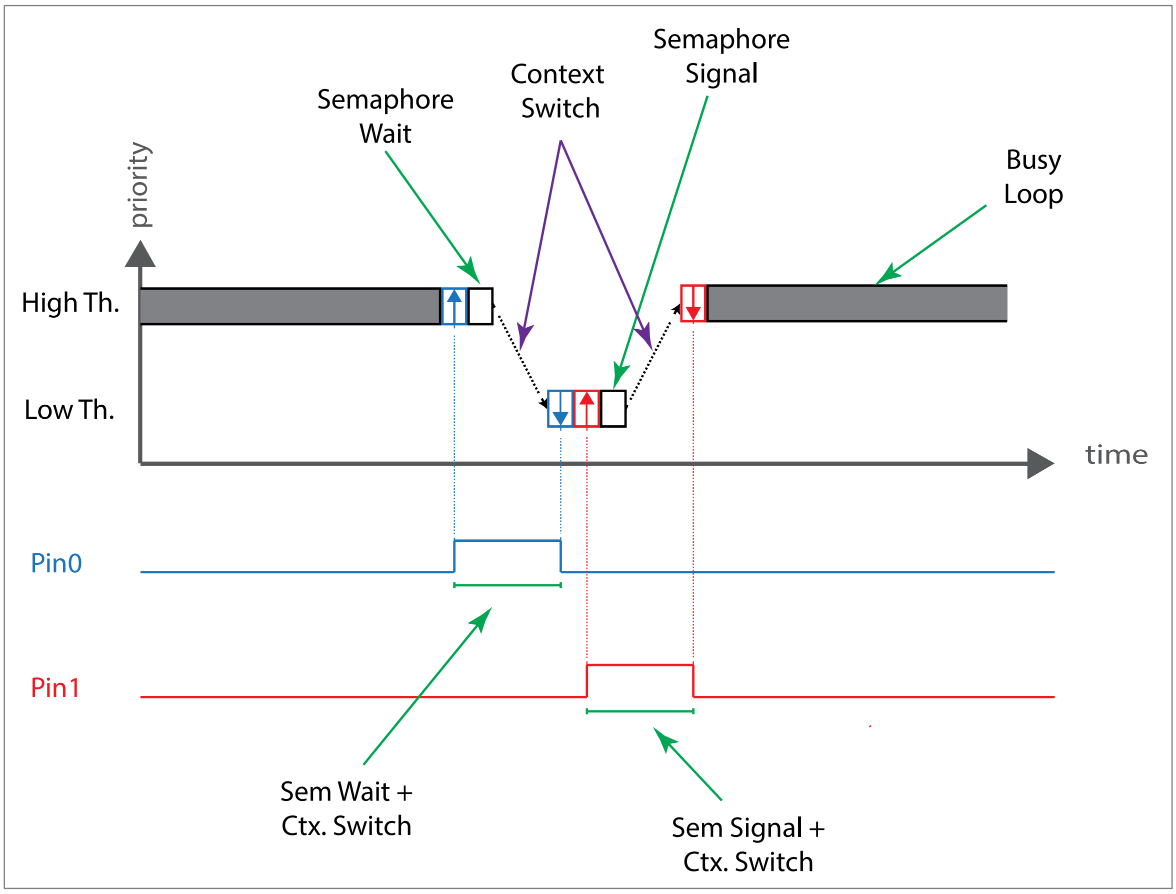

Figure 14. Detail of Context Switch

6.4 Results

The main results for the timing measurements are presented in Table 9. The values shown represent the average of the measurements taken during the scheduling cycles performed in a time frame of 100 ms. As the deviation from the average is less than the error limit of 46 ns, in all cases, it is omitted from the table. It is important to highlight that the developed scheduler, referred as "Ell-i" in the results table, have a performance between the two other analysed RTOSs for both tests: it is faster than CoOS but slower than ChibiOS/RT.

However, there is a significant difference in comparison with the timings of the other systems: the time needed to perform the Wait operation is longer than the one needed for the Signal operation. The possible reasons for this difference will be discussed in the following section. Aside from that difference, it is notable that ChibiOS/RT performs the Signal operation in only 80.5% of the Ell-i time, and the Wait operation in only 53.0 % of Ell-i time. It is even more relevant when comparing it with the time needed to save and restore the 17 registers of the full context: 1.271 ms [4]. Further analysis of the reasons behind this enhanced performance will be treated in the next section.

[4] 61 cycles (1271 ns at 48Mhz) = 15 cycles for exception entry + 16 cycles for exception return + 15 cycles for saving software registers (6 cycles for a STM instruction with 5 registers, 4 cycles for 4 MOV instructions, and 5 cycles for another STM with 4 registers) + 15 cycles for restoring software registers (same procedure but with the LDM instruction).

	ChibiOS/RT	*Ell-i*	*CoOS*
Semaphore Signal + CtxSwt (us)	4.79	5.95	10.83
Semaphore Wait with Blocking + CtxSwt (us)	3.19	6.02	9.29

Table 9. Timing Results

Apart from the timing analysis, the memory requirements, both Flash and RAM, are of high importance for the present study. The reason is that the scheduler was specifically designed to enable threading in the low-end MCUs based on the ARM Cortex-M0, which are very limited in memory for cost reasons. For a detailed and fair comparison, the Flash occupancy was calculated by grouping program objects and assigning them to specific constituting elements in order to isolate the RTOS code from the user application code. The information for the Flash memory is detailed in Table 10.

As in the timing tests, the Ell-i scheduler, together with all the necessary initialization routines and user code, takes slightly more program memory than ChibiOS/RT (5.66% more) and considerably less than CoOS (32.93% less). In concrete, isolating only the RTOS code, ChibiOS/RT uses only 77.11% of the memory required by Ell-i. Conversely, CoOS uses 57.62% more memory than the needed by the Ell-i core.

However, the RTOS program code is not the only factor that affects the final binary size, the user application is also relevant. However, as the functionality is identical in the three cases, the size of the user application allows the analysis of the memory efficiency of the system APIs. In that respect, the designed scheduler outperforms the other two, requiring only 91.34% of the space used by ChibiOS/RT and only 86.36% of the Flash requirements of CoOS.

	ChibiOS/RT	*Ell-i*	*CoOS*
Total (Bytes)	1800 • 1784, text • 16, data	1902 * • 1890, text • 12, data * Without automatic peripheral initialization, for fair comparison	2836 • 2824, text • 12, data
Data (Bytes)	16	12	12
System Initialisation (Bytes)	468 • 196, g_pfnVectors • 84, Reset_Handler • 188, System (2)	468 • 196, g_pfnVectors • 84, Reset_Handler • 188, System (2)	468 • 196, g_pfnVectors • 84, Reset_Handler • 188, System (2)
Standard Libraries (Bytes)	158 • 158, div (6)	0	194 • 194, div (6)

User Application (Bytes)	208 • 164, main (1) • 44, Thread (1)	190 • 158, main (1) • 32, Thread (1) ,	220 • 140, main (1) • 80, Thread (2)
RTOS (Bytes)	950 • 20, SysTick_Handler • 14, NMI_Handler • 916, Rest	1232 • 24, SVC_Handler • 72, PendSV_Handler • 84, SysTick_Handler • 1052, Rest	1942 • 70, PendSV_Handler • 88, SysTick_Handler 1784, Rest

Table 10. Flash Memory Results

Regarding the RAM memory usage, it is necessary to conceptually divide between the dynamic stack space requirements and the statically allocated variables used by the internal kernel operation. From the stack requirement perspective, it is almost entirely related with the user application, and only affected by the underlying RTOS by three specific aspects. First, stack space for interrupts handling needs to be allocated somewhere: either by adding some extra space in each user stack, or by concentrating all interrupts in an independent section. Second, the RTOSs also requires a stack for the Idle Thread, which is additional to all other user stacks. Third, in case the main line of execution is abandoned after the threading functionality is enabled, the stack used by this 'initial' thread is thereafter unused.

The results of the stack analysis are presented in the last row of Table 11. In relation with the allocation of stack space for interrupts, the Ell-i scheduler is the only one to allocate a specific stack for this purpose. As the other systems lack of this feature, the size calculation for user threads' stacks should also consider the possibility of interrupts being handled in addition to the normal flow of code.

Regarding the Idle Thread, the stack form Ell-I is the smallest, with only 96 bytes in comparison with 100 bytes from CoOS and 176 bytes from ChibiOS/RT. In addition, ChibiOS/RT and Ell-i transform the main line of execution into a thread, requiring only two stacks for the two running threads. In contrast, CoOS 'abandons' the main thread, requiring one additional stack for the initialisation thread (resulting in three independent stacks).

As discussed earlier, the RAM usage also depends on the statically allocated variables for storing internal data of the RTOS system. These variables are assigned to different code sections depending on the initial data they contain. Zero initialised variables are assigned to the .bss section, while variables with any other value are assigned to the .data section. As shown in the first row of Table 11, Ell-i requires only 52 bytes of combined allocation (.data + .bss). In contrast, ChibiOS/RT requires 80 bytes. This is an important figure as the data structures from Ell-i are more compact, what provides additional benefits when scaling to a larger number of threads.

Unfortunately, the 'minimal' configuration of CoOS still contains variables for event handling and other functions. This exemplifies one of the main problems with reduced configurations, as removing part of the system logic does not translate into resource saving (both, memory footprint and execution speed).

	ChibiOS/RT	*Ell-i*	*CoOS*
Total (Without Stacks Space) (Bytes)	80 • 16, data • 64, bss	52 * • 12, data • 40, bss * Without automatic peripheral initialization, for a fair comparison	668 * • 12, data • 656, bss * * Includes "events" related unused variables, which is non-minimal, and prevents a fair comparison
Data (Bytes)	16	12	12
Bss (Bytes)	64	40	656
Stacks	3 • Idle Thread (system) o 176 bytes • Low Prio. Thread (user) • Main, which converts to High Prio. Thread (system / user)	4 • Idle Thread (system) o 96 bytes • Low Prio. Thread (user) • Main, which converts to High Prio. Thread (system / user) • Interrupt Stack (system / user)	4 • Idle Thread (system) o 100 bytes • Low Prio. Thread (user) • High Prio. Thread (user) • Main, which is abandoned after threading starts (system)

Table 11. RAM Memory Results

6.5 Discussion

Before proceeding to examine the results, it is necessary to distinguish the code optimization level of the analysed RTOSs. In general, the Open Source RTOSs analysed have gone through years of crafting and incremental improvements. In comparison, the development for the Ell-i system was almost halted as soon as the desired functionality was achieved. This fact is a side consequence of the limited development time frame, usual for Master thesis related projects. However, the author is quite certain that after incremental improvements and revision from external developers, the performance of the system can be dramatically improved.

Another factor which influences the validity of the measurements is that the definition of "minimal configuration" for each RTOS, which is non-trivial and debatable. In fact, there was a discussion between the author of this document and the developers of FreeRTOS with the objective of granting permission to publish the timing and memory comparisons. The main reason for denying the permission was precisely the concept of "minimal configuration", as the FreeRTOS creators considered that removing functionality to improve performance metrics actually degrades the overall quality and usability provided by the standardly configured system.

The following is a brief analysis of the possible underlying reasons driving the obtained results. First, factors related to timing performance are discussed, followed by memory occupancy aspects.

Timing

In the other analysed systems, the time taken by the Wait operation is always shorter than the time required for the Signal operation ($T_{wait} < T_{signal}$). For ChibiOS/RT, the Wait latency is only 66.59% of the Signal latency; similarly, for CoOS it is 85.78. However, on the Ell-i scheduler, the time required for the wait operation is almost equal to the Signal operation ($T_{wait} = T_{signal}$). In fact, the Wait operation is 1.17% longer than the Signal operation. The underlying reason is just an oddity from the particular thread configuration used for the test. Theoretically, the execution time of the Signal operation should be larger than the Wait operation for the average and worst cases.

The *sched()* and the *find_waiting_thread()* functions in Snippet 8 can have a worst case execution time of a constant multiplied by the number of threads in the system. However, the two threads configuration, with the highest priority (HP) blocking on a semaphore, leads to a condition where the execution time of the Signal operation is almost equivalent in cost to the Wait operation. The particular branching of the original code that leads to these specific cases has been removed for simplicity of the pseudo code.

Signal (unblocking HP thread)

```
thread_t *new;
thread_t *curr;
semaphore_t *sem;

thread_t *waiting;

/* The worst case of this function
 * is the number of threads in the
 * system (2 in this case), however
 * as the HP thread is the one
 * blocked in the experiment, it
 * always has cost of 1, as it
 * stops at the first element
 * because of the priority order.
 */
waiting = find_waiting_thread(sem);

/* unblock HP thread */
waiting->sem = NULL;

/* The worst case of this function
 * is also the number of threads in
 * the system (2). As the HP thread
 * has been just unblocked in the
 * previous instruction, the
 * iteration will stop also at the
 * first element (HP thread), also
 * leading to a cost of 1.
 */
sched();
```

Wait (blocking HP thread)

```
thread_t *new;
thread_t *curr;
semaphore_t *sem;

/* block HP thread */
curr->sem = sem;

/* The worst case of this function
 * is the number of threads in the
 * system (2). As the HP thread has
 * been just blocked in the previous
 * instruction, the iteration will
 * stop at the second element (LP
 * thread), leading to a cost of 2.
 */
sched();
```

Snippet 8. Conceptual Semaphore Wait and Signal Pseudocode

As discussed earlier, it is remarkable that ChibiOS execution time is almost half of the required by the Ell-I scheduler for the Wait operation. The underlying reason is that ChibiOS uses *voluntary preemption* to reduce almost by half the number of registers that need to be saved and restored. This is possible because the point where the semaphore APIs are called within the thread execution is carefully crafted to obviate the necessity of saving the processor state and scratch registers. With this strategy, it is not necessary to use the automatic context saving mechanism of the ARM interrupt controller, allowing a pure software strategy which does not require any system interrupts such as PendSV or SVCall.

However, the *voluntary preemption* strategy is only useful when the context switch is between two threads. In case a semaphore is released from an interrupt context, it is necessary to save and restore the full context frame and, additionally, use any of the system interrupts (PendSV, SVCall, or NMI). In that regard, it is necessary to add logic to distinguish if a system API is being called from an ISR or from normal context. As a consequence, ChibiOS provides additional constraints to user defined ISRs which call system APIs (in particular, it is necessary to add a 'prologue' code that reads the LR register to detect from which context the APIs will be called).

Another timing property that is relevant for Real Time applications is the latency generated by blocking segments of kernel code. As an example, Ell-i and ChibiOS are compared in regard to the blocking time generated by a context switch from a semaphore wait system call (Figure 14). This system API can have different execution alternatives, depending of the state of the semaphores and threads at the time of the call (the shorter alternatives are depicted in smaller scale on the left side).

While ChibiOS performs the context switch in a single blocking event, Ell-i divides the call in up to three segments. The latter behaviour considerably improves latency, as incoming interrupts can be handled in between this segments, and consequently, sooner. However, this allows a more complex interleaving of events, as, for example, interrupts can modify the desired target thread of the context switch. Hopefully, the Ell-i scheduler is carefully crafted to handle this specific conditions, due to the state space analysis described in section Context Switch State Space

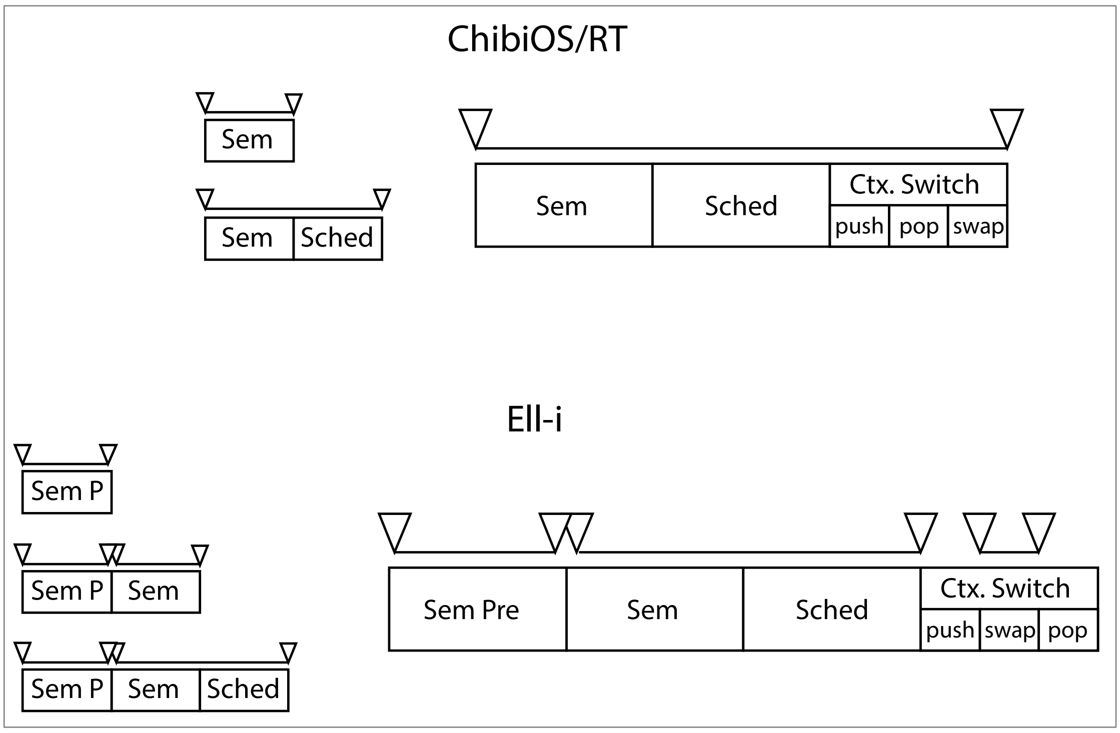

Figure 15. Blocking Time Diagram for Wait Operation for ChibiOS and Ell-i

Memory

The overall program memory (Flash) footprint of Ell-i is sufficiently small in comparison with the analysed RTOSs set, although it is slightly larger than ChibiOS. However, the RAM occupancy is considerably smaller in comparison with the closest counterpart (ChibiOS). This is really important, as

RAM is the main pricing factor, among the memories, and also in respect with the other MCU's components.

Another important feature of the Ell-i scheduler, also shared by ChibiOS, is the ability to statically initialise threads and semaphores (at compile time). In contrast, CoOS is only capable of initialising this elements at runtime. In fact, ChibiOS is capable of both, runtime and static initialisation. Although initialising elements at runtime provides flexibility, for memory constrained applications, it generates considerable costs. Initially, additional RAM is necessary for this initialisation process. Second, it is not possible to realise safety checks (e.g. data allocation or stack availability) at compile time. If this is required, additional program memory and RAM is necessary to allocate runtime safety logic.

Finally, protection against stack overflow is very important for real-life applications. As the worst-case stack requirements of an application are independent of the RTOS, the system can only provide two guarantees: that the space requested by the user is really allocated, and that the system will be as resilient as possible in case of stack overflow. As discussed above, it is possible to confirm memory availability for the required stack sizes at compile time (as Ell-i scheduler does), or by careful runtime allocation. For resiliency on a stack overflow event, the best alternative is to provide a stack layout that protects the system. In this regard, Ell-i scheduler is designed to allow overflow of the interrupt stack without effects. This feature is relevant because calculating the stack size requirements for interrupts involves analysis of the worst-case interrupt nesting, which is fairly complex and prone to underestimation.

Chapter 7 - Conclusions and Future Work

This main objective of this thesis was to test the feasibility of enabling multithreading in the cheapest low-end ARM Cortex-M0 MCUs, by designing and implementing a memory constrained scheduler. In this regard, competitive metrics were achieved in comparison with popular Open Source RTOSs. In particular, significant figures in performance, memory footprint and safety were obtained, evaluated and discussed.

The evaluation of performance and memory footprint was relatively straightforward, as the resulting metrics were completely objective. However, they depended on selecting an adequate workload. Even if, ideally, this workload would simulate a practical, common use case, using a very complex scheme could also prevent the isolation specific variables. The selected workload was considered adequate as it balanced between real applications while still allowed to understand the underlying factors. Additionally, it was the same workload used for the similar purposes in the revised literature.

It is also important to highlight that the timing measurements were more dependent on the workload than the footprint measurements. The underlying reason is that the memory analysis was done at compile time, and thus it was possible to separate the contribution of application data from system data. In contrast, timing measurements were extracted from runtime execution, and also depended on the measuring hardware framework.

As discussed above, safety metrics were also used to characterise the developed scheduler and the selected Open Source RTOSs. However, these metrics were more subjective, as the testing framework could not directly measure the 'safeness' or 'resilience' of the system. Nevertheless, by providing a detailed analysis, it was possible to distinguish safe practices. Some examples are: statically allocating space for stacks and system data structures, or analysing the possible interleaving of hardware events and system API calls generated by the Cortex-M0 interrupt controller.

As a summary, the designed scheduler usage of RAM was considerably lower, while Flash occupancy was close to the optimum. In regards to timing performance, it was not optimal as the scheduler did not took advantage of the *voluntary preemption* concept used by other systems. In terms of 'safeness', the Ell-i system was definitively safer than any other system, due to the usage of static allocation of all system data structures and user stacks, as well as the analysis of the hardware interrupt state space.

Turning now to the business side, the objective of providing Open Source Software that fits within the business model of Ell-i was also achieved. By collectively owning the copyright of the designed scheduler, the possibility of licencing the complete software stack to companies became more viable. Additionally, the scheduler is specially tailored to power the prototype Ell-i PoE node. This allows a complete package of Open Source Hardware tightly coupled with Open Source Software that can be further licensed to companies, or used to showcase the capabilities of the node.

Before proceeding to examine the directions of future work, it is necessary to relate the results of this work to a broader scope. Before a technological breakthrough improves the performance of memory, particularly RAM, it will be the bottleneck of computer hardware. Considering this constraint, any system that provides the desired functionality with limited memory usage is relevant. Furthermore, there are multiple applications which are simpler to implement when deployed over a threading system. This includes networking and sensing applications, systems with real-time constraints, and also legacy code that was originally targeted to 8bits or 16bits MCUs and requires porting to modern architectures.

As discussed earlier, this development was targeted to the ‘Internet of Wired Things’. However, there are no technical limitations that prevent its use into generic IoT systems, particularly wireless systems. Even if the system was targeted to PoE devices, which are not energy constrained, the fundamental decisions regarding safety and reduced memory footprint apply. Other design decisions, like the ones regarding system interrupts or clocks configuration, might need re-evaluation for energy constrained systems. Nevertheless, the implemented scheduler could be used as a basis for a wireless oriented Open Source RTOS.

With respect to future directions of work, there are plenty of design and implementation decisions that could be reconsidered, in addition to multiple possible testing framework changes. However, it is important summarize them, and only highlight the ones that are important for this thesis. Three elements for future work were selected and are discussed below.

The main design change would consist on taking advantage of the *voluntary preemption* when there is a context switch between threads. This was the reason why ChibiOS achieved better timing performance than the Ell-i system, even when the data structures from the former are more complex than the ones from the latter. With this change, it is expected that Ell-I would achieve the fastest context switch among the systems evaluated for this thesis.

Another direction of future work is related to testing. Although there are multiple elements that could be improved, two changes are considered for discussion. First, it would be ideal to test the selected RTOSs with more threads, and compare the behaviour. This is important because scalability is a very important measurement of performance that was not evaluated. Second, another relevant use case that was not revised consists of switching between an ISR and a thread (or vice versa). This is also relevant as, apart from providing a figure for a different kind of context switch, it could offer some insight about the safety of the system, as this interacts with the hardware interrupt controller.

The last point is related to re-evaluating the design decisions with different constraints. In concrete, it would be interesting to add limitations to power consumption, as battery life is of high importance for wireless applications. This could provide new directions of development, which could target more general IoT systems, and, possibly, offer new ideas for improvements that also benefit the wired applications, including but not limited to PoE.

References

[1] B. Perens, “Open Source Definition,” in *Open sources: Voices from the open source revolution*, C. DiBona and S. Ockman, Eds., O'Reilly Media, Inc., 1999.

[2] E. Raymond, “The cathedral and the bazaar,” *Knowledge, Technology \& Policy,* vol. 12, no. 3, pp. 23-49, 1999.

[3] S. J. Vaughan-Nichols, “UK's security branch says Ubuntu most secure end-user OS,” 17 Jaunuary 2014. [Online]. Available: http://www.zdnet.com/uks-security-branch-says-ubuntu-most-secure-end-user-os-7000025312/.

[4] Red Hat, Inc., “Red Hat Reports Fourth Quarter and Fiscal Year 2013 Results,” 27 March 2013. [Online]. Available: http://investors.redhat.com/releasedetail.cfm?ReleaseID=751668.

[5] Red Hat, Inc., “Investor FAQs,” 2014. [Online]. Available: http://investors.redhat.com/faq.cfm.

[6] I. Murdock, “Open Source and the Commoditization of Software,” in *Open sources: Voices from the open source revolution*, C. DiBona and S. Ockman, Eds., O'Reilly Media, Inc., 1999.

[7] J. Lock, “Open Source Hardware,” 2013.

[8] R. Acosta and others, “Open source hardware,” 2009.

[9] E. Rubow, “Open source hardware,” 2008.

[10] M. N. Asay, “Open Source and the Commodity Urge: Disruptive Models for a Disruptive Development Process,” in *Open sources 2.0: The continuing evolution*, C. DiBona, M. Stone and D. Cooper, Eds., " O'Reilly Media, Inc.", 2005.

[11] F. Hecker, “Setting up shop: The business of open-source software,” *IEEE software,* vol. 16, no. 1, pp. 45-51, 1999.

[12] S. Weber, The success of open source, vol. 368, Cambridge Univ Press, 2004.

[13] TechTarget, “IBM acquires Gluecode, fuels open source,” 10 May 2005. [Online]. Available: http://searchsoa.techtarget.com/news/1087132/IBM-acquires-Gluecode-fuels-open-source. [Accessed 2014].

[14] IDC Corporate USA, “IDC: Smartphone OS Market Share,” 2014. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp. [Accessed 2014].

[15] M. Olson, “Dual licensing,” *open sources,* vol. 2, pp. 71-90, 2005.

[16] Raspberry Pi Foundation, “Frequently Asked Questions - I want to be a Raspberry Pi reseller,” 2014. [Online]. Available: http://www.raspberrypi.org/help/faqs/#buyingReseller. [Accessed 2014].

[17] Arduino, “Arduino Ethernet Shield,” 2014. [Online]. Available: http://arduino.cc/en/Main/ArduinoEthernetShield. [Accessed 2014].

[18] Arduino, “Arduino Ethernet,” 2014. [Online]. Available: http://arduino.cc/en/Main/ArduinoBoardEthernet. [Accessed 2014].

[19] J. Restakis, Humanizing the Economy: Co-operatives in the Age of Capital, New Society Publishers, 2013.

[20] D. G. Olson and J. Logue, *Fix Globalization: Make it more inclusive, democratic, accountable and sustainable,* 2002.

[21] E. v. Hippel and G. v. Krogh, “Open source software and the “private-collective” innovation model: Issues for organization science,” *Organization science,* vol. 14, no. 2, pp. 209-223, 2003.

[22] A. Hars and S. Ou, “Working for free? Motivations of participating in open source projects,” in *System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on*, 2001.

[23] W. Ke and others, “Motivations for participating in open source software communities: Roles of psychological needs and altruism,” 2008.

[24] A. H. Maslow, “A theory of human motivation.,” *Psychological review,* vol. 50, no. 4, p. 370, 1943.

[25] A. Burns, “Scheduling hard real-time systems: a review,” *Software Engineering Journal,* vol. 6, no. 3, pp. 116-128, 1991.

[26] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM),* vol. 20, no. 1, pp. 46-61, 1973.

[27] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell and A. J. Wellings, “Fixed priority pre-emptive scheduling: An historical perspective,” *Real-Time Systems,* vol. 8, no. 2-3, pp. 173-198, 1995.

[28] E. Bini, G. C. Buttazzo and G. M. Buttazzo, “Rate monotonic analysis: the hyperbolic bound,” *Computers, IEEE Transactions on,* vol. 52, no. 7, pp. 933-942, 2003.

[29] L. Sha, T. Abdelzaher, K.-E. {\AA}rz{\'e}n, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky and A. K. Mok, “Real time scheduling theory: A historical perspective,” *Real-time systems,* vol. 28, no. 2-3, pp. 101-155, 2004.

[30] Y. Wang and M. Saksena, “Scheduling fixed-priority tasks with preemption threshold,” in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, 1999.

[31] R. J. Bril, J. J. Lukkien and W. F. Verhaegh, “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption,” *Real-Time Systems,* vol. 42, no. 1-3, pp. 63-119, 2009.

[32] R. J. Bril, J. J. Lukkien and W. F. Verhaegh, “Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited,” in *Real-Time Systems, 2007. ECRTS'07. 19th Euromicro Conference on*, 2007.

[33] A. Moitra, “Voluntary preemption: A tool in the design of hard real-time systems,” in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, 1991.

[34] J. Schneider, “Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems,” in *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, 2000.

[35] J. S. Snyder, D. B. Whalley and T. P. Baker, “Fast context switches: Compiler and architectural support for preemptive scheduling,” *Microprocessors and Microsystems,* vol. 19, no. 1, pp. 35-42, 1995.

[36] ARM Holdings, “Procedure Call Standard for the ARM Architecture,” 30 Nov 2012. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/IHI0042E_aapcs.pdf. [Accessed 2014].

[37] V. Barthelmann, “Inter-task register-allocation for static operating systems,” in *ACM SIGPLAN Notices*, 2002.

[38] E. W. Dijkstra, “Cooperating sequential processes (EWD-123). EW Dijkstra Archive,” *Center for American History, University of Texas at Austin,* 1965.

[39] L. Sha, R. Rajkumar and J. P. Lehoczky, “Priority inheritance protocols: An approach to real-time synchronization,” *IEEE Transactions on computers,* vol. 39, no. 9, pp. 1175-1185, 1990.

[40] K. M. Zuberi and K. G. Shin, “An efficient semaphore implementation scheme for small-memory embedded systems,” in *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, 1997.

[41] A. M. Cheng and F. Jiang, “An Improved Priority Ceiling Protocol to Reduce Context Switches in Task Synchronization,” 2005.

[42] T. P. Baker, “A stack-based resource allocation policy for realtime processes,” in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, 1990.

[43] Arduino, “Arduino - Introduction,” 2014. [Online]. Available: http://www.arduino.cc/en/Guide/Introduction. [Accessed 2014].

[44] D. Kushner, “The Making of Arduino,” 26 Oct 2011. [Online]. Available: http://spectrum.ieee.org/geek-life/hands-on/the-making-of-arduino. [Accessed 2014].

[45] Arduino, “Arduino - Home,” 2014. [Online]. Available: http://arduino.cc/.

[46] Processing, “Processing2 - Overview,” 2014. [Online]. Available: https://processing.org/overview/. [Accessed 2014].

[47] Wiring, “Wiring - About,” 2014. [Online]. Available: http://wiring.org.co/about.html. [Accessed 2014].

[48] Creative Commons, “Attribution-ShareAlike 4.0 International (CC BY-SA 4.0),” [Online]. Available: http://creativecommons.org/licenses/by-sa/4.0/. [Accessed 2014].

[49] Arduino, “Frequently Asked Questions,” 2014. [Online]. Available: http://arduino.cc/en/Main/FAQ. [Accessed 2014].

[50] GNU Operating System, “GNU Lesser General Public License,” 12 Apr 2014. [Online]. Available: https://www.gnu.org/licenses/lgpl.html. [Accessed 2014].

[51] P2P Foundation, “Arduino - Business Model,” 3 Apr 2011. [Online]. Available: http://p2pfoundation.net/Arduino_-_Business_Model. [Accessed 2014].

[52] Adafruit Industries, “Million dollar baby – Businesses designing and selling open source hardware, making millions,” 2010. [Online]. Available: http://www.adafruit.com/pt/fooeastignite2010.pdf. [Accessed 2014].

[53] R. MacDonald, “Open source hardware worth $1billion by 2015,” Linux User & Developer, 11 Jun 2014. [Online]. Available: http://www.linuxuser.co.uk/news/open-source-hardware-worth-1billion-by-2015. [Accessed 2014].

[54] Arduino Team, “Open Source Hardware Summit Speech 2011, Presentation Slides,” 16 Sep 2011. [Online]. Available: http://www.slideshare.net/arduinoteam/open-source-hardware-summit-speech-2011. [Accessed 2014].

[55] ARM Holdings, “Product Backgrounder,” Dec 2004. [Online]. Available: https://web.archive.org/web/20050119214204/http://www.arm.com/miscPDFs/3823.pdf. [Accessed 2014].

[56] IC Insights, “MCU Market on Migration Path to 32-bit and ARM-based Devices,” 25 Apr 2013. [Online]. Available: http://www.icinsights.com/news/bulletins/MCU-Market-On-Migration-Path-To-32bit-And-ARMbased-Devices/. [Accessed 2014].

[57] J. Yiu, "ARM Microcontroller Updates - Markets, Technologies and Trends," 15 Jul 2013. [Online]. Available: http://www.arm.com/zh/files/event/20130715_AES_Joseph.pdf. [Accessed 2014].

[58] S. Deutsch , "Microcontroller Maniacs Rejoice: Arduino Finally Releases the 32-Bit Due," 20 Oct 2012. [Online]. Available: http://www.wired.com/2012/10/arduino-due/. [Accessed 2014].

[59] Arduino, "Arduino Due," 2014. [Online]. Available: http://arduino.cc/en/Main/arduinoBoardDue. [Accessed 2014].

[60] Arduino, "Arduino Zero," 2014. [Online]. Available: http://arduino.cc/en/Main/ArduinoBoardZero. [Accessed 2014].

[61] Arduino, "Arduino WiFi Shield," 2014. [Online]. Available: http://arduino.cc/en/Main/ArduinoWiFiShield. [Accessed 2014].

[62] Arduino, "Arduino Boards," 2014. [Online]. Available: http://store.arduino.cc/category/11. [Accessed 2014].

[63] Arduino, "Shields / Official," 2014. [Online]. Available: http://store.arduino.cc/category/5. [Accessed 2014].

[64] Arduino, "Arduino Yún," 2014. [Online]. Available: http://arduino.cc/en/Main/ArduinoBoardYun. [Accessed 2014].

[65] Arduino, "Arduino Build Process," 2014. [Online]. Available: http://arduino.cc/en/Hacking/BuildProcess. [Accessed 2014].

[66] Arduino, "Arduino - Reference," 2014. [Online]. Available: http://arduino.cc/en/Reference/HomePage.

[67] Arduino, "Arduino - Libraries," 2014. [Online]. Available: http://arduino.cc/en/Reference/Libraries. [Accessed 2014].

[68] ARM Holdings, "ARMv6-M Architecture Reference Manual," September 2010. [Online]. Available: http://ecee.colorado.edu/ecen3000/labs/lab3/files/DDI0419C_arm_architecture_v6m_reference_manual.pdf. [Accessed 2014].

[69] ARM Holdings, "ARMv7-M Architecture Reference Manual," February 2010. [Online]. Available: https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf. [Accessed 2014].

[70] STMicroelectronics, "STM32F051xx Datasheet," January 2014. [Online]. Available: http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00039193.pdf. [Accessed 2014].

[71] STMicroelectronics, “RM0091 - Reference Manual,” May 2014. [Online]. Available: http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031936.pdf. [Accessed 2014].

[72] M. Eisen, “Introduction to PoE and the IEEE802. 3af and 802.3 at Standards,” *presentation slideware,* 2010.

[73] “IEEE Standard for Information technology-- Local and metropolitan area networks-- Specific requirements-- Part 3: CSMA/CD Access Method and Physical Layer Specifications Amendment 3: Data Terminal Equipment (DTE) Power via the Media Dependent Interface (MDI) Enhancements,” *IEEE Std 802.3at-2009 (Amendment to IEEE Std 802.3-2008),* pp. 1-137, Oct 2009.

[74] Microchip, “ENC28J60 Stand-Alone Ethernet Controller with SPI Interface,” 2012. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/39662e.pdf. [Accessed 2014].

[75] Z. Shelby, K. Hartke and C. Bormann, *The Constrained Application Protocol (CoAP),* IETF, 2014.

[76] Arduino, “Compare board specs,” 2014. [Online]. Available: http://arduino.cc/en/Products.Compare. [Accessed 2014].

[77] P. Nikander, B. Nyman, T. Rinta-aho, S. D. Sahasrabuddhe and J. Kempf, “Towards software-defined silicon: Experiences in compiling Click to NetFPGA,” in *1st European NetFPGA Developers Workshop, Cambridge, UK*, 2010.

[78] P. Nikander, “Ell-i Runtime Designe - Static initialisation,” 24 May 2013. [Online]. Available: https://github.com/Ell-i/Hackathon/wiki/ELLi-runtime-design#static-initialisation. [Accessed 2014].

[79] ARM Holdings, “Cortex-M0 Devices Generic User Guide,” 2009. [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0497a/BABBGBEC.html. [Accessed 2014].

[80] ChibiOS/RT, “Documentation and Guides,” 2014. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:documents.

[81] Real Time Engineers Ltd., “About FreeRTOS,” 2013. [Online]. Available: http://www.freertos.org/RTOS.html.

[82] Micrium, “uC/OS-II Specifications,” 2014. [Online]. Available: http://micrium.com/rtos/ucosii/specifications/. [Accessed 2014].

[83] Micrium, “uC/OS-III Specifications,” 2014. [Online]. Available: http://micrium.com/rtos/ucosiii/specifications/. [Accessed 2014].

[84] L. Otava, “Analysis of Selected RTOS Characteristics,” 2012.

[85] CooCox, “Free/Open ARM Cortex MCU Development Tools,” 2014. [Online]. Available: http://www.coocox.org/CoOS.htm.

[86] G. Ugurel and C. Bazlamacci, “Context switching time and memory footprint comparison of Xilkernel and μC/OS-II on MicroBlaze,” in *Electrical and Electronics Engineering (ELECO), 2011 7th International Conference on*, 2011.

[87] ChibiOS/RT, “License,” 8 Jun 2013. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:license. [Accessed 2014].

[88] Real Time Engineers Ltd., “License Details,” 2013. [Online]. Available: http://www.freertos.org/a00114.html. [Accessed 2014].

[89] CooCox, “CoOS Terms and Conditions,” 2011. [Online]. Available: http://www.coocox.org/policy.htm. [Accessed 2014].

[90] CooCox, “CoOS / License,” 19 Apr 2012. [Online]. Available: https://github.com/coocox/CoOS/blob/master/LICENSE. [Accessed 2014].

[91] eCos, “eCos License Overview,” 2009. [Online]. Available: http://ecos.sourceware.org/license-overview.html. [Accessed 2014].

[92] Canonical Ltd., “GNU Tools for ARM Embedded Processors,” 2014. [Online]. Available: https://launchpad.net/gcc-arm-embedded/. [Accessed 2014].

[93] ARM Holdings, “STM32F051xx,” Jan 2014. [Online]. Available: http://www.st.com/web/en/resource/technical/document/datasheet/DM00039193.pdf. [Accessed 2014].

[94] Saleae LLC, “Logic16 Tech Specs,” 2014. [Online]. Available: https://www.saleae.com/logic16/specs. [Accessed 2014].

[95] ChibiOS/RT, “Counting Semaphores, Binary Semaphores and Mutexes explained,” 5 Jun 2012. [Online]. Available: http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:semaphores_mutexes. [Accessed 2014].

[96] CooCox, “Intertask Synchronization,” 2011. [Online]. Available: http://www.coocox.org/CoOSGuide/CoOS_Semaphores.htm. [Accessed 2014].

Appendix A - Measurement of Hardware Exception Handling Times

This measurements were performed with the same equipment configuration as the one presented in section Hardware Testing Framework. The only difference is that the operation frequency of the MCU was decreased to 8 Mhz (by using the internal oscillator instead of the PLL as main clock source). With that modification, it is possible to measure with cycle precision regardless of the interference generated by the layers of peripherals and connections before the input of the Logic analyser.

For each measurement, a specific assembly code was executed in an endless iteration, in order to generate a waveform similar to the one described in the section 6.3 Workload. As it is not possible to only measure an isolated event (e.g. exception entry), a detailed calculation of the timing behaviour of the GPIO pin set and unset instruction was also realized to obtain precise measurements. It is important to highlight that this measurements, when analysed statistically, present no relevant variation apart from the error generated by the Logic analyser sample rate. This error was two orders of magnitude smaller than a single cycle time, and therefore is considered negligible.

Direct Measurements

Event	Cycles
Toggle	2
STR (instruction)	2
Exception Entry + Toggle	17
BX + Tail Chaining + Toggle	17
BX + Exception Return + Toggle	18
STR + PendSV Asynchronous Delay + Exception Entry + Toggle	21
SVC + Exception Entry + Toggle	20
STR + (1 + N) Cycles Count + Tim. Async. Delay +Exception Entry + Toggle	26 + N (N == 2) 25 + N (N > 2)

Indirect Measurements

Event	Cycles
Exception Entry + Toggle	15
BX + Tail Chaining	15
BX + Exception Return	16
PendSV Asynchronous Delay	2
SVC (instruction)	3
Timer Asynchronous Delay	6 (N == 2) 5 (N > 2)