FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Embedded OS for Kallisto

Margarida Pereira Marques

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: João Oliveira

Second Supervisor: Luís Almeida

October 28, 2021

Abstract

The Internet of Things (IoT) has led to an increasing number of connected devices with the need to run several applications concurrently. Thus, the software of low power embedded systems is becoming increasingly complex supported by improvements in CPU power and efficiency that allowed more and more features being implemented on the devices.

Consequently, developing firmware for these kind of devices on bare-metal is hard and not straightforward on bare-metal since there is the need for managing several tasks, communication and peripherals at the same time. An embedded and multitasking Operating System is convenient to ease this development.

Fraunhofer Portugal has an IoT solution composed of hardware, firmware and software named Kallisto that is fully build on top of bare-metal and which is starting to suffer with the problems referred above.

In this dissertation we analyse a set of Real-Time Operating Systems and compare their capabilities when installed on embedded devices.

The goal is not only to evaluate if a particular RTOS is a good alternative for Kallisto, but, from a more general point of view, to try to understand what are the key points when choosing an RTOS in the current panorama of IoT.

After analysing several OS, the primary selection included FreeRTOS, Zephyr and RIOT. FreeRTOS and Zephyr are both very mature operating systems, and even if FreeRTOS consists only on the Kernel, it is often used with the SoC of Kallisto. RIOT is a less mature but promising operating system, specially in what comes to the real-time behaviour. We evaluated the performance of the Kernel and their BLE stacks, also used by the Kallisto applications. Finally, we evaluated the RTOS using a Kallisto implementation. Taking into account the previous results, Zephyr seems to be the right decision for Kallisto, because its overhead in what comes to memory footprint, energy consumption and runtime performance compensate for the easiness of use and the amount of services it offers.

Acknowledgments

I would like to thank both my supervisors for the availability and the help they provided during the development of this thesis.

To Antonio and Catarina, for supporting me unconditionally and being always there. Without you, the hard times would have been much harder and the good times less happy.

Finally, to my family, for showing me that with resilience and calm, we eventually reach our goals. I hope I can keep growing and learning in the way you taught me to.

Margarida Marques

Contents

List of Figures

List of Tables

Abbreviation and Symbols

BLE	Bluetooth Low Energy
SoC	System-on-Chip
RTOS	Real-time Operating System
IoT	Internet of Things
MCU	Microcontroller Unit
CPU	Central Processing Unit
RAM	Random Access Memory
MMU	Memory Management Unit
MPU	Memory Protection Unit
ISR	Interrupt Service Routine
GPIO	General-Purpose Input/Output
API	Application Program Interface
HAL	Hardware Abstraction Layer
SDK	Software Development Kit
RTT	Real-Time Terminal
IDE	Integrated Development Environment

Chapter 1

Introduction

During recent years, a remarkable rise in what comes to the usage of embedded systems took place in a large range of application fields, from personal devices, such as wearable devices, to industrial and automotive processes, but also in fields like business, defence and entertainment. This is particularly noticeable in the context of the Internet of Things (IoT), where an application deals with an heterogeneous set of devices, many of them resource constrained, that communicate over the Internet to achieve the application goals.

The software running on these devices is becoming more and more complex, supported by the improvements in computational power, communication bandwidth and efficiency. Current IoT devices need to implement many features, but still under the typical constraints imposed by their embedded nature.

Many devices are programmed in a bare-metal implementation, i.e., without the support of any operating system (OS) thus having to deal with the details of the physical platform, and lacking generic services for managing and running applications. However, this is no longer suitable in many situations, due to the increasing complexity of the whole software. On the other hand, general purpose operating systems (GPOS) are not suitable either, because of the limited resources of the hardware platform. Embedded operating systems, instead, provide some features that are crucial in developing applications, by providing threading, portability, communication capabilities and improved security and privacy features, while dealing with the physical platform constraints.

1.1 Motivation

Fraunhofer Portugal has an IoT solution composed of hardware, firmware and software named Kallisto. It is based on the nRF52840 SoC, which is built around a 32-bit ARM® Cortex™-M4 CPU. This platform has been suffering from the issues presented above being programmed on bare-metal. As in many IoT applications, the main focus in our case is the data that the system can get from the environment around it, and this means that Kallisto must include functions for monitoring and controlling multiple processes. These usually have specific real-time requirements,

i.e., timing constraints that must be met, beyond the logical correctness of the applications, for the system to the effective.

Therefore, our research will focus on Real-Time Operating Systems (RTOS) that provide basic services to execute multi-tasking applications with a deterministic timing behaviour on MCUs, considering typical resource limitations of the hardware platform. The real-time constraints have a clear importance when selecting an OS, because of their timing performance in what comes to the latency in handling certain types of events and the predictability in the software execution. This latency might have a great impact on the precision of the data acquired by the system. However, an RTOS is not necessarily an embedded OS. In fact, it might be quite complex and need many resources. Our interest is specifically on embedded RTOS.

1.2 Objectives

This work aims at finding an embedded RTOS for Kallisto that deals with the complexity required by modern applications, including their real-time constraints and synchronisation requirements, but at the same time coping with limited resources, particularly imposing small memory footprint and energy consumption. The specific objectives of the dissertation are:

1. Selection of a short list of RTOS that support the Kallisto board, considering the characteristics that might be more relevant to the Kallisto applications, namely IoT-based;

2. Development of a benchmark and associated timing measurements that allow comparing the RTOS in the short list;

3. Porting of the selected RTOS to the Kallisto platform and practical measurement of the run-time performance of the operating systems, as another key factor of comparison and final decision;

4. Porting of key applications to the Kallisto system enhanced with the selected RTOS.

1.3 Structure of this dissertation

After this introduction, this dissertation starts by analysing, in Chapter 2, the meaning and the relevance of some specific features of operating systems, and their importance when choosing an operating system, not only according to the resources of the hardware platform but also the demands of the application. Then, it proposes a benchmark to compare the performance of the features of the RTOS.

Chapter 3 presents a comparative analysis of existing RTOS and proposes a short list of candidates, together with a brief explanation of their key points, that were determinant on the choice.

Chapter 4 presents the results of the tests performed, along with the conclusions that can be drawn.

Chapter 5 includes a small introduction to BLE behaviour and usage and then it introduces a benchmark for testing the BLE stacks that are used by each of the chosen RTOS, as well as the results from those tests.

Chapter 6 presents and explains the application chosen to be implemented on the different RTOS. Then, it introduces the benchmark that was used to compare those implementations and their behaviour and finally it presents the results of those tests, along with a critical assessment.

Finally, Chapter 7 sums up the previous chapters and concludes the dissertation.

Chapter 2

State of Art in RTOS

In many computing systems, the hardware comprises a relevant diversity of components with a wide range of functionalities. Desirably, the application programmers shall use hardware functionalities without knowing low-level programming details associated to that hardware. An operating system is an interface between applications and the hardware, with the purpose of abstracting hardware functionalities and simplifying and optimising their usage.

The operating systems typically implement services to manage the hardware, such as CPU management, memory management, file management and device management. In our dissertation, we focus on operating systems for embedded systems, and more specifically real-time operating systems. These will be defined later.

This chapter introduces embedded and real-time operating systems in detail, starting from their typical software architecture (Fig. 2.1) and describes the meaning and the relevance of some specific features of operating systems. It makes it clear that the choice, even if the features are essentially qualitative depends significantly on the system.

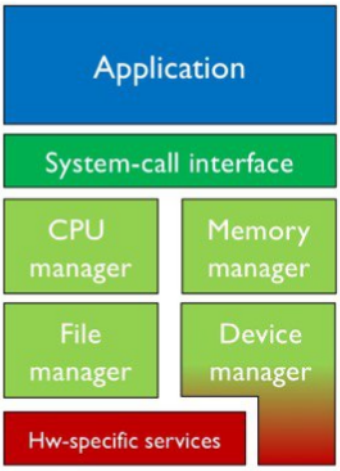

Figure 2.1: Generic software architecture of an embedded system. The green squares, between the hardware (red) and application (blue), are the services typically implemented by the operating system.

2.1 Embedded Operating Systems

Most programmable devices are embedded in some kind of larger system, i.e., the so-called embedded systems. In fact, Barr at [6] states that around 98% of the CPUs produced each year are used in embedded systems. One class of such devices that has received significant attention in recent times is the one that supports the Internet-of-Things.

We will use this class of devices as the target of our studies in this dissertation.

The authors Hahm et al. in [14] explain that there are two categories of IoT devices: low-end and high-end devices. The high-end ones include single-board computers and smartphones (Fig. 2.2) and the low-end are those with stronger limitations on resources, for instance the Arduino platforms (Fig. 2.3).

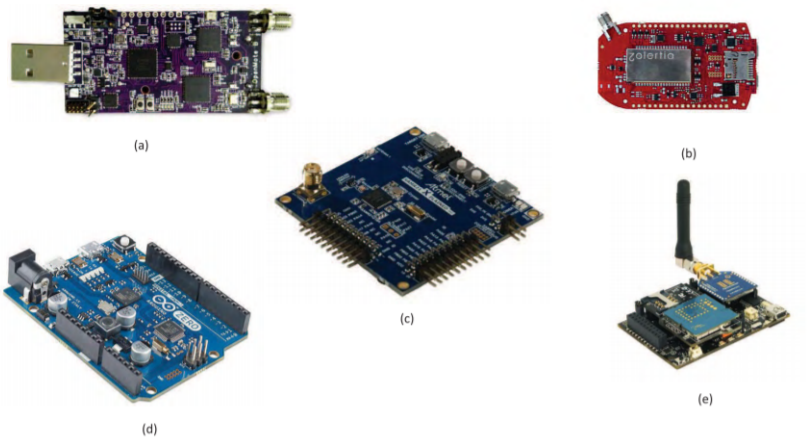

Figure 2.2: Examples of some low-end IoT devices: (a) OpenMote-B (b) Zolertia ReMote (c) Atmel SAM R21 (d) Arduino Zero (e) Waspmote. Picture taken from [24].

IoT interconnects both high-end devices, which use traditional operating systems such as Linux, and low-end devices. As said before, low-end devices have many resource constraints, such as limited memory, small computational power, and scarce power supply.

In some of these devices, the ones that have extreme resource constraints, the application software is typically executed on bare metal, i.e., without any software platform between the application and the hardware. Thus, the application software has to be tailored to the hardware specifications, making it hardware dependent and complicating its structure with concerns that are not part of the application functionality. Consequently, the applications are limited in structural complexity and they are highly specialised, in the way that they offer uniquely the services that the application uses.

Others, however, are not so constrained and support some level of hardware abstraction, which reduces specialisation and allows reuse of hardware features across applications and even execution of multiple applications. These systems benefit from, or even require, the usage of an OS to abstract hardware details and facilitate application development, deployment and control. OS services are made available to the application(s) through APIs (application programming interfaces)

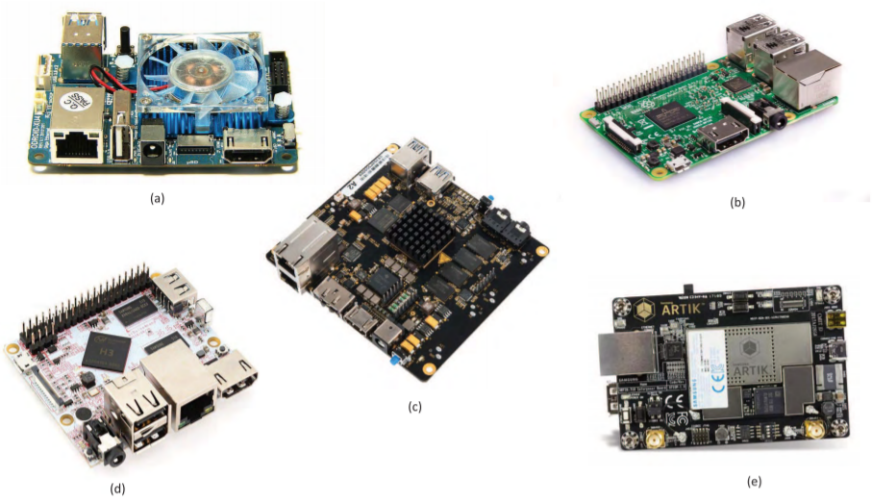

Figure 2.3: Examples of some high-end IoT devices: (a) Odroid-XU4 (b) Raspberry Pi 3 B+ (c) BeagleBone-X15 (d) pcDuino4 Nano (e) Samsung ARTIK 710. Picture taken from [24].

that facilitate the development, maintenance and scalability of application software for heterogeneous hardware platforms. The solutions of OS taken into account are focused on relatively simple 32-bit MCUs, such as the ones based on ARM Cortex Mx architectures. This means that, among other restrictions, the OS we will consider must have a small memory footprint and low energy consumption.

Processes need certain resources, such as CPU time, memory, files, I/O devices. These resources need to be given to the process, either during its execution or when the process is created. The OS is responsible for the process management, which, according to Silberschatz in [30], includes scheduling processes and threads on the CPU, creating and deleting processes, suspending and resuming processes, providing mechanisms both for process synchronisation and process communication.

In most applications, there is the need for memory management too, which consists on keeping track of which parts of the memory are currently being used and which processes are using them, deciding which processes and data to move in and out of memory and allocating and deallocating memory space as needed.

Other objective of the OS is to provide a uniform and logical view of the storage available, independently of its physical properties, i.e., the so-called storage management. This comprehends: file-system management, mass-storage management and caching. Another fundamental feature of an OS is I/O management. This is carried out by the so-called I/O systems that include the drivers for the specific hardware devices.

These services are commonly implemented by any OS, and in the majority of systems they are the justification for needing an OS in the first place.

2.2 Real-time Operating Systems

Most low-end devices, such as those used as nodes in wireless sensor networks (WSN), are equipped to sense or react to the environment. Some applications demand a deterministic and real-time response for communication, sensing and data processing tasks, as it is explained by Mathane and Lakshmi in [19].

RTOS are operating systems that facilitate meeting deadlines associated to the tasks. Missing these deadlines has effects that can go from undesired to catastrophic. RTOS must, thus, be deterministic and, frequently, preemptive. They must be designed in a way that they support the richest feature set possible, while not compromising the deadlines and predictability of the system. Note that the concept of predictability is orthogonal to the concept of "fast" responses.

According to Hambarde et al. in [15], the key features of an RTOS that determine its design are task priorities, which will determine which task will run in a certain moment, and also a reliable and sufficient inter-task communication, which ensures data integrity and a clear synchronisation among tasks. Besides this, there are other characteristics that can interfere in the predictability of the execution, such as the memory management. These will be explained in the next section.

2.3 Features of interest of embedded RTOS

In this section we visit the features of embedded RTOS that we consider more relevant for our purposes and which will support us in the choice for the most adequate RTOS.

From the point of view of the applications software architectures, support for concurrency is very important. Thus, two features take particular relevance, namely the model of the tasks, i.e., the way they are programmed, and the scheduling of the tasks, i.e., the sequence in which they are executed. Concerning the model, two main groups exist, namely basic and extended ones. A basic task is a sequence of statements executed once for each activation. This essentially corresponds to a software function that has the task code, and which is invoked every time the task is executed. An extended task, on the other hand, is a function that starts and never ends. It contains an endless loop that implements the operation of the task and its triggering mechanism. As opposed to basic tasks, extended ones can have an initialisation part that is executed when the task starts, only, i.e., before entering the infinite loop. The triggering mechanism, including the specification of the task period if it is a periodic task, is part of the task code. In basic tasks, the triggering mechanism stays within the multitasking kernel and is defined using specific system calls invoked by the tasks creator, outside the tasks themselves.

Another important feature is the scheduling algorithm that an OS implements. In fact, the scheduling algorithm is crucial to satisfy the deadlines. This can be preemptive or non-preemptive. In a preemptive scheduler tasks can yield execution to other tasks according to some sort of priority, following the occurrence of events. In non-preemptive schedulers, tasks do not yield execution to other tasks, executing completely, instead, before switching over. A compromise is the so-called

cooperative model in which tasks can yield execution to other tasks, but by explicit invocation of a system call, only.

Most scheduling algorithms are priority based, meaning that there is a priority associated to each task. There are two types of scheduling: dynamic priority scheduling and static priority scheduling. In static priority scheduling the priorities of the tasks remain constant during the execution, but in dynamic priority scheduling they can change. The static priority scheduling algorithms focus on determining the priorities a priori to obtain the desired timing behaviour. Most of RTOS support static priority scheduling, only. Among the dynamic priorities scheduling algorithms, one of the most used ones is EDF, in which a task with an earlier deadline gets a a higher priority. A strong advantage of EDF is that it is commonly used because it allows scheduling with high CPU utilisation factors. The processor utilisation factor is the fraction of processor time spent in the execution of the tasks.

Another important aspect is if the scheduling of the tasks is determined before execution (static scheduling) or during the execution (dynamic scheduling). Even though the first case is more deterministic, according to Stankovic et al. in [33], for many real-time systems static scheduling is quite restrictive and inflexible.

A reliable inter-task communication mechanism is also determinant for the timing behaviour. Tasks must communicate in a way that they guarantee data integrity and synchronisation. Some of these mechanisms are mutexes and semaphores.

According to Silberschatz in [30], a scheduling challenge arises when a higher priority process needs to read or modify data that are currently being accessed by lower priority processes. Since data is typically protected with a lock, the higher-priority process will have to wait for a lower priority process to finish with the resource. This is called blocking. The situation becomes more complicated if the lower-priority process is preempted in favour of another process with a higher priority. When this happens, it is called priority inversion and there are some solutions to limit this effect. Between the most used ones there is the Priority Inheritance Protocol (PIP) and the Immediate Priority Ceiling (IPC). In PIP, when a task blocks one or more higher priority tasks, it temporarily inherits the highest priority of the blocked tasks, which prevents medium priority tasks from preempting it and prolonging the blocking duration of the higher priority ones. In IPC, the priority of a task that enters the critical section for an exclusive resource is set to the highest priority among the tasks sharing that resource.

2.3.1 Time Management

Usually, the OS executes the scheduler and the dispatcher periodically based on a timer tick interrupt [9]. When the timer expires, its interrupt service routine (ISR) has to check whether there are tasks to be activated, and if so, insert them in a ready task queue structure and if the task at the head of the queue has higher priority than the running task then, if preemption is allowed, switch these two tasks and dispatch the higher priority one.

There is another alternative, the so-called tickless scheduler, where the periodic timer interrupt is eliminated and replaced with a one-shot, interval timer. This alternative is explained by Barry in

[7] and Mitra in [20]. When an interrupt occurs, the scheduler checks whether an operation needs to be performed. Using an interval timer, the OS can anticipate when the next event will occur, program the interval time and wait for it to fire. When the interval time fires, then the scheduled activity is performed. Besides the obvious advantage of power saving when using tickless schedulers, comparing to a tick-based OS that may have to wake up from its power saving sleep state every few milliseconds, there is an important improvement of the resolution of the scheduling function. With a tick-based OS, the resolution for activating a task is determined by the period of the tick. On the other hand, when using a tickless scheduler, the resolution is the one of the timer itself, which is much higher.

2.3.2 Programming model

The authors Sabri et al. in [27] state that the programming model of an OS is very important for its performance, and it might also have a great impact on the ease of application development, as it defines the "model of abstraction" of the underlying system.

There are two different paradigms when it comes to the task models in an RTOS. The first is an event-driven model. In this case the program has a main loop in which it is waiting for events. These may be generated by the OS or by the application, and generally correspond to network and I/O notifications, timers, or other application-specific event. When an event arrives, if the task that services that event is available, it is called. When tasks end, they go back to a sleep state waiting for the following events. This model relies on asynchronous calls so that the tasks that serve the events do not block. In these circumstances, it is possible to use a single stack shared by all tasks in the system (see Section 3.3). Events are kept in an event queue until they can be serviced and then they are served sequentially, so the context switching is less complex and less time-consuming. More tasks allow handling more events, thus increasing the system throughput (served events) until the CPU is saturated. After that, the throughput will be constant, and the latency will naturally increase. Another advantage of event-driven systems is their ability to react to asynchronous external events which occur at instants that are not known in advance.

The other model is the multi-threaded model. In this model, every event immediately triggers its handling task, i.e., a thread, and they execute concurrently, scheduled by the OS. Note that there is no event queue. When switching from one task to another, the context of the previous one must be saved in its Process Control Block (PCB) and the new one must be restored. There is also the concern for the synchronisation operations protecting shared resources which may cause unbounded blocking and deadlocks. The possibility for a task to get blocked imposes the need for a separate stack per task. According to Welsh et al. in [41], this model is relatively easy to program, but its overhead can degrade the performance, specially when the number of threads is large: the throughput will decrease and the latency will increase, being prone to thrashing.

Two other popular models are related to the interface with the environment, the event-triggered and the time-triggered models. In the event-triggered case, tasks are directly triggered by asynchronous events, similarly to the multi-threaded model. The main aspect is that there is no control on the time between consecutive events, thus these systems are susceptible to event bursts that

cause load peaks, potential exhaustion of buffers and high jitter in the timing behaviour. Conversely, in the time-triggered systems, the tasks are initiated periodically at predetermined moments. It is much easier to predict the state of the system at a given point, but according to Scheler and Schroder-preikschat in [28] this is not necessarily a problem of event-triggered systems in real-time, because predictability is independent from determinism. However, when it comes to aperiodic and sporadic events it is very inefficient to use time-triggered system. It is necessary to poll these events with a relatively high frequency, and still the delay on serving such events is still quite large. Event-triggered systems would solve this problem, but dealing with the drawbacks explained above.

2.3.3 Memory Management

In embedded devices, one of the critical resources is physical memory, be it for program or data, volatile or non-volatile, because memory chips cost money and consume energy. Thus, OS memory footprint is a relevant feature to take into consideration, because it corresponds to memory that is consumed for system management only, thus unavailable to applications.

The desirable amount of memory for a given system depends on the application and its design. The basic configuration of an RTOS (including scheduling, interrupts, memory management) may even imply a small footprint, but when adding certain features, for instance Bluetooth communication stack, the footprint might increase significantly [18]. Thus, the choice between a full fledged RTOS that includes many features or a minimal one that has a specific application as target is not straightforward, because it causes a trade-off between richness of the feature set on one side, with higher abstraction and application support, and memory footprint on the other side.

The Internet Engineering Task Force (IETF) defines 3 classes of constrained devices, according to the available memory both in what comes to data size and code size. According to Hahm et al. in [14], Class 0 does not usually justify the use of an operating system. Class 1 and 2 have 10 kB of RAM or more, and 100 kB of Flash or more. The OS will have to operate on these typical values of memory capacity. In the particular case of the SoC nRF52840 that will be used later in our project, it has 1 MB Flash and 256 kB RAM. Thus, the memory footprint when choosing an appropriate OS is not a significant constraint, with many options that are suitable for these memory capacities.

Usually, the reference embedded system architecture follows the concept of memory hierarchy: very fast volatile cache memory, more of medium-speed main memory, a greater amount of cheap, nonvolatile magnetic or solid-state disk storage and, in some cases, external storage (Fig. 2.4). In the figure below it is clear that as the capacity increases, the performance gets worse. Cache memories are small but they are also fast. The goal of the OS is to abstract this hierarchy to facilitate its usage. This is the so-called memory management, that consists on keeping track of which parts of the memory are in use and allocate and deallocate memory to processes when needed in what comes to the main memory. The point is to exploit locality (nearness or re-use of accesses) to get the best out of both capacity and performance, allowing most accesses to use small but fast memory.

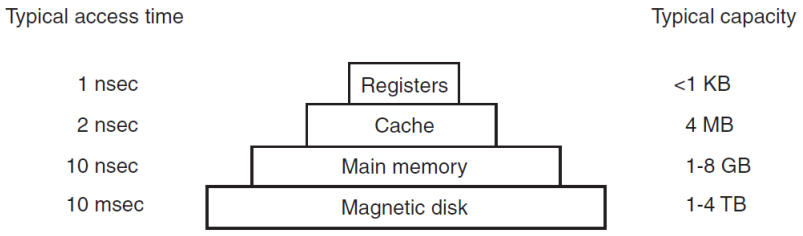

Figure 2.4: Memory hierarchy. Picture taken from [36].

Another important concept is the organisation of the system memory according to its usage in multiple specific areas. There is the program, stack and heap areas, as well as for constants, initialised data and non-initialised data (Fig 2.5). Memory can be virtualised and these areas can be present in each process.

Memory Area	Section Name	Section Type	Write Operation	Initial Value	Contents
Program area	.text	Code	Disabled	Yes	Stores machine codes.
Constant area	.rodata	Data	Disabled	Yes	Constant data. This section may not be produced, especially for host compilers (e.g. UNIX/PC)[4]
Initialized data area	.data	Data	Enabled	Yes	Initialized global and static data.
Non-initialized data area	.bss	Data	Enabled	No	Stores global data whose initial values are not specified (zero initialized). BSS - "Block Started by Symbol"
Stack area	—	—	Enabled	No	Required for program execution. Dynamic Area Allocation.
Heap area	—	—	Enabled	No	Used by a library function (malloc, realloc, calloc, and new). Dynamic Area Allocation.

Figure 2.5: System memory areas, possibly virtualised by each process.

In complex systems, a process has its own address space, independent of those belonging to other processes. However, the total amount of RAM needed by all the processes is often much more than the amount of available memory, so not all address spaces can be simultaneously in the main system memory, as it can be seen in Fig. 2.6. The simplest strategy, called "swapping", consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running. The other strategy, called virtual memory, allows programs to run even when they are only partially in main memory. When using virtual memory, each program has its own address space within memory pages. Each page is a contiguous range of addresses. These pages are mapped onto physical memory, but not all pages have to be in physical memory at the same time to run the program. The virtual addresses go to an MMU (Memory Management Unit) that maps the virtual addresses onto the physical memory addresses.

As it is explained by Tanenbaum and Boschung in [36], another important point is how much

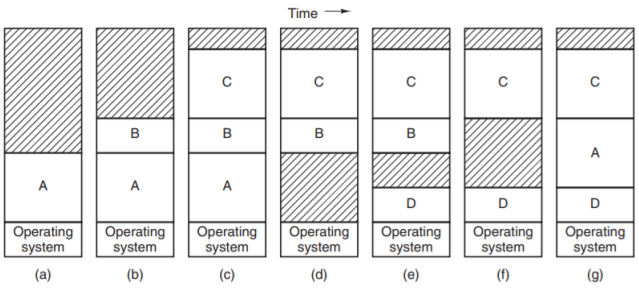

Figure 2.6: Allocation and deallocation of memory for processes. Picture taken from [36].

memory should be allocated for a process when it is created or swapped in. If processes are created with a fixed size that never changes, then the allocation is simple - this is called static allocation. This makes the execution very fast and efficient. The other possibility is that data segments of processes can grow by dynamically allocating memory from a heap. When memory is assigned dynamically, the OS must manage it. To do so, there are four types of dynamic memory allocation/deallocation algorithms: sequential fit, buddy allocator system, indexed fit, bitmapped fit, as explained by Shah and Shah in [29]. This allows to have more flexibility, but the drawback is a non deterministic behaviour: there is no guarantee of an efficient memory usage and there might be fragmentation. The dynamic memory allocation also requires the implementation of functions for freeing up the memory no longer in use. According to D. and M. in [9], RTOS specially designed for small embedded system should have very simple memory management policies, because the memory space is relatively small. In any case, the best option would be an OS that supports both types of allocation in a configurable way. Small embedded systems never use virtual memory for its overhead. Likewise, real-time systems seldom use dynamic memory for the unpredictability that it implies. High-end devices, on the other hand, may use both.

Finally, the stack management is also very important, potentially requirement a significant amount of memory. In conventional OS, each process must have its own private stack space. This is used to store its context (i.e. the CPU registers), its local variables and the return addresses and local variables of all invoked functions. However, if the number of tasks is large, a large amount of memory might be required for all stacks. A solution for this is the so-called stack sharing: tasks can share a single stack space. This is only possible when these tasks are independent, i.e. when operations interleave in a consistent direction, for example, according to a fixed priority order, only. If a task is suspended while a lower priority one executes, such order is violated and a single stack is inadequate. For example, if task A pushes a certain variable, then task B another, then task A pops the first value from the stack, then it will pop the value stored by task B, and this is an erroneous behaviour. In small systems, a single stack should be used whenever possible, given its superior memory usage efficiency. In more resourceful systems, a multi-stack approach is more adequate since it offers higher programming flexibility.

2.3.4 Development Tools

The development tools provided by an OS are also important. These include simulation, testing and debugging tools. For example, the capability of debugging not only allows a faster development, but it improves the reliability of the code. This decreases the cost and the effort applied in the development and allows a better estimation of the performance.

In what comes to debugging tools, tool chains such as gcc usually include corresponding debugging tools, and they are widely spread and commonly used. There is a large variety of used tools, and some might be as simple as the use of a serial interface for a simple debug.

As to simulation and testing tools, they are important because of the distributed nature of the IoT systems. A widely used approach to deal with the hardware-related part of the testing, such as the testing of device drivers, is to use hardware emulation tools. Network emulators and simulators such as Cooja (Contiki) allow for the integration of OS code. According to Scheler and Schroder-preikschat in [28], nowadays, simulation is much limited to networking.

Currently, most OS include IDEs that not only allow developing the code, but also to build, cross-compile, load, debug and run.

The SoC nR5284 includes some mechanisms for debug and trace. It implements serial wire debug (SWD) interface. It also supports real-time debugging, which allows interrupts to execute to completion in real time when breakpoints are set in thread mode or lower priority interrupts. This enables developers to set breakpoints and single-step through the code without the risk of failing the handling of real-time event-driven threads that run at higher priority. For example, this enables the device to continue to service the high-priority interrupts of an external controller or sensor without failure or loss of state synchronisation while the developer steps through code in a low-priority thread. For trace, it includes a Trace Port Interface (TPI).

2.3.5 Security

Another key aspect is the security level provided. Due to a growing potential for attacks of malicious software on systems and networks, this issue is more and more relevant. IoT security challenges include data integrity, authentication, and access control which might be heterogeneous across the system. With this concern, it also comes the need for maintainability, since the protocols and execution platforms should be constantly reviewed and updated.

According to Belleza and De Freitas in [8], for commercial applications, open-source OSs can be an advantage, because for IoT devices privacy and security are fundamental and open-source software can address easily these two characteristics due to the collaborative way the development takes place.

The main security measures include permission or access authorisation, authentication usage, cryptographic usage, and subsystem specific usage. One of the measures of an OS might be support authenticated, authorised, and encrypted data transfers and analyse all incoming and outgoing network traffic through a firewall. Another important measure is including thread separation, stack and memory protection.

The IEEE 802.15.4 standard provides security mechanisms at the MAC layer, but nowadays there is no standard protocol or suite of standard protocols that ensures the end-to-end security at network and/or higher layers. Some of the most important security mechanisms implemented by the RTOS are:

- Identification and authorisation of users
- Authorisation and privilege levels
- RTOS network firewall
- Usage of secure communication channels such as IPSec, SSH, SSL or VPN access
- Support the use of official encryption certificates (e.g. SSL certificates for device terminal connections via SSH)
- Memory protection via the memory management unit (MMU) and isolation from the kernel-mode operation provides a layer of protection against malicious code
- Secure Boot Loading allows to verify authenticity of boot loader and also secure mechanism in place to prevent and detect tampering of boot image
- Secure Data Storage. RTOS system should have capabilities to store data securely using techniques such as encryption and partitioning.

2.3.6 Run-time performance

The run-time performance is naturally a key aspect when talking about RTOS.

The run-time performance usually depends mostly on the interrupt service latency and the time taken for context switching. Scheduling algorithms have a great impact on the performance of an OS, as they determine the frequency of context switching and the level of mutual interference.

This aspect might be determinant when choosing an RTOS for a specific application and it will be approached with some detail later in the comparative experimental tests using suitable benchmarks.

2.3.7 Hardware features

It is important to state that choosing an OS suitable for a specific platform is not trivial: when it comes to the IoT, there is a large variety of processing hardware and communication technologies, and thus it is hard to deal with such heterogeneity. As said before, the platform Kallisto is based on the SoC nRF52840 [22]. The processor is a 32-bit ARM Cortex-M4. The solutions of OS to consider in this work have to support the hardware and protocols that the SoC nRF5280 uses (Fig. 2.7).

This target platform includes a vast set of specific hardware devices, namely timers, GPIO, PWM, ADC, RTC (Real-time clock), I^2C, SPI, UART, USB, flash memory interface, SPI, QSPI

(w/ external flash), audio peripherals (I2S and PDM (digital microphone interface)) and DMA. To facilitate application development, it is important that the OS provides abstract interfaces to these devices that hide the associated complexity within the so-called device-drivers.

In what comes to communication, a particular technology that is offered by the platform and must be supported by the OS is Bluetooth Low Energy (BLE) 5.0. It is a more power efficient and less costly version of Bluetooth that still maintains the same communication range as the original version. The key difference is that BLE does not continuously stream data, but remains in sleep-mode until there is something to send. BLE is common for portable, battery-driven devices that only send data periodically with low duty-cycle.

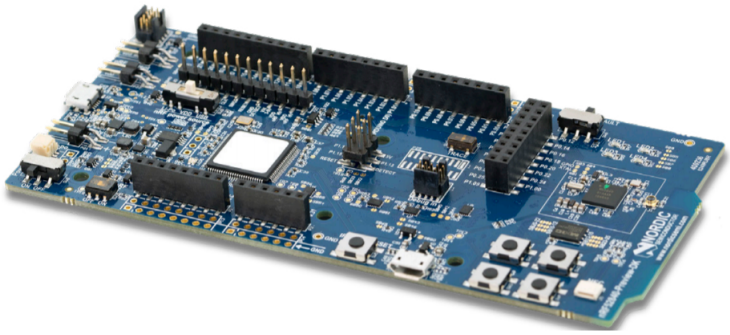

Figure 2.7: nRF52840 Development Kit. Picture taken from the nRF52840 DK documentation.

2.3.8 POSIX

Portable Operating System Interface for Computer Environments (POSIX) is an IEEE standard that defines a set of rules and services to provide a common base for RTOS. According to Harbour in [16], the main goal of defining a standard is the portability: an application can be easily ported across different OS if both are POSIX compliant. If an OS is POSIX compliant, it means that it provides standard interfaces for the system calls and services that the OS provides. It defines the APIs between the OS and the application, and the syntax and semantics of the interfaces, in what comes to the data types and function prototypes. The implementation of those services depends on the OS, it is not in the scope of the standard to define it.

POSIX has a set of subsets, called profiles, that define other services that compliant OS must have to ensure determinism in the timing behaviour, i.e., a standard suitable for small RTOS for embedded systems. POSIX.13 defines four profiles for different sizes of embedded RTOS. However, the authors of [16] argue that even these profiles for the smallest OS are quite complex for the simplest platforms (e.g. 8-bit platforms). For these there are other more appropriate standards, such as OSEK/VDX. POSIX is suitable for platforms that can hold a kernel with a footprint above 10 kB.

Among the main services defined in the POSIX standard for real-time, the following are probably the most important ones [23].

1. Concurrency between processes that include threads

2. Scheduling Algorithms:
   - FIFO for processes/threads with the same priority
   - Round-Robin
   - Sporadic Server
   These are typically used within fixed priority algorithms. They can be applied at a thread level or a process level according to the scope:
   - System contention scope: all threads in the system compete with the others
   - Process contention scope: schedules first the process, and then the thread in that process
   - Mixed contention scope: some threads have a "system" scope, and others a "process" scope.

3. Synchronisation

   POSIX defines a type of variable for implementing mutex, as well as the associated functions for its use. It also defines two mechanisms to limit priority inversion: immediate priority ceiling and priority inheritance. It also defines semaphores and condition variables, with the respective types of variables and functions. Message queues are also included, with support for both polling or waiting for new messages from other processes or threads.

4. Timing Services

   POSIX implements multiple clocks, namely real-time clock, monotonic clock and boottime clock, together with the services for the synchronising with them.

5. Memory Management

   POSIX includes shared memory (memory regions shared between multiple processes) and memory locking (functions to prevent virtual memory swapping of physical memory pages)

According to Obenland in [23], the advantages of using standards when developing software go beyond the portability of the software. Industry adopted standards have other advantages: usually, the developed software might be used with other commercially available software, and then the life-cycle of a software application requires many updates and changes. Using standards improves interoperability and portability, making it all easier.

However, using a standard might not always be the right choice. The standard might not provide the functionalities needed for the application, it might decrease its performance or it might demand too much resources. All of this must be considered in the context of the application: if it is a simple application and the portability is not relevant, then the overhead does not justify the compliance with the standard.

2.3.9 Other characteristics

Besides the technical aspects, there are other intangible aspects that are also important. One is the maturity of the OS. However, it is very difficult to measure. In this work, we assess maturity by means of three factors: the date of the first release, the date of the last release and their update rate in time. The size and activity level of the community supporting the OS is also an important factor, and also an indicator of the OS maturity.

Another factor is the diversity of programming languages the OS supports, but this was not taken into account, because in this case the only language of interest for the development of the application is C and it is a language usually supported by most embedded OS.

2.4 Benchmarking RTOS

This section presents some benchmarks used for the comparison of runtime performance among RTOS, to complement the comparison based on static features that we discussed in the previous subsection.

The authors Weiderman and Kamenoff in [40] define a series of requirements and associated experiments called the Hartstone Uniprocessor Benchmark (HUB) to test if a uniprocessor system is able to handle certain types of real-time applications. This is one of the first works to focus on this matter, and it is an important reference for others. For instance, in [12], Golatowski and Timmermann use the HUB for educational purposes, when teaching fixed priority scheduling analysis. The authors Pinto et al. in [25] propose another benchmark, which is an adaptation of Hartstone, but adapted to the modern robotics context and its applications. The HUB focuses more on the evaluation of the scheduling algorithm. It includes five sets of experiments with tasks differing in their activation patterns:

1. PH (Periodic Tasks, Harmonic Frequencies)
2. PN (Periodic Tasks, Nonharmonic Frequencies)
3. AH (Periodic Tasks, Aperiodic Processing)
4. SH (Periodic Tasks with Synchronization)
5. SA (Periodic Tasks with Aperiodic Processing and Synchronization)

Each task is characterised by its execution time and its deadline. There are 4 parameters that can be changed: the number of tasks, execution time of tasks, blocking time of tasks and deadlines. One experiment consists on changing one of this parameters until the system no longer works correctly, i.e., a deadline is missed. These tests are relevant specially when the system is under a large load (many tasks to perform within a certain time). This might be relevant for our project, for instance in an operation mode with low duty-cycle, in which it must perform many operations within a relatively short time.

Other works focus not so much on the scheduling itself, but on the architecture of the OS. In [34], Stewart presents some techniques to measure the execution time of both user code and the operating system overhead. It starts by distinguishing software-oriented tests, which provide measurements with millisecond resolution, and fine-grain techniques that use specific hardware, such as logic analysers and oscilloscopes, and provide microsecond resolution. It also describes the main contributions for the operating system overhead: the overhead of context switching from one task to another and the overhead incurred due to being interrupted by an interrupt handler. It suggests to create some tasks that merely toggle bits on a digital output port, which will be analysed with the logic analyser. A good approximation of the context switching overhead can be calculated as shown in Eq. 2.1, which parameters are explained in Figure 2.8. To measure the overhead of the interrupt handler, a bit must be toggled in the beginning of the ISR and again in the end. The duration of that pulse is the overhead of the interrupt handler.

$$\Delta_{switching} = \frac{t_1 - t_2 - t_3}{2} \tag{2.1}$$

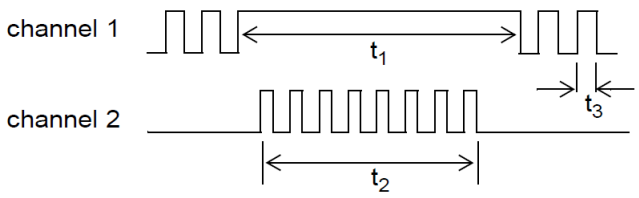

Figure 2.8: Measuring the context switch overhead. Picture taken from [34]

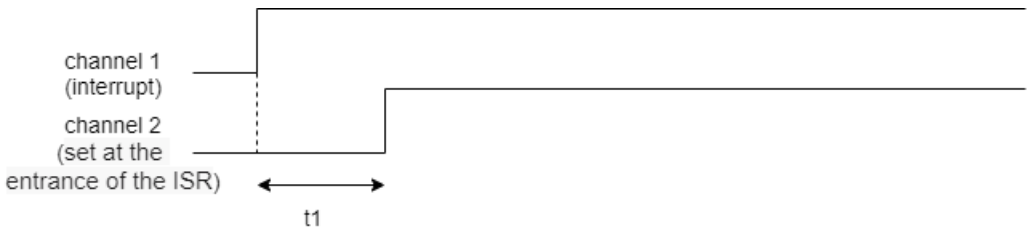

Figure 2.9: Latency of the interrupt handler.

To measure the latency of the interrupt as in Figure 2.9, the hardware interrupt request line should be connected to one of the logic analyser channels. A signal should be toggled in the beginning of the interrupt routine. The difference in time between when the IRQ line becomes active and the pulse of the interrupt handler is seen is the interrupt latency.

The authors Hambarde et al. in [15] define five main metrics of RTOS performance, namely:

1. Interrupt Service Latency

   This latency consists in the time difference between the moment that an interrupt is generated and the moment that the associated interrupt handler starts, marked by setting an output pin. This is analysed externally, either with a signal generator and an oscilloscope or logic

analyser, or alternatively with a timer, taking the RTOS in conjunction with the hardware as a black box. This corresponds to the interrupt handler latency mentioned in [34] and described above. This measurement can be made in 2 different scenarios: with a small number of tasks (possibly without active tasks) and with a significant number of short tasks.

Note that this is different from measuring the overhead of an interrupt handler. That would measure the time the interrupt handler routine takes to execute, which might be interesting when focusing on the overhead of the operating system in a particular application.

2. Jitter of the Interrupt Service Latency

   The jitter is obtained from several latency measurements, by calculating the maximum time difference between two consecutive interrupt latency measurements. Thus, the jitter measures the maximum uncertainty in a specific measurement, in this case the latency, and it is very important in RTOS, as it degrades predictability.

3. Maximum Reaction Rate

   Maximum Reaction Rate is obtained using the method proposed by ISA (International Society for Measurement and Control), according to Aroca and Caurin in [3]. It is the maximum interrupt frequency that can be handled by the RTOS with reliability. The inverse of that frequency is the worst case response time.

   This test consists in setting a system that copies an input signal to an output port. The pulses generated in the input are counted and compared to the ones counted in the output. If they are the same, then the system is stable. The frequency must be incremented until the number of pulses diverge. The maximum stable operation, i.e., before the pulse counts diverge, allows computing the maximum reaction rate.

   Again, this test can be performed in the two load scenarios described before.

4. Interrupt Blocking Time

   Interrupt Blocking Time is defined as the sum of interrupt blocking time during which the kernel is pending to respond to an interrupt. This includes saving the tasks context, determining the interrupt source and invoking the interrupt handler. This feature is very important, because most embedded systems are interrupt-driven, and thus this latency might have a big impact on the system throughput.

   The Interrupt Service Latency is a different measurement, because it does not include other interrupts that might happen. It is the latency on serving the interrupt without any blocking. On the other hand, the Interrupt Blocking Time includes the impact of the OS that might disable interrupts for some time. They are measured in the same way, but in different conditions of interrupts.

5. Inter-Process Communication

Most RTOS include synchronisation and inter-task communication services. These include mutexes, semaphores, message queues, events, mailboxes. The latency of these services is important because it can be reasonably long, due to the complexity of the primitives.

Other works focus more on specific metrics, e.g. the work in [3] focuses more on the interrupt service latency, its jitter and the maximum reaction rate.

Mumolo et al. in [21] specify the following parameters as good indicators of the performance of the kernel.

1. Context switch time
   It is the time for the execution of the context switch routine of the scheduler.

2. Jitter time
   It is the delay between the instant in which a periodic process is activated and the instant in which it actually starts. This corresponds to the time in which the task is ready, but not yet running. Note that what we refer to as "jitter" does not measure the uncertainty in a specific measurement, but the uncertainty in the moment in which the task will start running.

3. Deferred interrupt latency
   It is the delay between an interrupt event and the execution of its serving task. This applies to RTOS that implement a deferred mechanism for interrupt serving, where the task for serving the interrupt is scheduled along with the other tasks, and not executed immediately in the ISR.

4. Kernel overhead
   The time spent for the kernel overall.

In [39], Ungurean base their RTOS selection on the latency in handling critical operations triggered by both internal and external events. They focus on the comparison of task context switching. An output GPIO is used to signal the beginning and the end of the operation to be measured. The scenarios presented below show some experiments for when the event is used as a synchronisation mechanism between two tasks. However, the same experiments can be performed for semaphores or mailboxes.

1. Scenario 1 - signal an event causing a context switch (Fig. 2.10)

   This allows measuring the time for the context switching triggered by the lower priority task when sending a signal event to a task that is blocked on the wait primitive.

2. Scenario 2 - wait for an event causing a context switch (Fig. 2.11)

   This allows measuring the time for the context switching from a higher priority task to a lower priority task. When calling the wait primitive to wait for an event, a context switch is triggered and the task with lower priority will run.

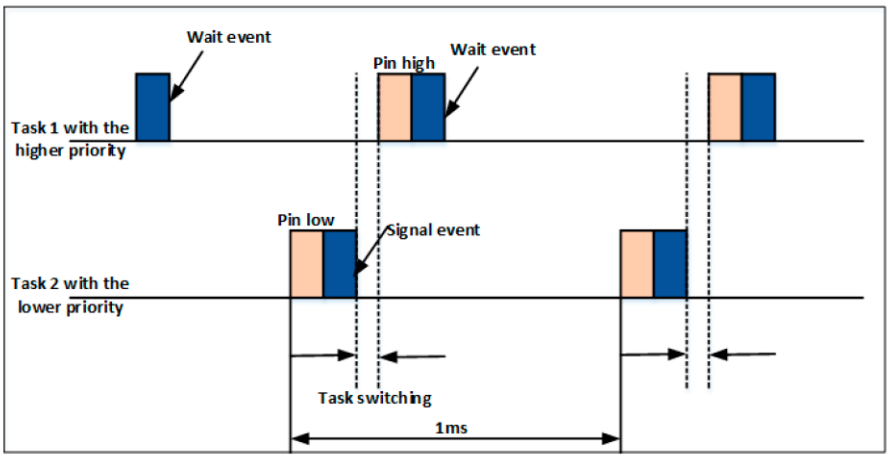

Figure 2.10: Description of scenario 1 to measure interrupt latency with an event causing a context switch. Task 1 is waiting for an event. In the meanwhile, Task 2 starts running and it will signal the event. This will trigger Task 1 to start running again, allowing us to measure the time for the signal primitive when it causes context switch.

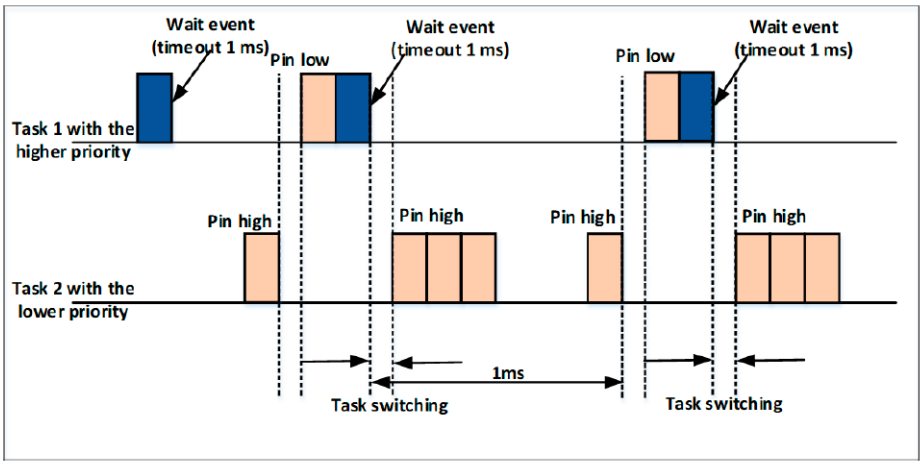

Figure 2.11: Description of scenario 2 to measure context switching from a higher priority task to a lower priority task. Task 2 is running, but it is preempted by the Task 1. Task 1 eventually stops, waiting for an event, and therefore allowing Task 2 to run again, allowing us to measure the time for the wait primitive when it causes context switch.

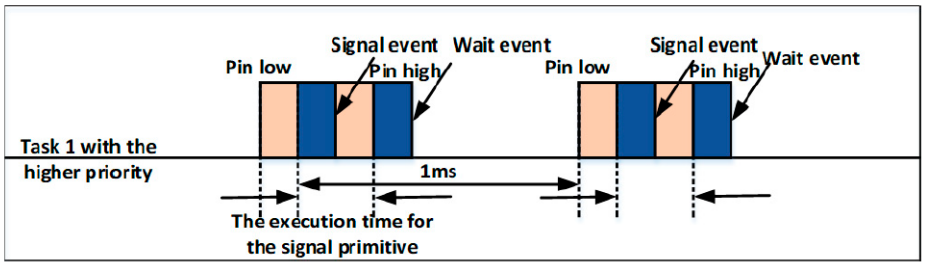

Figure 2.12: Description of scenario 3 to measure the execution time for the signal primitive for event triggering. The task is running and it signals an event, and after that it continues running. This allows to measure the time for the signal primitive without context switch.

3. Scenario 3 - signal an event without context switching (Fig. 2.12)

   This allows measuring the execution time for the signal primitive for event triggering.

4. Scenario 4 - wait for an available event (Fig. 2.13)

   This allows measuring the execution time for the wait primitive used to receive an existing event.

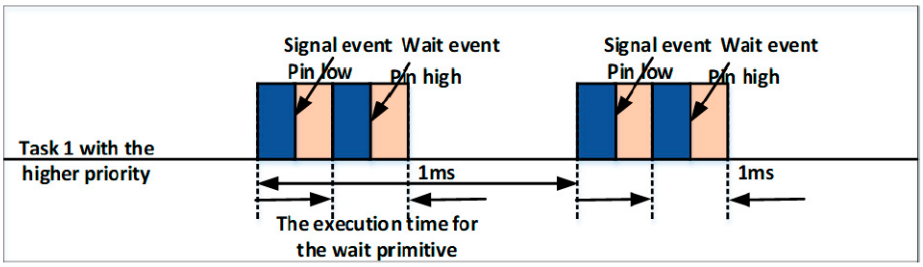

Figure 2.13: Description of scenario 4 to measure the execution time for the wait primitive for event receiving. The task is running and it waits for an event that was already signalled, so after that it continues running. This allows to measure the time for the wait primitive without context switch.

Finally, some works go beyond general task scheduling and kernel events and measure RTOS features in specific application contexts. One example that is particularly relevant to our work is Wireless Sensor Networks (WSN). The RTOS of a sensor node must have the capability of responding to events in a deterministic way. The latency in answering to an event and the throughput of serving events will characterise the performance of the OS.

Duffy et al. in [10] define a network model in which sensor nodes must not only perform the sensing task but also a packet processing task, as they must receive and pass packets from and to other nodes. This last task has higher priority. The point is to measure the average task execution time of the packet forwarding task and its jitter. For event-driven RTOS, the execution time starts in the beginning of the interrupt routine and it finishes when the packet processing task

is completed and removed from the ready queue. For multi-threaded applications, it starts in the same instant and it finishes when the packet processing thread finishes and is sent to the wait state.

Other measurements focus more on the network. The throughput is the actual rate that information is transferred, latency is the delay between the sender and the receiver decoding it. The jitter, i.e., the variation in packet delay at the receiver, is also interesting.

The delay can be measured using the Round Trip Delay (RTD) technique between two nodes in the network. This allows measuring the overhead of the protocol stack. The round-trip delay is calculated by one of the nodes with the information of the timestamps taken by both nodes, i.e. using the software timer. The timestamps must be extracted just before sending the packet and right after receiving it, thus including the whole round trip process. The network delay can also be measured with an oscilloscope, measuring the time difference between the transmission and the reception.

Fig. 2.14 explains better the components of the RTD. According to Silva et al. in [32], the first TX is the time for the RTOS to prepare a packet until it is sent in the physical layer. Then there is the Communication, which is the transmission time for the actual physical communication. The Rx is the reception time, i.e. the delay since the packets arrives at the physical layer of the receiver until it is delivered to the application at the receiver. Overall, the time spent in the protocol stack is the sum of Tx and Rx. If we know the transmission rate of the physical connection and the frame size, and the nodes are directly connected to one another without any transmission interference, then it is also possible to calculate the time spent executing the protocol stack.

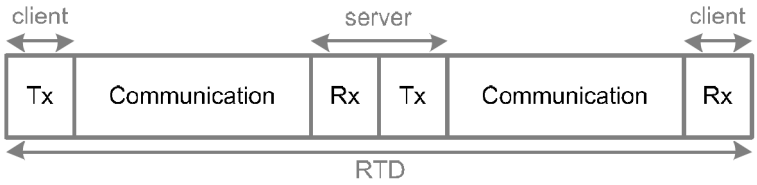

Figure 2.14: Round Trip Delay. Picture taken from [32]

Another interesting experience is to send packets from one node to another one with an increasing rate and to see the maximum throughput of packets that can be sent and received without any loss.

2.5 Summary

This chapter reviewed some basic features of operating systems targeted to embedded platforms for real-time applications. We saw the main features of an embedded OS and an RTOS. Then we discussed the most relevant features that such an OS must offer, and which will be used to guide the desired OS comparison and selection that will follow in the next chapters.

We also presented some general benchmarks proposed in the read literature for evaluating the performance of OS. Some relevant metrics were explained, together with the respective experimental tests.

Chapter 3

Defining a short list of RTOS for Kallisto

The first step for choosing the set of candidate operating systems for the Kallisto platform was to gather an extensive list of options. For this, we searched the Wikipedia[1] with the following key: "Comparison of real-time operating systems". This lead to an initial set of 192 options.

Secondly, we constrained our search to RTOS that were active (not closed or defunct) and that supported the target micro-controller and base board. These criteria allowed eliminating most of the initial options, leading to a remaining set of 14 RTOS, namely:

- Apache Mynewt
- Contiki-NG
- FreeRTOS
- RIOT
- Zephyr OS
- Huawei LiteOS
- Azure RTOS
- embOS
- LynxOS
- MicroC/OS (µC/OS)
- NuttX
- PikeOS

[1] https://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems

- RT_Thread
- TNKernel

Next, we added constraints based on the hardware criteria, as explained in Chapter 2. There are not many requirements in what comes to the hardware, apart from some features that are commonly present in most RTOS. However, the support for BLE is not a standard feature of an RTOS. Thus, the following RTOS were removed from the list because they do not have support for BLE: Azure RTOS, embOS, LynxOS, MicroC/OS (µC/OS), NuttX, PikeOS, RT_Thread, TNKernel. In what comes to Huawei LiteOS, this RTOS supports a just a short list of devices and it lacks wide support in terms of forums and supporting community. This led to a final selection of 5 RTOS, namely:

1. Apache Mynewt
2. RIOT
3. Contiki-NG
4. Zephyr
5. FreeRTOS

In the remainder of this chapter, we will briefly describe these RTOS focusing on the following aspects:

- Toolchain and IDE support
- Maturity
- Dynamic memory
- Scheduling
- Synchronisation and inter-task communication
- IO
- Networking
- Storage and display
- Shell access
- Memory protection

3.1 Apache Mynewt

Apache Mynewt[2] is an open-source RTOS that focuses on providing an easy development of applications for embedded systems where power and cost are driving factors, more specifically constrained low-end IoT devices. The RTOS provides a rich set of libraries across hardware and program tasks for consistent management and monitoring. Also available is network protocol software starting with a BLE compliant stack. Another important aspect is the unified tools framework for developers to compose and port the software easily to their devices, and enable system services for remote management. Its software architecture is shown in Fig. 3.1.

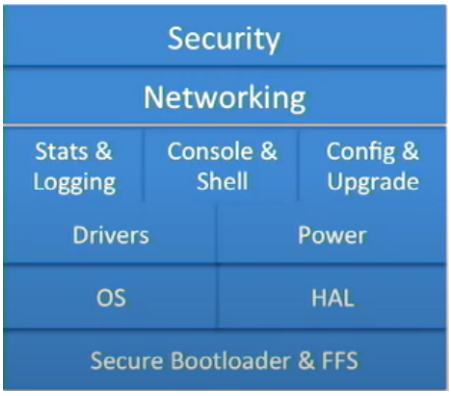

Figure 3.1: The architecture of the Apache Mynewt software. Picture taken from LAS16-104: MyNewt technical overview.

1. Toolchain and IDE Suport:
   Newt is a smart build and package management system for embedded contexts. It also includes command line package management and build system (Newt Tool)

2. Maturity:

   - First release: March 2016
   - Latest release: April 2020
   - Update rate: 7 months

3. Memory footprint:

   - Data memory size ranges from 8 to 16kB.
   - Program size ranges from 64 to 128kB.

4. Dynamic memory:
   It includes an API for dynamic memory allocatio and allows memory blocks, called memory pools, to be dynamically allocated from a designated memory region.

[2]https://mynewt.apache.org/

5. Scheduling:
   Fixed priorities, preemptive kernel. Priorities are defined from 0 (highest priority) to 255 (lowest priority).

6. Synchronisation and Inter-task communication

   - Mutex
     These include the PIP (Priority Inheritance Protocol).
   - Semaphore
   - Event Queues

7. IO

   - Timer
   - GPIO
   - PWM
   - ADC
   - RTC (Real-time counter)
   - I2C
   - SPI
   - UART
   - Flash memory interface

8. Networking

   - 802.15.4
   - BLE 5.0 with NimBLE
   - Bluetooth Mesh
   - Wi-Fi
   - LoRaWAN
   - LoRa PHY
   - CoAP
   - 6LoWPAN

9. Storage & Display:
   Support for multiple file systems, such as Newtron Flash Filesystem and FAT. Others can be added.

10. Shell Access:
    Does not refer.

11. Memory Protection:
    Does not have a memory protection unit.

12. POSIX Compliant:
    No.

3.2 Contiki-NG

Contiki-NG[3] is an operating system for IoT resource-constrained devices in the Internet of Things. In 2017, Contiki-NG started as a fork of the Contiki operating system, with the following goals: focus on dependable (reliable and secure) standard-based IPv6 communication, focus on modern IoT platforms (e.g. ARM Cortex M3 and other 32-bit MCUs) and modernize the structure, configuration, logging and platforms to reflect these goals. It is a lightweight OS written in C, with low-memory capabilities and an event driven architecture. The communication model uses messages that are passed through multiple events. The source code is available as open source with a BSD license. The software architecture is shown in Fig. 3.2.

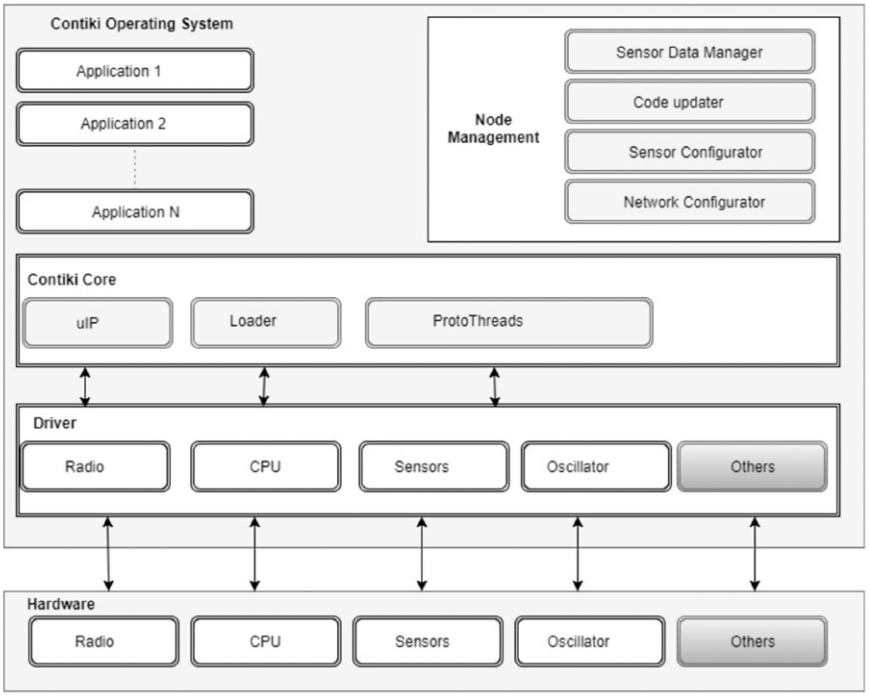

Figure 3.2: The architecture of the Contiki-NG software. Picture taken from [26].

1. Toolchain and IDE Suport:
   The Contiki-NG Docker image is recommended for easy setup of a consistent development environment. Alternatively, the toolchains can be natively installed on the system. Contiki also conveniently supports hardware used on several open testbeds such as IoT-LAB.

[3] https://www.contiki-ng.org/

2. Maturity:

   - First release: November 2017
   - Latest release: May 2020
   - Update rate: 6 months

   The source code is maintained by a merge team that reviews the contributions from the community. A number of source code forks exist, in which new features are developed by independent teams (also because Contiki has a big relevance in the academic context), or in which companies maintain their own versions of Contiki, possibly with support for their own hardware.

3. Memory footprint:

   - Data memory size: > 10 kB
   - Program size on the order of a 100 kB

4. Dynamic memory:
   Contiki-NG supports both static and dynamic memory allocations, but it is designed primarily for static allocation. It has some libraries for memory management.

5. Scheduling:

   The Contiki-NG model is based on an event-driven cooperative scheduling approach. It uses two types of events: Asynchronous and Synchronous.

   Asynchronous events are put in a queue and get dispatched to the receiving processes in a round-robin fashion. An asynchronous event can be posted to a specific process, or it can be broadcast. In the latter case, the kernel will dispatch the event to all processes. There is no way to specify the order in which processes will receive a broadcast event.

   Synchronous events cause the receiving process to get scheduled immediately. This will normally cause the posting process execution to stop until the receiving process has finished consuming the event, at which point the posting process resumes. Synchronous events cannot be broadcast.

   Contiki also provides a pseudo mechanism for multithreading using protothread.

6. Synchronisation and inter-task communication

   - Mutex
   - Global Interrupt Manipulation
     Allows to implement platform/CPU-independent functions for entering / exiting critical code sections.

7. IO

- Timer
- GPIO
- RTCs (Real-time counter)
- SPI
- USB
- Flash memory interface
- SPI based MMC/SDcard interface
- QSPI (w/ external flash)

8. Networking
   - 802.15.4
   - BLE 5.0
   - CoAP
   - 6LoWPAN
   - RPL
   - MQTT
9. Storage & Display:
   Coffee File System
10. Shell Access:
    Yes.
11. Memory Protection:
    Does not have a memory protection unit.
12. POSIX Compliant:
    No.

3.3 FreeRTOS

FreeRTOS[4] is one of the market-leading RTOS for microcontrollers and small microprocessors. It is distributed freely under the MIT open source license. It is widely spreaded, thus it is supported by a large community and it includes several equally popular forks, such as SafeRTOS and OpenRTOS. FreeRTOS is designed to be small, simple, portable, and easy to use.

It is based on a preemptive microkernel. The fact of begin based on a microkernel architecture provides not only more flexibility in the development of the application, but specially robustness

[4] https://freertos.org/RTOS.html

(since a crashing device driver will not affect the stability of the whole system). The software architecture is shown in Fig. 3.3.

The way of thinking is a bit different then the other RTOS in this section. In fact, the only provided functionalities are scheduling, mutexes, semaphores, and software timers. It does not provide any networking capabilities and neither defines a driver model or MCU peripheral abstraction. For testing and debugging the system also depends on third-party solutions. These components are available within many additional tools and libraries that are part of the FreeRTOS ecosystem.

However, it is frequently used along with the devices by Nordic Semiconductors, such as the SoC in which Kallisto is based, called nRF52. Nordic's SDK, in which the current implementation of Kallisto applications is based, provides an implementation of a FreeRTOS port to nRF52. The port is also fully compatible with all SoftDevice implementations, which are the BLE stacks from Nordic. Thus, we decided that it would be very interesting to explore.

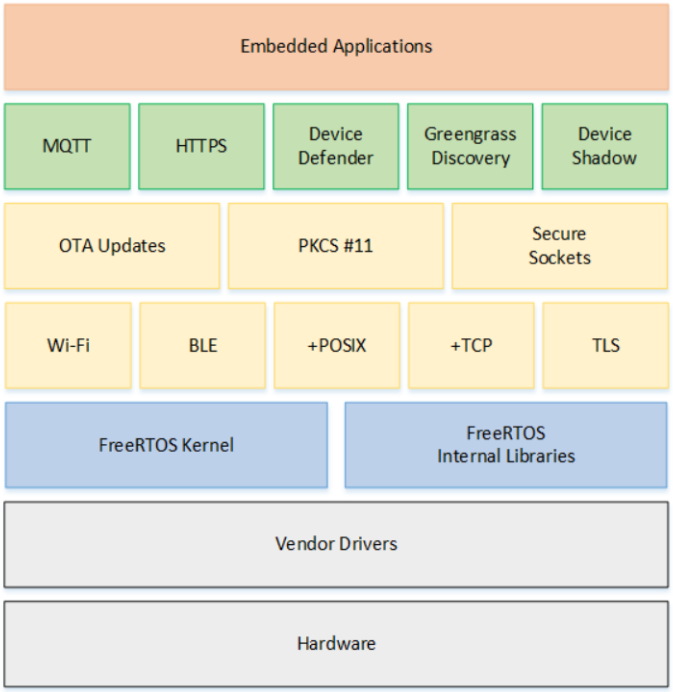

Figure 3.3: The architecture of the FreeRTOS software. Picture taken from [1]

1. Toolchain and IDE Suport:
   FreeRTOS applications can be developed and tested using both Visual Studio and the Eclipse IDE, without the need of copying all the RTOS source files to the project directory.
   FreeRTOS can also be used along with Nordic's SDK and its toolchain, using for example SEGGER Embedded Studio.

2. Maturity:

   - First release: December 2003

- Latest release: December 2020
- Update rate: more or less 2 months

3. Memory footprint:
   - Data memory size < 1kB
   - RTOS kernel binary image from 6 kB to 12 kB.
4. Dynamic memory:
   FreeRTOS provides both static and dynamic allocation. FreeRTOS can allocate the memory used by the RTOS objects from the special FreeRTOS heap. This RTOS includes a memory several heap management schemes that range in complexity and features and include 5 different memory allocation algorithms.

   Another option is to give the developer the ability to provide the memory themselves, statically, allowing objects to be created without any memory allocated dynamically.
5. Scheduling:

   FreeRTOS is based on a preemptive priority-based scheduler, where to each task is assigned a priority from 0 to a maximum, where the maximum is defined within FreeRTOSConfig.h.

   Any number of tasks can share the same priority, and then tasks of equal priority that are ready will share the available processing time using a time sliced round robin scheduling scheme.

   It includes a tickless scheduler.
6. Synchronisation and Inter-task communication
   - Mutex
     Mutexes include PIP (Priority Inheritance Protocol) to deal with priority inversion.
   - Semaphore
   - Queues
   - Direct to Task Notifications
     These can implement semaphores, events, mailboxes in a lightweight manner.
   - Stream and Message Buffers
7. IO and Networking

   FreeRTOS provides a collection of MIT licensed libraries available for use in resource-constrained devices. However, these libraries must be included latter. Generally, when talking about FreeRTOS, we are referring to the Kernel, because by default FreeRTOS does not include any drivers nor any MCU peripheral abbstraction.

   FreeRTOS libraries are tested and optimised to be used with the FreeRTOS kernel. There are three downloads that offer different services:

- The libraries in the FreeRTOS+ download provide connectivity and utility functionality suitable for building smart microcontroller-based devices and connecting IoT devices to the cloud.
- The AWS IoT libraries provide connectivity and security libraries functionality for building smart microcontroller-based devices and connecting to the cloud.
- The libraries in the FreeRTOS Labs download directory are fully functional, but undergoing optimisations or refactoring to improve memory usage, modularity, documentation, demo usability, or test coverage.

8. Storage & Display:
   FAT file system.

9. Shell Access:
   User friendly and full-featured Shell interface.

10. Memory Protection:
    FreeRTOS provides official Memory Protection Unit (MPU) support on ARMv7-M.

    FreeRTOS MPU ports enable MCU applications to be more robust and more secure by restricting access to resources such as RAM, executable code, peripherals, and memory beyond the limit of a task's stack. The user can define the memory regions that can be accessed by a task or a group of tasks. For example, one can prevent code from ever executing from RAM as doing so will protect against many threats such as buffer overflow exploits or the execution of malicious code loaded into RAM.

11. POSIX Compliant:
    Yes.

3.4 RIOT

RIOT[5] aims to implement all relevant IoT open standards in a secure, durable and privacy-friendly way. The source code is available on GitHub under LGPLv2.1. RIOT was developed with the particular requirements of IoT in mind and aims for a developer friendly programming model and API. It is also a microkernel-based OS with multi-threading support (Fig. 3.4), thus making it more robust against bugs in single components.

Even though RIOT is not officially supported by Nordic, most of the Nordic libraries are OS agnostic, meaning that they can easily be ported to an RTOS. However, the ones that are supported are only FreeRTOS and Zephyr, and thus those are the ones for which they provide support in development.

1. Toolchain and IDE Suport:
   Well known tools for building (gcc, make), static analysis (cppcheck, coccinelle), dynamic

[5] https://doc.riot-os.org/

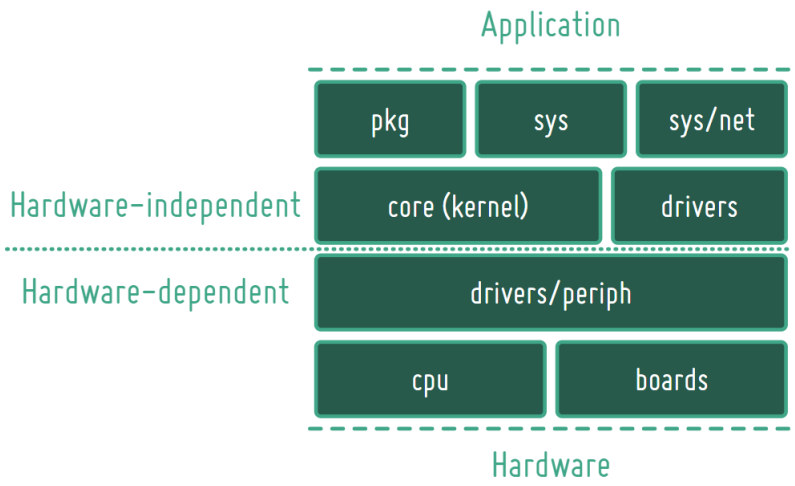

Figure 3.4: The architecture of the RIOT software. Picture taken from [2]

analysis (Valgrind), network sniffing (Wireshark), standard debugger (gdb), performance profilers (gprof) and unit testing (embUnit) can be used. It also supports some testbeds such as IoT-LAB. RIOT does not include any IDE, which might make the development more complex.

2. Maturity:
   - First release: August 2018
   - Latest release: October 2020
   - Update rate: 3 months
3. Memory footprint:
   - Minimal data memory size of 1.5 kB
   - Minimal program memory size of 5 kB [5]
4. Dynamic memory:
   RIOT makes extensive use of preallocated structs to cater both for reliability and real-time requirements. It also supports dynamic allocation. When it comes to the Kernel, it only allows the exclusive use of static memory.
5. Scheduling:
   The kernel uses a scheduler based on fixed priorities and preemption. A class-based run-to-completion scheduling policy is used: the highest priority (active) thread runs, only interrupted by interrupt service routines (ISRs). There are no context switches mimicking parallel execution of tasks of the same priority.

   The scheduler used in RIOT is tickless. It does not depend on CPU time slices and periodic system timer ticks. The system does not need to periodically wake up unless something is

actually happening, e.g., an interrupt triggered by connected hardware. If no other thread is in running state and no interrupt is pending, the system switches by default to the idle thread – which has lowest priority. The idle thread in turn switches to the most energy-saving mode possible, thus optimising energy consumption.

6. Synchronisation and inter-task communication:

   - Mutex Mutexes include PIP (Priority Inheritance Protocol) to deal with priority inversion.
   - Semaphore
   - Event Queues
   - Message Queues
   - Mailboxes

   These mechanisms are submodules of the kernel and are thus optional, and compiled only on demand.

7. IO

   - Timer
   - GPIO
   - PWM
   - ADC
   - I2C
   - SPI
   - UART
   - USB
   - QSPI (w/ external flash)

   RIOT does not include vendor libraries. According to [4], this is done on purpose to minimise code duplication and to avoid that design decisions tie RIOT to a particular technology.

8. Networking:

   - CoAP
   - MQTT
   - 802.15.4
   - Ethernet
   - BLE 5.0
   - ZigBee

- LPWAN
- 6LoWPAN
- RPL
- CAN (Controller Area Network)

9. Storage & Display Constant file system resident in arrays and dynamic device file system. Provides support for FATFS and little FS.

10. Shell Access:
    RIOT provides a command-line interpreter (CLI) similar to a shell in Linux.

11. Memory Protection:
    Does not have a memory protection unit.

12. POSIX Compliant:
    Yes.

3.5 Zephyr OS

Zephyr OS[6] is based on a small-footprint kernel designed for use on resource-constrained and embedded systems: from simple embedded environmental sensors and LED wearables to sophisticated embedded controllers, smart watches, and IoT wireless applications. Zephyr is permissively licensed using the Apache 2.0 license. The software architecture is shown in Fig.3.5.

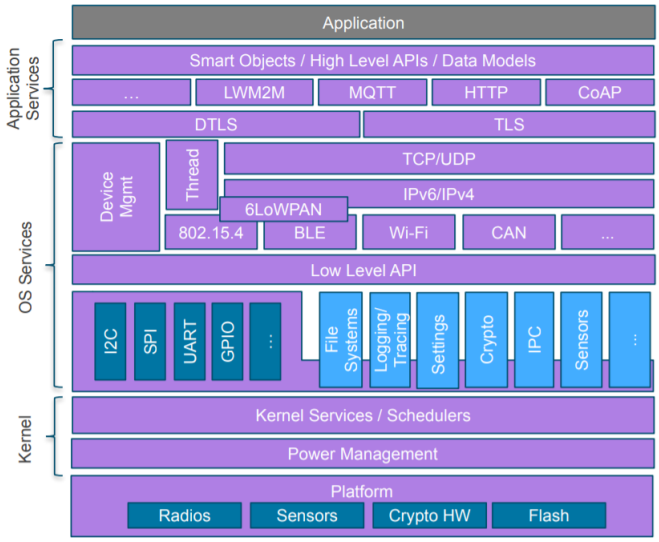

Figure 3.5: The architecture of the Zephyr OS software. Picture taken from [35]

[6] www.zephyrproject.org

1. Toolchain and IDE Suport:
   The Zephyr Software Development Kit (SDK) contains toolchains for each of Zephyr's supported architectures, which provide a compiler, assembler, linker, and other programs required to build Zephyr applications. These tools include west.
   Besides this, Zephyr can be used along with PlatformIO, without the need of managing and fetching the files.

2. Maturity:

   - First release: February 2016
   - Latest release: September 2020
   - Update rate: 3 months

3. Memory footprint:

   - Minimal program memory size of 2-3 kB. The program memory size for other configurations is presented on Zephyr website[7]Zephyr Website
   - Data memory size stated as "small footprint"

4. Dynamic memory:
   Zephyr provides a collection of utilities that allow threads to dynamically allocate memory, either by allocating fixed size memory blocks (Memory Slabs) or bocks of any size from the Memory Pool, using the "buddy memory allocation" algorithm. The memory pool is configured at compilation time, but this implementation is still not deterministic.

5. Scheduling:

   Zephyr provides a comprehensive set of thread scheduling choices:

   - Cooperative and Preemptive Scheduling
   - Earliest Deadline First (EDF)
   - Meta IRQ scheduling implementing "interrupt bottom half" or "tasklet" behavior
   - Timeslicing for preemptible threads of equal priority

6. Synchronisation and inter-task communication:

   - Mutex
     The thread that has locked a mutex is eligible for priority inheritance, and it is possible to configure the maximum priority the task can take.
   - Semaphore
   - FIFO, LIFO and Stack
   - Message Queues

[7]https://docs.zephyrproject.org/latest/samples/basic/minimal/README.html

- Mailboxes

7. IO
   - Timer
   - GPIO
   - PWM
   - ADC
   - RTC (Real-time counter)
   - I2C
   - SPI
   - UART
   - USB
   - Flash memory interface
   - Audio peripherals: I2S
   - DMA
8. Networking
   - CoAP
   - MQTT
   - Ethernet
   - 802.15.4 with 6Lo
   - BLE 5.0 with 6Lo
   - WiFi with IP offload
   - CANbus with 6Lo
9. Storage & Display:
   Virtual File System Interface with LittleFS and FATFS Support.
10. Shell Access:
    A multi-instance shell subsystem with user-friendly features. Support for static commands and dynamic sub-commands.
11. Memory Protection:
    Implements configurable architecture-specific stack-overflow protection, kernel object and device driver permission tracking.

    It also offers thread isolation with thread-level memory protection userspace, and it allows to configure memory domains
12. POSIX Compliant:
    Yes.

3.6 Summary

This chapter presented the RTOS selection process that allowed narrowing an initial set of 192 options down to a short list of five. Then, we carried out a brief description of each of these RTOS, focusing on certain parameters of interest. This supports a qualitative comparison that is shown in Table 3.1.

		Apache MyNewt	**Contiki-NG**	**FreeRTOS**	**RIOT**	**Zephyr**
Maturity	First release	March 2016	November 2017	December 2003	August 2018	February 2016
	Latest release	April 2020	May 2020	December 2020	October 2020	September 2020
	Update rate	7 months	6 months	2 months	3 months	3 months
Memory footprint	Data memory size	8-16 KB	> 10 KB	<1 KB	>1.5 KB	"small footprint"
	Program memory size	64-128 KB	100 KB	6-12 KB	>5 KB	2-3 KB
Dynamic memory		yes	no	yes	yes	yes
Scheduling		preemptive, priority based	cooperative	preemptive, tickless, priority based	preemptive, tickless, priority based	configurable, all of the other and more
Programming model		multi-threaded	event-driven with possible thread implementation	multi-threaded	multi-threaded	multi-threaded
Shell		(no-information)	yes	yes	yes	yes
Memory protection		no	no	yes	no	yes
POSIX compliant		yes	no	yes	yes	yes

Table 3.1: Qualitative comparison of the RTOS in the short list.

The table shows more clearly the differences between the RTOS. Apache MyNewt stands out because of the program memory size, which is clearly larger then the others. Contiki-NG stands out for the same reason. Besides, it is not POSIX compliant.

We thus select, for the tests that will be performed, the remaining three RTOS: FreeRTOS, RIOT and Zephyr. This is an interesting set: Zephyr is an OS whose main concern is the easiness of development and flexiblity, FreeRTOS is a full real-time OS, and RIOT seems to be an interesting middle-way option to be explored.

However, it is important to point out again that FreeRTOS consists just on the Kernel. The goal of taking it into account in this comparison is to understand if it is a good option to be used along with Nordic's SDK. If a specific application is growing in complexity and it gets harder to handle the complexity of context switches and their glitches then an RTOS such as FreeRTOS provide threads to mitigate that problem. The drivers from the SDK can be reused.

The next chapter will address the definition of an adequate benchmark that will allow a quantitative comparison and final selection of the RTOS to be used on Kallisto.

Chapter 4

Benchmarking RTOS for Kallisto

This chapter presents the results of a runtime (quantitative) performance comparison among the selected RTOS, to complement the qualitative comparison based on static features that we discussed in the previous chapter. The runtime performance comparison is carried out resorting to a suitable benchmark that provides relevant dynamic metrics. These metrics were explained with detail previously in Chapter 2.

1. RTOS internal management
   (a) Context Switching overhead
   This will be measured according to Stewart in [34].
   (b) Interrupt Service Latency
   This will be measured according to Stewart in [34].
   (c) Maximum Reaction Rate
   As explained before, the Maximum Reaction Rate will be determined with the method proposed by ISA (according to Aroca and Caurin in [3]).
   (d) Jitter time on task execution
   According to Mumolo et al. in [21].
2. Inter-Process Communication
   These measurements will be determined using the four scenarios described by Ungurean in [39]. The services that will be measured are the following:
   (a) Mailbox
   (b) Queue
   (c) FIFO, LIFO and Stack
   (d) Message Queue
   (e) Message and Stream Buffers
3. Synchronisation Primitives These measurements will be determined using the four scenarios described by Ungurean in [39].

(a) Semaphores

(b) Mutexes

4. Notification Mechanisms These measurements will be determined using the four scenarios described by Ungurean in [39].

   (a) Thread Flags

   (b) Events

   (c) Direct to Task Notifications

The following results are the average of the measurements. For the tests made with software timer, 500 measurements were taken. For the tests in which we measured the time with the oscilloscope, 100 measures were taken. The deviation was very small for all these cases, less than $+/-5\%$ so we opted for not showing it. The results and the respective figures are organised in a way that facilitates the comparison of the RTOS.

4.1 RTOS internal management

4.1.1 Context Switch

The time for the context switch is clearly different for the three RTOS, specially in what comes to Zephyr, as we can see in Figure 4.1. In this case, this interval is approximately twice the value of RIOT, with a difference of almost of 14μsec. The difference between FreeRTOS and RIOT is less than 4μs. This is a measurement that will affect the inter-process communication performance, since some of the experiments is to measure their performance when they cause a context switch.

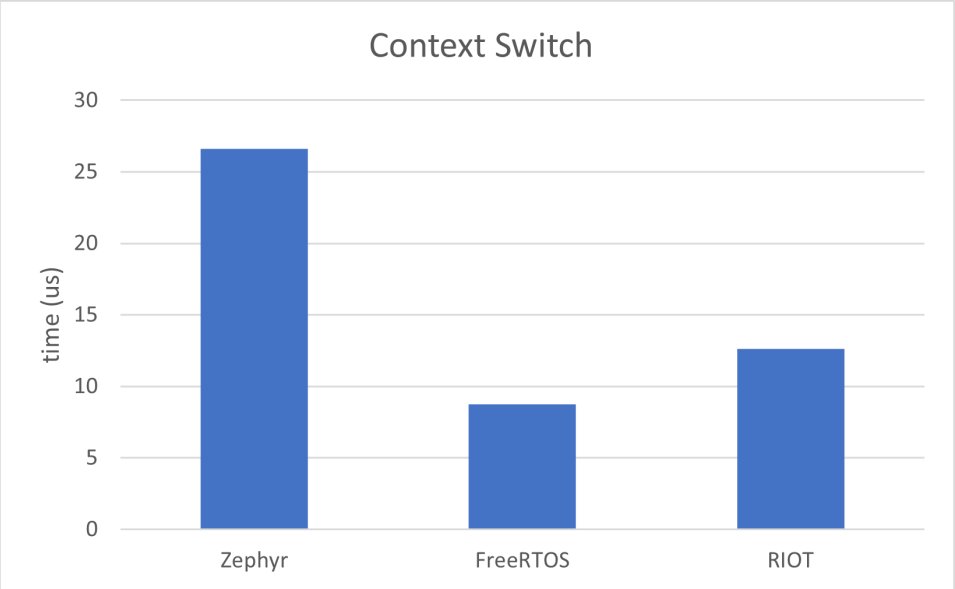

Figure 4.1: Average time for context switching, measured in μs.

4.1.2 Task Execution Jitter

The task execution jitter is the delay between the instant in which a periodic process is activated and the instant in which it actually starts. The jitter of the task execution is clearly larger in RIOT, since it takes almost 2μs for the task to actually start, as we can see in Figure 4.2.

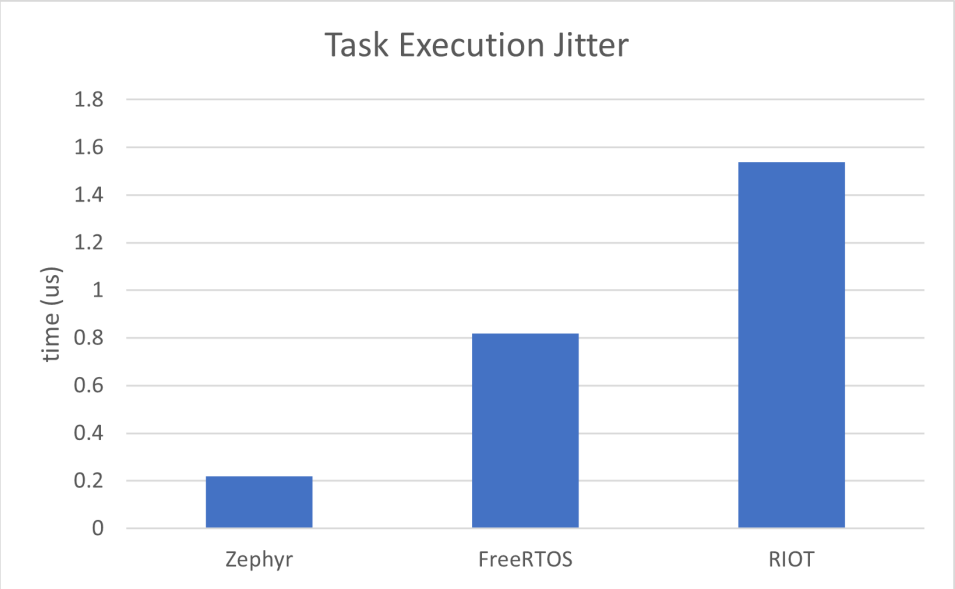

Figure 4.2: Task execution jitter measured in μs.

4.1.3 Interrupt Latency

The interrupt latency is not very different in Zephyr and FreeRTOS, since the difference approximately 1μsec, as shown in Figure 4.3 For RIOT, it is particularly small. Note that this is a considerably important feature in RTOS, because it shows how fast the OS attends an external event.

4.1.4 Maximum Reaction Rate

Figure 4.4 shows the maximum interrupt frequency that can be handled by each RTOS with reliability. The maximum frequency is related, among other factors, with the interrupt latency, so it was expected that RIOT would show a good behaviour compared to the others. In fact, the maximum frequency it can handle depends on the time it takes to attend the interrupt (interrupt latency) and the time it takes to set the output. FreeRTOS has a large interrupt latency, which limits the maximum frequency it can arrive to.

Zephyr shows a worse behaviour in almost every measurement of the RTOS internal management, except for the task execution jitter, which is much smaller. In this test, RIOT shows a larger jitter, even though the difference is only approximately 1.3μs. However, even though it is called jitter, this does not mean that RIOT is not deterministic: it is important to emphasise once more that, both for RIOT and for the others, the measurements have a very small deviation.

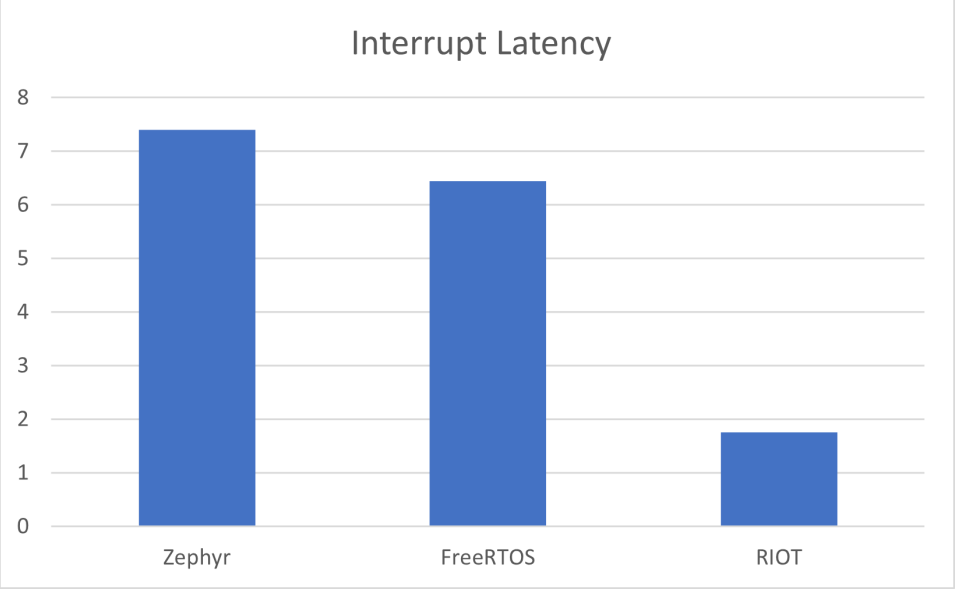

Figure 4.3: Interrupt latency measured in μs.

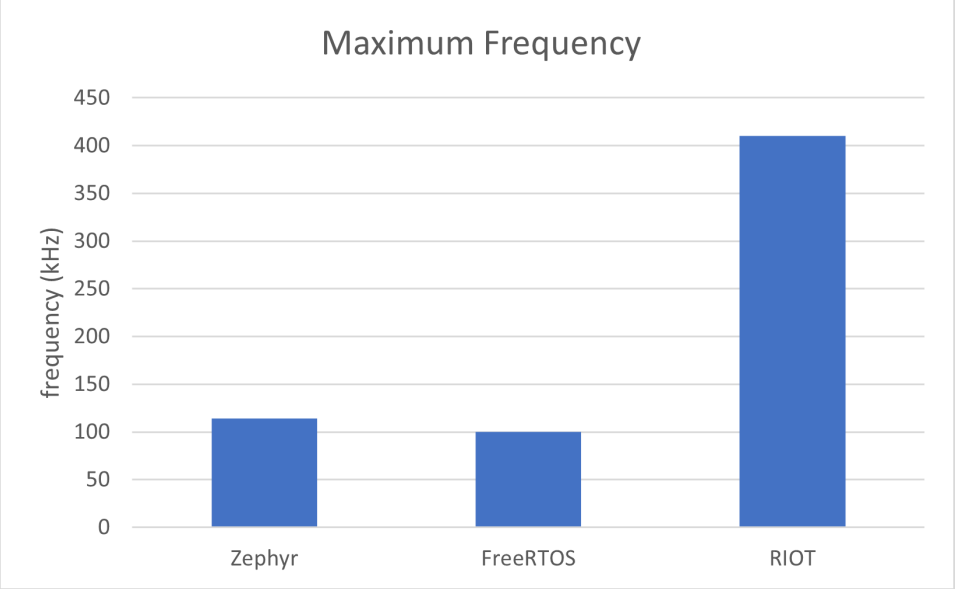

Figure 4.4: Maximum frequency in kHz.

4.2 Intertask Communication

4.2.1 Queue, LIFO and Stack handling

In both Zephyr and FreeRTOS, queues are the primary form of intertask communications.

Zephyr defines some specific types of queues. A FIFO is a kernel object that implements a traditional first in, first out (FIFO) queue, allowing threads and ISRs to add and remove data items of any size. The FIFO is what they generally refer to when talking about queues. On the other hand, a LIFO is a kernel object that implements a last in, first out queue. A stack is a a kernel object that implements a LIFO queue, but it works only with integer data values.

The FreeRTOS queue is based on a FIFO. The main difference is that in Zephyr you must always pass the pointer to the data you want to send, so you must allocate a buffer to do it. In FreeRTOS, you can pass the data directly.

The results of the comparison of the types of queues in Zephyr and the queues in FreeRTOS is shown is Figure 4.5. In fact, we can see that the differences between the types of queues in Zephyr are not significant (less than 7%). In what comes to the queues in FreeRTOS, the difference is quite large, approximately 10μs for the second test, but on the other hand the time for the primitives when they do not cause context switching is higher, and this is due to the fact that the time for context switch is also higher in Zephyr.

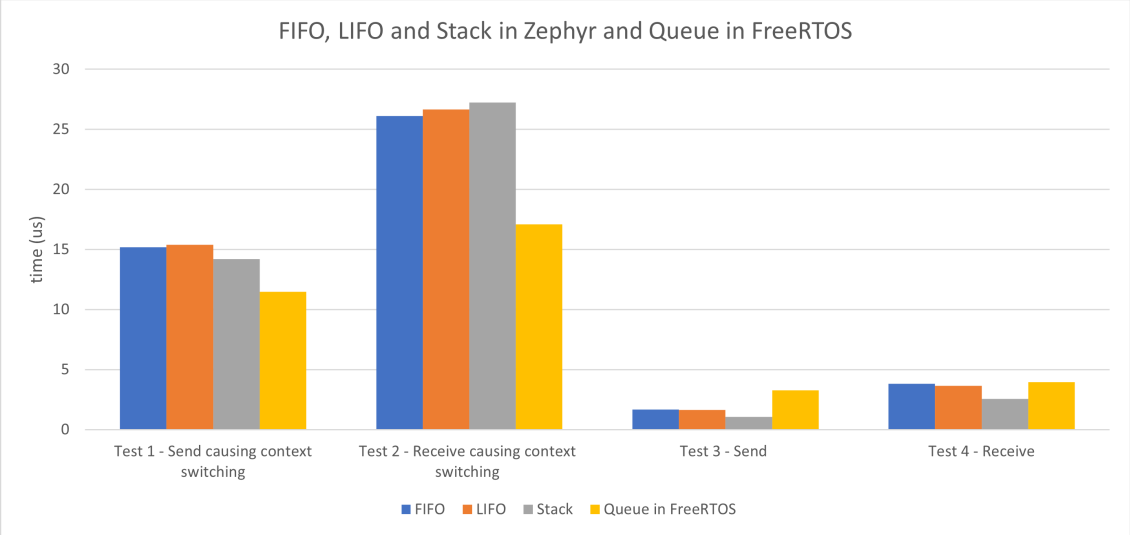

Figure 4.5: Average time for FIFO and LIFO handling primitives, measured in μs, for Zephyr, only.

4.2.2 Mailbox, Message Queue and Stream Mechanisms

Both RIOT and Zephyr implement message queues. In Zephyr, a message queue is a kernel object that allows threads and ISRs to asynchronously send and receive fixed-size data items. The results are also very similar to the other primitives based on the queue, as expected.

In RIOT, the content can either be provided as a 32-bit integer or a pointer and messages can be sent and received, blocking and non-blocking. It supports both synchronous and asynchronous message and they can also be sent and received by threads and ISRs.

Zephyr and RIOT also implement mailboxes. In Zephyr, a mailbox goes beyond the capabilities of a message queue: it allows threads to send and receive messages of any size synchronously or asynchronously, and not just asynchronously as in the queues. For instance, for a synchronous send operation, the operation normally completes when a receiving thread has both received the message and retrieved the message data.

In RIOT, however, mailboxes are the equivalent to the generic queues in Zephyr, and they do not support the synchronous mode of operation, unlike the previous message queues.

The results of both primitives are shown in Figure 4.6. Zephyr takes much longer than RIOT for mailboxes, even when compared to the other primitives. The difference is around 20μs for the second test. This is due to the fact that, as it was explained, the mailboxes in Zephyr are much more complex than the mailboxes in RIOT. In fact, it would make more sense to compare the message queues in Zephyr to the Mailboxes in RIOT: in that case, the difference is not so high for tests 1, 3 and 4.

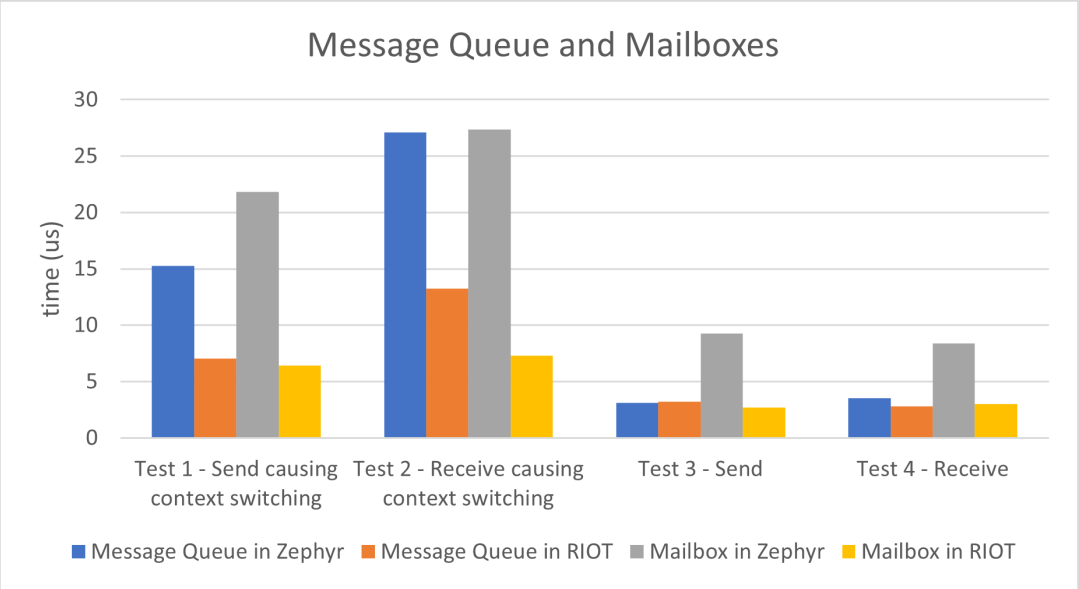

Figure 4.6: Average time for mailbox handling primitives, measured in μs.

FreeRTOS supports message and stream buffers. Stream buffers are kernel objects that are used in a single reader single writer scenario (for instance, passing data from an interrupt service routine to a task) and data is passed by copy. Stream buffers pass a continuous stream of bytes and message buffers pass variable sized but discrete messages. The stream buffers show a slightly better performance when compared to message buffers, as we can see in Figure 4.7 because these are implemented on top of stream buffers.

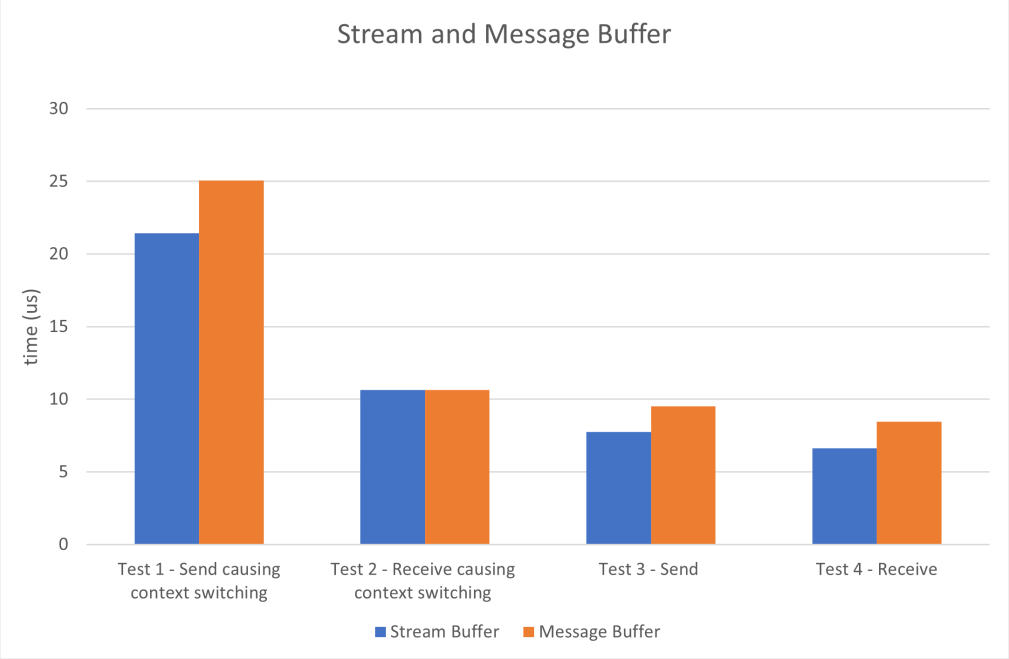

Figure 4.7: Time for the primitives associated to the message and stream buffer, measured in μs.

4.3 Synchronisation primitives

4.3.1 Semaphores handling

RIOT does not implement semaphores, so the test was done for Zephyr and FreeRTOS, only, and the results are presented in Figure 4.8. Here the differences are not as relevant, as it depends on the test. For instance, Zephyr shows a greater difference among the primitives when these cause context switching, but a smaller difference when these do not cause context switching. In fact, this difference is not relevant, since it represents at most 2μs.

4.3.2 Mutex handling

All three RTOS implement mutexes. In what comes to the time taken by the mutex handling primitives, RIOT shows a good behaviour when compared to the others, as it can be seen in Figure 4.9. Zephyr is distinguishable for the large times involved, specially when the primitives cause a context switching, just as it happened with the semaphores.

4.4 Notification mechanisms

RIOT includes specific flags, called Thread Flags, that allow notifying threads of conditions in a race-free and allocation-less way. These flags can be set or unset from ISRs, other threads or by the thread itself. Sometimes, it is preferable over messages or mutexes, as it allows signalling more than one event and it is guaranteed that a flag can always be set, since it does not depend on queue space or on the fact that another thread is waiting.

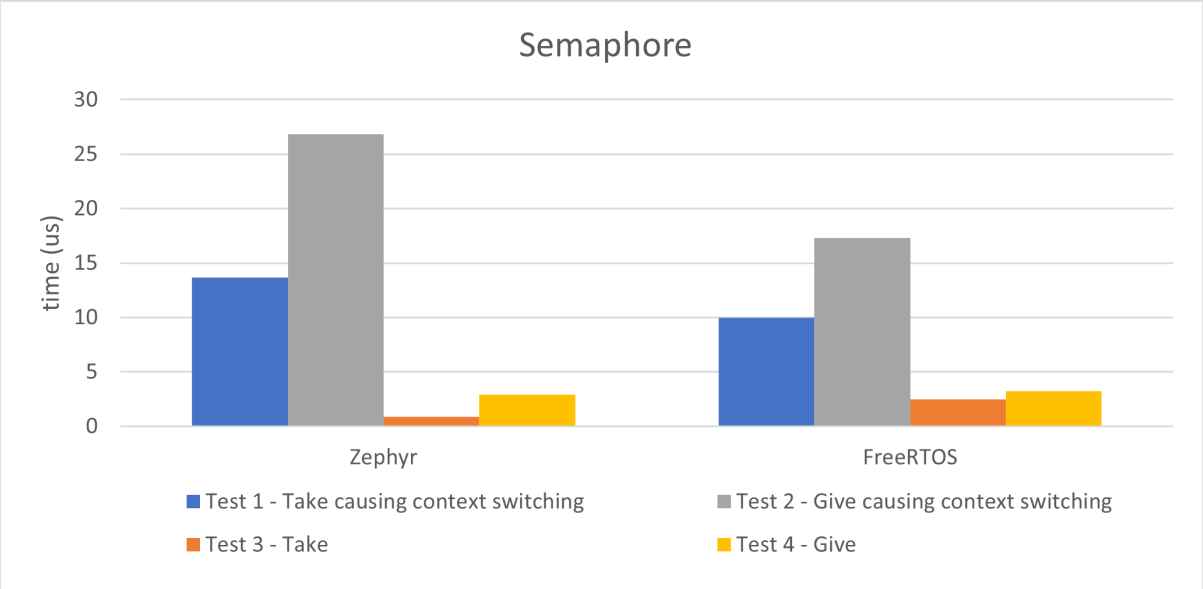

Figure 4.8: Average time for semaphore handling primitives, measured in μs.

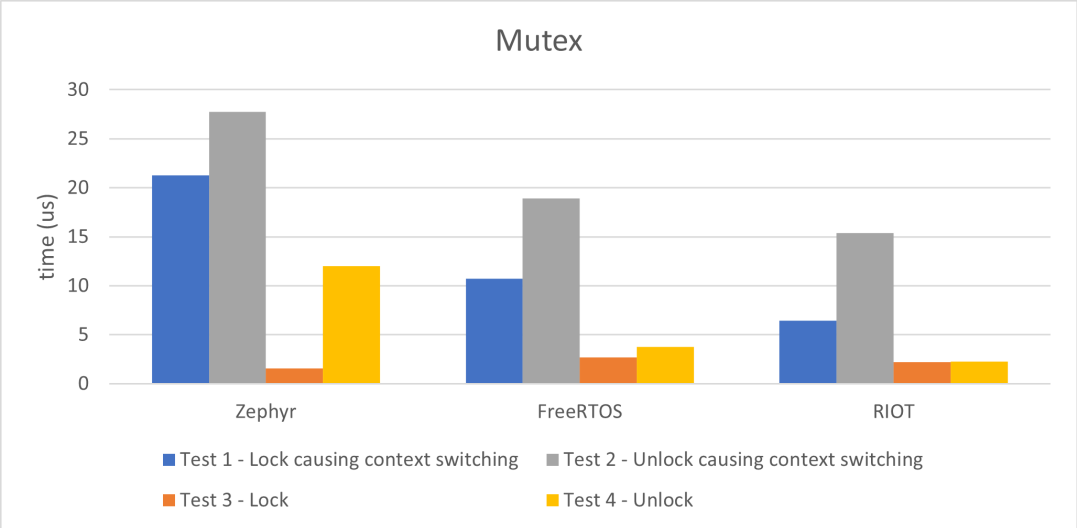

Figure 4.9: Average time for mutex handling primitives, measured in μs.

In FreeRTOS, event bits are used to indicate if an event has occurred or not. Event bits are also referred to as event flags.

FreeRTOS supports direct to task notifications. Each task has an array of task notifications, and each can be 'pending' or 'not pending'. A direct to task notification is an event sent directly to a task, rather than indirectly to a task via an intermediary object such as a queue, event group or semaphore.

It is possible to implement semaphores, mailbox and events with direct to task notifications, in a lightweight alternative. These should be supposedly faster, and indeed the obtained results show that they are, but only one task that can be the recipient of the event.The time for the primitives associated to the events with notifications are faster than the ones for normal events, as it can be seen in Figure 4.10.

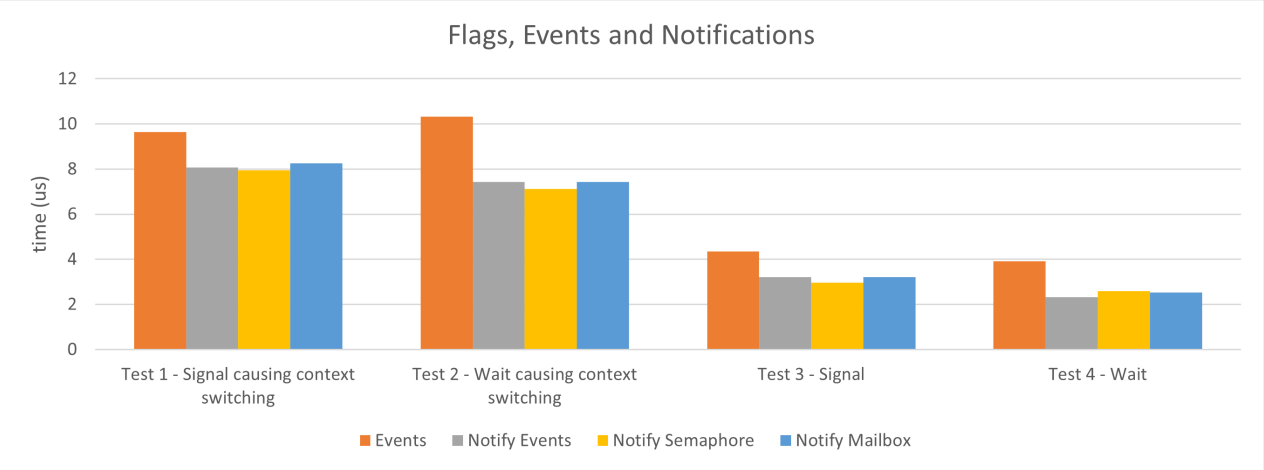

Figure 4.10: Average time for the primitives associated to the notifications as mailboxes, measured in μs.

4.5 Summary

The results of the experiences presented in the last sections are summarised in Table 4.1, along with the main qualitative characteristics from the RTOS presented in Chapter 2. To facilitate the interpretation of the results, only the results for tests 3 and 4 are presented for the intertask communication, synchronisation primitives and notification mechanisms. These tests are, respectively, the execution time for the signal and the wait primitives, when these do not cause context switching. The results for the time measurements are all presented in μs, except for the task execution jitter that is presented in *ns* (as explicitly written in the table).

Some of the most difficult points to establish a clear comparison between the behaviour of the three RTOS are the intertask communication services. As it was clear, they do not support the same services.

In what comes to data passing, for instance, Zephyr offers a lot of services. These differ on the size of the data you can send and receive, if they work on the ISRs, or on the structure of

the queue. However, as it was mentioned, the performance for these primitives (queue, FIFO, LIFO and stack) were very similar (the results are always around 15 μs, 26 μs, 1.5 μs, 3.6 μs, respectively for tests 1, 2, 3 and 4). Mailboxes, however, show a worse performance, due to their complexity of implementation.

On the other hand, FreeRTOS offers queues, mailboxes implemented with notifications and stream and message buffers. Overall, it is better than Zephyr in what comes to the queues (the only exception is test 3, when sending a message without context switching). The implementation of the mailbox with notifications shows a better performance, specially when the primitives cause context switching, but with the limitations presented in Section 4.4.

At last, RIOT offers mailboxes and messages. In mailboxes, it is clearly better than Zephyr and better than the mailbox implemented with notifications, on FreeRTOS. In what comes to message queues, it is again better than Zephyr. If compared to the message and stream buffers on FreeRTOS, it also shows a much better behaviour. It offers more services than FreeRTOS, because it can be used in an asynchronous way, but less when compared to Zephyr.

When talking about notification mechanisms, the flags in RIOT show a better performance, even when compared to direct to task notifications, but this difference is not significant.

In what comes to the synchronisation services, Zephyr and FreeRTOS offer semaphores and mutexes, and overall the times for FreeRTOS are smaller for both cases. The implementation with notifications shows even smaller times, but once again with the limitations of the notifications. RIOT has an implementation of mutexes, which shows a better behaviour in comparison to the others. However, it is clear that the services it offers in what comes to synchronisation are very limited, once again.

In chapter 2, we compared both the maturity of the RTOS, their communities ad the variety of tools they offer. In what comes to their maturity, the amount of available documentation and examples offered by Zephy were a great help when getting started. The contrary thing happened in RIOT, in which the documentation seems to be quite incomplete. The same thing happens with the community: RIOT has its own forum just like FreeRTOS and Zephyr, but unfortunately it is not as active as one would expect. As to the tools they offer, in Zephyr and FreeRTOS the availability of an IDE makes it much easier both to develop the application and to debug it.

On Zephyr, the configurations of Kernel objects or drivers are in the Kconfig file. In FreeRTOS (using the Nordic SDK), the same thing happens but there are some issues with legacy drivers and overwriting files that were not yet solved by Nordic.

		FreeRTOS	RIOT	Zephyr
Maturity	First release	December 2003	August 2018	February 2016
	Latest release	December 2020	October 2020	September 2020
	Update rate	2 months	3 months	3 months
Memory footprint	Data memory size	<1 KB	>1.5 KB	"small footprint"
	Program memory size	6-12 KB	>5 KB	2-3 KB
Dynamic memory		yes	yes	yes
Scheduling		preemptive, tickless, priority based	preemptive, tickless, priority based	configurable, all of the other and more
Programming model		multi-threaded	event-driven with possible thread implementation	multi-threaded
Shell		yes	yes	yes
Memory protection		yes	no	yes
POSIX compliant		yes	yes	yes
Context Switch		8,7	12,6	26,6
Task Execution Jitter (ns)		819	1638	218
Interrupt Latency		6,44	1,76	7,4
Maximum Reaction Rate (kHz)		100	410	114
Intertask Communication	FIFO	3.27 / 3,97	NA	1,67 / 3,83
	LIFO	NA	NA	1,63 / 3,64
	Stack	NA	NA	1,05 / 2,56
	Message Queue	NA	3,21 / 2,84	3,14 / 3,53
	Mailbox	NA	2,71 / 3,02	9,30 / 8,38
	Stream Buffer	7,75 / 6,62	NA	NA
Synchronisation Primitives	Semaphores	2,46 / 3,21	NA	0,89 / 2,89
	Mutexes	2,71 / 3,78	2,21 / 2,27	1,59 / 12,02
Notification Mechanisms	Flags (RIOT) and Events (FreeRTOS)	4,347 / 3,906	2,961 / 2,394	NA
	Direct To Task Notifications	Events: 3,21 / 2,33 Mailboxes: 3,21 / 2,52 Semaphores: 2,96 / 2,58	NA	NA

Table 4.1: Full comparison of the RTOS. The values for time measurements are all in μs, except stated otherwise. The values (xx / yy) are the results of tests 3 and 4, respectively signal and wait for the primitive without causing context switching.

Chapter 5

Bluetooth Low Energy

According to Tosi et al. in [37], Bluetooth Low Energy (BLE) is a wireless technology whose good performance and diffusion makes it one of the best alternatives for short-range communication in many applications, such as healthcare, consumer electronics, smart energy and security. BLE started as a part of the Bluetooth 4.0 Core Specification, with the goal to design a radio standard specifically optimised for low cost, low bandwidth, low power and low complexity.

5.1 BLE Stack Architecture

The full stack of a BLE-based application is represented in Figure 5.1. It includes:

- Application (*app*): this represents the user application that relies on BLE.
- Host, which includes:
  - Generic Access Profile (GAP)
  - Generic Attribute Profile (GATT)
  - Logic Link Control and Adaptation Control (L2CAP)
  - Attribute Protocol (ATT)
  - Security Manager Protocol (SMP)
  - Host Controller Interface on the Host side (HCI)
- Controller, which includes:
  - Host Controller Interface on the Controller side (HCI)
  - Link Layer (LL)
  - Physical Layer (PHY)

The BLE stack spans over the Host and Controller. At the bottom, i.e., the Physical Layer (PHY), the BLE radio band goes from 2.4000 GHz to 2.4835 GHZ. It is divided in 40 channels,

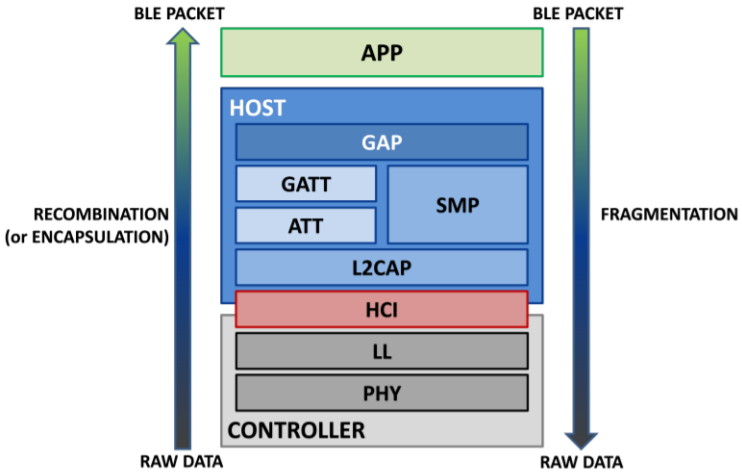

Figure 5.1: Bluetooth Low Energy Stack. Picture taken from [37]

from which three (37, 38 and 39) are reserved for advertising packets and the others are meant to exchange data packets in connections. The PHY layer originally worked at 1 Mbps. The information is transmitted using a scheme called Gaussian Frequency-Shift Keying (GFSK), so this rate means that it can transmit 1 million symbols per second (1 bit per symbol).

Bluetooth 5 introduced a new transmission mode with a doubled symbol rate, i.e. it works at 2 Mbit/s. This allows a user to double the number of bits sent over the air during a given period, or conversely reduce energy consumption for a given amount of data. The PHY using a transmission rate of 1Mbit/s was renamed LE 1M PHY, and the new mode at a doubled symbol speed was named as the LE 2M PHY. Default operation starts with LE 1M PHY leaving it to the application to switch to the LE 2M PHY.

Bluetooth 5 also introduced the long-range mode, also known as Coded PHY. The raw data is still transmitted at the rate of 1 Mbps, but the encoding brings down the bit rate to 500 kbps or 125kbps, depending on the configuration. This mode decreases the data rate, but allows to increase the range.

The Link Layer (LL) interfaces with the Physical Layer and defines the roles that a device can have:

- Advertiser: a device that sends advertising packets.
- Scanner: a device that scans advertising packets.
- Master: a device that initiates a connection and then manages it.
- Slave: a device that accepts a connection request and then responds to the master.

The first two roles are relative to the context of broadcasting and the last two roles are defined in the context of an active connection. The difference between the two will be better explained later on.

Above the Link Layer, the Host Controller Interface (HCI) takes care of the communication between the host and the controller. The Bluetooth specification allows several configurations: the host and the controller might or might not be in the same chip. This layer manages the communication between them across a serial interface. According to Townsend et al. in [38], typical examples of the first type of configuration include most smartphones, tablets, and personal computers, in which the host runs in the main CPU, while the controller is located in a separate hardware chip connected via UART or USB. On the other hand, in the nRF52840 controller, the host and the controller are in the same chip.

The L2CAP layer is at the bottom of the host and is responsible for two important functions. Firstly, it takes multiple protocols from the upper layers and it encapsulates them into the standard BLE packet format. These protocols are the Attribute Protocol (ATT) and the Security Manager Protocol (SMP), that will be described later.

It is also responsible for performing fragmentation and recombination of BLE packets, as it is represented in Figure 5.1. BLE packets sent by the application are divided into smaller ones that can fit into the BLE packets that will be sent. When receiving these smaller packets, they are reassembled together into a single BLE packet that will be sent to the upper layers and then to the application.

The Security Manager Protocol (SMP) includes some security algorithms that deal with the encryption and decryption of the data packets. This protocol defines two roles during the establishment of a connection: the initiator and the responder, which will later turn respectively into the master and the slave.

The Attribute Protocol (ATT) is based on a client-server architecture: the client requests data from the server, and the server sends data to the client. Usually, the client and server roles correspond to the master and slave of the Link Layer, but a device can be a client, a server, or both. The ATT defines the concept of attribute, too, represented in Figure 5.2. An attribute is a data container to which the protocol assigns an identifier called *handle*, a Universally Unique Identifier (UUID), a set of permissions and a value. The UUID specifies the type and nature of the data contained in the value. The ATT protocol also specifies the set of possible operations.

Handle	Type	Permissions	Value	Value length
0x0201	$UUID_1$ (16-bit)	Read only, no security	0x180A	2
0x0202	$UUID_2$ (16-bit)	Read only, no security	0x2A29	2
0x0215	$UUID_3$ (16-bit)	Read/write, authorization required	"a readable UTF-8 string"	23

Figure 5.2: Examples of attributes. Picture taken from [38].

The Generic Attribute Profile (GATT) encapsulates the ATT layer. It establishes in detail how to exchange the data over BLE, in what comes both to the transfer procedures and formats. The data is organised in a hierarchy, as in Figure 5.3: each service can group related characteristics.

A characteristic represents a type of user data. The attributes are the smallest defined data entity, and they consist in pieces of information that can be data or metadata that describe the structure of characteristics or services. An example of this architecture for the Heart Rate Service, taken from [38], is presented in Figure 5.4. This service provides the heart rate to the central device. GATT uses the same roles as ATT, i.e. client and server. BLE defines some standard services and characteristics, but it is possible for the developers to define their own services and characteristics by creating themselves the UUIDs.

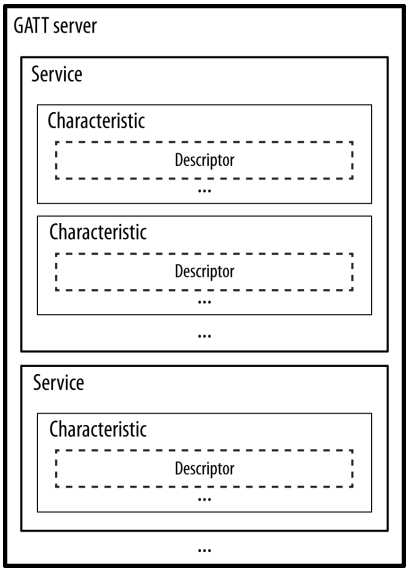

Figure 5.3: Hierarchy defined by GATT. Picture taken from [38].

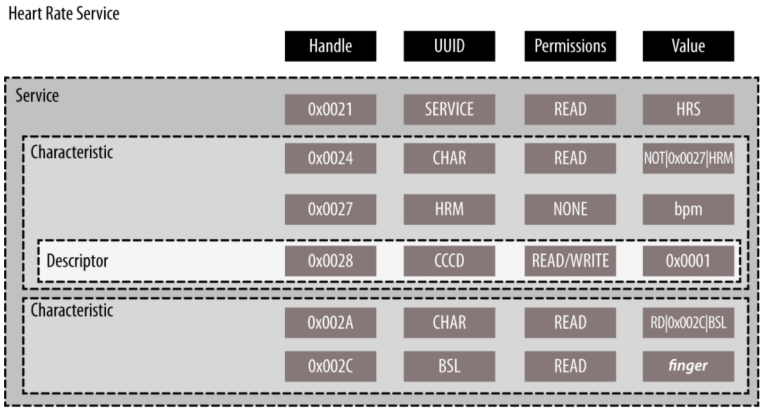

Figure 5.4: GATT Heart Rate Service. Picture taken from [38].

Finally, the Generic Attribute Protocol (GAP) is the highest level of the stack. It defines the roles that a device can have, as well as the procedures, and it also manages the connection establishment and the security.

5.2 BLE Communication

The BLE communication includes two modes: broadcasting and connection-oriented.

5.2.1 Broadcasting

This mode does not require a connection establishment. Advertising packets are sent out by a device and any scanning or receiving device in the listening range will receive them. This is the fastest way to transmit to many devices at the same time, without the overhead of a full connection establishment, but it is not the recommended way if the data is subjected to security and reliability requirements.

The advertising packets can be used to broadcast data, or they might be used by a slave to show availability to establish a connection with a master. During broadcasting, a device might have one of two roles: broadcaster/advertiser, that periodically sends an advertising packet, and observer/scanner that continuously scans, at periodic intervals, if there is any advertising packet, such as in Figure 5.5. These packets are sent at a fixed rate defined by the advertising interval. The scan interval and the scan window parameters define how often and for how long does the device scan.

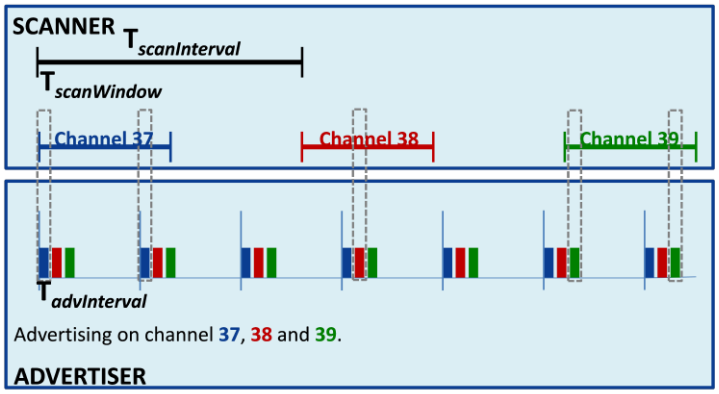

Figure 5.5: Scan and advertisement time behaviour. Picture taken from [37]

There are two types of scanning, as represented in Figure 5.6.

- Passive Scanning: the scanner listens, and the advertiser does not know if the packets were received by the scanner.

- Active Scanning: the scanner sends a Scan Request packet after receiving an advertising packet. When the advertiser receives it, it responds with a Scan Response packet. Like this, the advertiser can send more data to the scanner.

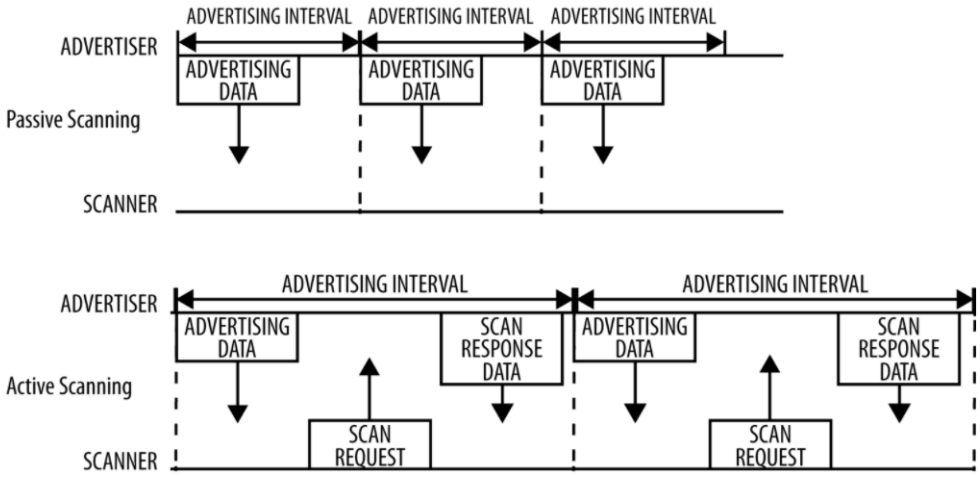

Figure 5.6: Passive and active scanning. Picture taken from [38]

5.2.2 Connection-Oriented

A connection consists on a periodic data exchange between two devices. No device outside the connection has access to that data. The two devices are:

- Central (master): it scans for advertising packets and responds for initiating a connection. During the connection, it is responsible for starting the periodical data exchanges.
- Peripheral (slave): it sends the advertising packets during the broadcasting phase, and then it accepts the connection request from the master. During the connection, it exchanges data with the master.

Note that during the broadcasting phase that takes place before establishing a connection, the central and the peripheral correspond respectively to the observer/scanner and the broadcaster/advertiser and they can also be called by responder and initiator, using the terminology defined by the SMP layer. A connection follows a predefined timing behaviour, as well shown in Figure 5.7.

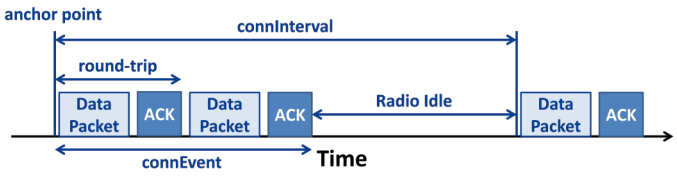

Figure 5.7: Parameters of connection. Picture taken from [37].

The devices exchange data during the connEvent period, and the rest of the time there is no communication (Radio Idle). The master initiates the data exchange in periodically with a period called Connection Interval (connInterval). The start of each connInterval is called anchor

point, and it triggers a number of acknowledged packet transmissions that fit in the configured connEvent. If the devices do not exchange data in consecutive connEvents for more than a certain time, called Connection Supervision Timeout (connSupervisionTimeout), then the connection is terminated. Finally, the Connection Slave Latency is the number of connEvents that can be ignored by the slave without the risk of disconnecting. The corresponding time should be less than the Supervision Timeout.

5.2.3 Expanding the BLE Packet

The structure of a BLE Packet at the PHY layer is shown in Figure 5.8. The Preamble (PRE) is 1 or 2 bytes depending on the radio data rate, for 1 Mbps or 2 Mbps, respectively, and it conveys this information to the receiver. The Access Address (AA) identifies the communication on a physical link, thereby excluding packets sent to other devices. The PHY Service Data Unit is the LL Protocol Data Unit (PDU), and it depends on the type of communication. The Cyclic Redundancy Code (CRC) is used for error detection.

Figure 5.8: BLE PHY Packet. Picture taken from [37].

In a broadcasting packet (Figure 5.9), usually sent by an advertiser, the PDU is divided into a header and the payload. The header indicates the PDU type, Channel Selection (ChSel), Transmitter and Receiver addresses (TxAdd and RxAdd) and the length of the payload. There is also 1 bit reserved for future use (RFU). The payload contains the advertiser's address and the data itself.

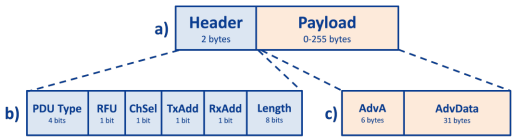

Figure 5.9: BLE LL broadcasting packet. Picture taken from [37].

A connection packet (Figure 5.10) contains a header, the payload and an optional Message Integrity Check code (MIC). The header includes the Logic Link Identifier (LLID, which identifies if the packet contains data or control messages), the Next Expected Sequence Number (NESN), the Sequence Number (SN), More Data (MD), an RFU and, at last, the length of the payload. This length is at most 27 bytes, unless if using the LE Data Packet Length Extension (DLE), introduced in Bluetooth 4.2 specification, that increases this value to 251 (Figure 5.10). Furthermore, with the DLE extension, a new PDU size can be negotiated by either side at any time during a connection. However, this feature is not supported by either iOS or Android devices, so it is not an interesting feature for Kallisto.

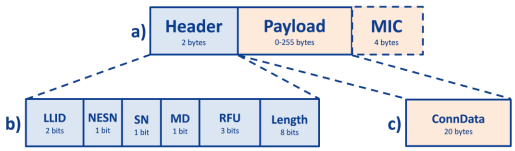

Figure 5.10: BLE LL connection packet. Picture taken from [37].

5.2.4 BLE Network

The simplest BLE network includes a master and a slave, only, and it is called piconet. A device can be connected to multiple piconets, possibly with different roles. A set of connected piconets is called scatternet. A common type of scatternet topology is a mesh network, which requires the nodes to cooperate with each other to efficiently route data.

5.3 Testing the BLE Communication

To test BLE communication we will use an experimental setup composed of a server device, a client device and a BLE sniffer. The server is the nRF52480 microcontroller, on which Kallisto is based and the client is a mobile phone with BLE. The sniffer[1] is a board based on the nRF52832 microcontroller that captures sequences of packets exchanged between the server and the client, which allows analyzing them on Wireshark. To analyze the timings of the transmissions we used timestamps acquired by the BLE sniffer and not by Wireshark. Wireshark runs on a computer attached to the BLE sniffer through a USB dongle, which perverts the Wireshark timings. The timing parameters that we used were the packets transmission time and the time differences between transmissions (delta time).

Our tests focused on the behaviour of the nRF52840 as a server or a peripheral, and not as a client. This is the most common situation, in which the nRF52840 SoC provides data will provide data to a certain client, and it is also how it works in the Kallisto system. The baremetal approach uses the SoftDevice stack, by Nordic, which is provided through precompiled binary files. There are some key features that deserve being mentioned:

1. Bluetooth 5.1 compliant
   (a) Extended Advertising support
   (b) Custom UUID support
   (c) Link layer supporting LE 1M PHY and LE 2M PHY
   (d) LE Data Packet Length Extension
   (e) LE Secure Connections pairing model
   (f) Configurable ATT MTU

[1] infocenter.nordicsemi.com/index.jsp?topic=/ug_sniffer_ble/UG/sniffer_ble/intro.html

2. Memory isolation between the application and the protocol stack for robustness and security
3. Asynchronous, event-driven behavior
4. Any RTOS can be used

Software triggered interrupts in a reserved IRQ are used to signal events from the SoftDevice to the application. The application is then responsible for handling the interrupt and for invoking the relevant SoftDevice functions to obtain the event data.

The biggest advantage to point out is the concern with the memory isolation in the SoftDevice. The SoftDevice and the application have separated FLASH and RAM sections, but share the call stack. Besides this, the SoftDevice reserves the highest interrupt levels for its timing critical operations, hence it will always preempt the application. However, the robustness that this guarantees comes with a disadvantage: the lack of flexibility when using the SoftDevice.

Zephyr has its own BLE stack, that was built through the contributions of several companies and individuals involved in existing open source implementations of the Bluetooth specification (such Linux's BlueZ) as well as the design and development of BLE radio hardware. The main features of Zephyr's BLE stack are:

1. Bluetooth 5.0 compliant
   (a) Custom UUID support
   (b) LE Secure Connections pairing model
   (c) Link layer supporting LE 1M PHY and LE 2M PHY
   (d) LE Data Packet Length Extension
   (e) Highly configurable
   (f) Support for all combinations of Host and Controller builds:
      i. Controller-only (HCI) over UART, SPI, and USB physical transports. This option is used often with Nordic devices.
      ii. Host-only over UART, SPI, and IPM (shared memory)
      iii. Combined (Host + Controller)
2. Asynchronous, event-driven behavior
3. Used frequently with Nordic devices

A particularly interesting feature of Zephyr's BLE stack is that it provides several options for combinations of Host and Controller that the other stacks do not. This feature is not interesting right now for Kallisto, but in the context of IoT it might be an interesting option. Instead of a single-chip configuration, in which a single microcontroller implements all three layers and the application itself (such as in Kallisto), dual-chip configuration is also possible. This configuration uses two separate ICs, one running the Application and the Host, and a second one with the

Controller and the Radio Hardware. For example, we can use the Zephyr Controller with the Linux BLE Host stack (BlueZ) running on any processor capable of supporting Linux. Conversely, combining an IC running the Zephyr Host with an external Controller that does not run Zephyr is also supported.

RIOT does not have its own BLE stack, it uses NimBLE BLE stack developed by Apache as an alternative to the proprietary SoftDevice on Nordic chipsets and thus it is very easily portable to Nordic devices. The main features of the NimBLE stack are:

1. Bluetooth 5.0 compliant
   (a) Extended Advertising support
   (b) Custom UUID support
   (c) Link layer supporting LE 1M PHY and LE 2M PHY
   (d) LE Data Packet Length Extension
   (e) LE Secure Connections pairing model
   (f) Configurable ATT MTU
2. Asynchronous, event-driven behavior
3. Used frequently with Nordic devices

FreeRTOS does not include a BLE stack. As it was explained before, this RTOS consists of the kernel, only. Thus, it was not included in these tests that aimed at comparing BLE stacks.

5.3.1 Scan Request Reception Ratio

Several articles suggest the measurement of the advertising packets reception ratio [11] when evaluating the performance of the implementation of BLE in smartphones. However, in this case we are interested in the performance of the server, so we are interested in the scan request reception ratio. This can be calculated exactly in the same way. It is the ratio between the number of Scan Request packets sent by the client, i.e., the smartphone, and the packets received by the server, which is the SoC nRF52840. This can be easily measured in Wireshark.

This test was performed for an advertising interval of 40 ms.

As it can be seen in Figure 5.11, all the RTOS show a good behaviour, (ratio close to 1), with a difference smaller then 0,025. This leads us to conclude that this feature does not depend on the BLE stack of the RTOS, but on the BLE controller. Note that a low value would mean that several requests were occasionally needed to get a response, thus increasing the time to establish a connection, which is not the case.

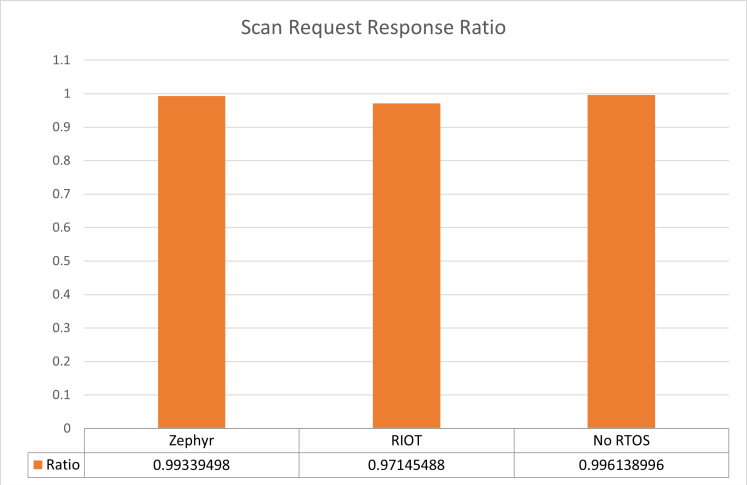

Figure 5.11: Scan Request Reception Ratio measurements.

5.3.2 Maximum Throughput

The maximum throughput is the maximum rate of data that can be delivered over a communication channel. It can be measured either in packets per second or bits per second (bps). In the last case, it is calculated according to Equation 5.1, where D is the amount of data captured by the receiver during a time interval T. D can be obtained by multiplying the number of packets by their size.

$$C = D/T \tag{5.1}$$

Note that, in Data Communications, it is common to differentiate between throughput and goodput. The former considers that D includes the total packet size, thus including the overhead introduced by all the layers in the stack. Conversely, the latter considers that D includes just the application data. In our case that would correspond to the goodput, i.e. the number of useful information bits sent per unit of time. However, in this case, the packet includes the overhead introduced by the other stack layers. In our case, we are interested in the throughput.

In order to measure the throughput, Linhares in [17] suggests the following procedure, where the peripheral node sends a certain number of data packets to the central node. By analysing the packets received, we can calculate the interval between the timestamp from the last and the first packet.

To see the maximum throughput of data that the SoC can send, we have to consider the two forms of data exchange that BLE supports: either the master sends a read command to the slave and the slave answers with data, or we use notifications that are sent by the slave to the master. When reading, the client must send an ACK packet to the server before the server can send the next data packet. BLE specifies 150us as the time interval between consecutive packets called Inter Frame Space. Because the receiver cannot prepare the response within this interval, the response is postponed to the next connection event, significantly reducing the throughput. For this reason, we decided to use notifications in our tests.

The BLE specification also defines the largest Maximum Transfer Unit (MTU) at the application layer as 512 bytes. In reality, the MTU is set to a slightly larger value (for instance, in Android is 517 bytes) to accomodate the ATT header. It also specifies the maximum LL PDU size as 27 bytes. Besides this, most BLE devices define a maximum number of packets that can be exchanged for each connection event in each connection interval, depending on the BLE stack configurations.

Therefore, the throughput will depend on both the maximum data and the connection interval. If the maximum data is low, then for higher throughput we need to use the minimum connection interval. Otherwise, the ratio connEvent/connInterval will be too small with a long idle time. However, using a larger MTU, if the connection interval is too short, the overhead introduced by the protocol stack will be high. For example, consider a situation in which an amount of data corresponding to a connEvent of 25ms has to be transmitted recurrently. Using connInterval > 25ms will allow the data to be sent in a single connEvent. For example, with connInterval=30ms we get a throughput of 25/30 =83%. However, if connInterval=20ms is used instead, we need two connEvents to transfer all the data, leading to a throughput of 25/40 =63%.

On the other hand, BLE specifies a maximum amount of data that can be transferred in a connEvent. Thus, increasing the connInterval beyond this value will monotonically decrease the throughput, but it will also increase the idle time. The work in [17] states that from some point the variation of the connection interval does not have a significant impact on the maximum throughput. That work actually suggests performing the throughput experiments for the following connection interval values: 15, 20, 25, 30, 35, 40, 45, 50, 75, 100, 125, 150, 175, 200, 250, 300, 350, 400, 450, 500 and 1000 milliseconds.

In what concerns the management of the data transfer during the connection event, it is important to realise that a connection event may terminate earlier than expected for different reasons, namely:

1. A packet delivery fails
2. The maximum number of packets were already sent
3. The connection Interval finished

The first situation is related to errors that are uncorrelated with the RTOS in use, thus we will not consider it in our tests.

The second situation happens when the data to be sent does not fit in the configured connEvent. However, when testing with a certain value of connection interval for all the RTOS, the values of throughput will be the same, because it will always correspond to the maximum number of packets configured per connection event. Note that this limitation can come either from the maximum MTU or from the maximum number of packets that the device can send during a connection interval.

Finally, the third situation will occur if, by the end of the connection interval, the device has not been able to send all the packets, yet.

If the maximum application layer MTU (517B) can be completely sent during the minimum connection Interval, then this leads to the maximum throughput achievable. If, otherwise, it takes more than the minimum connection interval to send all the packets, then this corresponds to the third situation. We expect to understand from the tests if the RTOS introduce an overhead that is big enough or limitations to configuration parameters that influence the BLE behaviour. If so, the goal is to determine the maximum throughput achievable when using each RTOS.

However, the value of the Inter-Frame Space is determined by the BLE specification, and it is 150 us. Also the transmission rate is defined, meaning that the time it takes to send a packet of 27 bytes is always the same, and it does not depend on the BLE stack we are using. As we will see in the results, the only thing that changes is how the packets are sent.

The conditions of the test are the following:

- 1 Mbps PHY
- Connection interval of 15 ms, which is the minimum value supported by Android devices
- 27 bytes of data per packet, which is the maximum PDU supported without the DLE
- ATT_MTU is 517, which is the maximum value supported by Android
- Data payload of 512 bytes

Taking into account the value presented before, this results in $512/27 = 19$ L2CAP packets per message. With approximately 100 measures for each case, we measured the time it takes to transfer all packets. The average of the measures is the value T in Equation 5.1, and D is 512 bytes.

The expectations were that the packets could not be all transferred within one connection interval, because the documentation of nRF52840DK clearly states that there is a maximum number of packets that can be exchanged during one connection interval, and this value would be around 6-7 packets. This number would not be enough to allocate 512 bytes of data. However, this did not happen. It was possible in the three cases to transmit all the packets within one connection interval - 15 ms.

This means that in the three cases we can reach the maximum throughput with the conditions established before. Naturally, for higher connection intervals, the same thing would happen: the RTOS would be able to transmit all the data within the connection interval.

There is, however, some difference in the values in Figure 5.12. The differences between the maximum throughputs observed come from the fact that, even though they all send the 512 bytes of data, they do not always send packets with 27 bytes. Both RIOT and Zephyr exhibit lower throughput because they send some smaller L2CAP packets, thus needing more packets to send the whole application information unit.

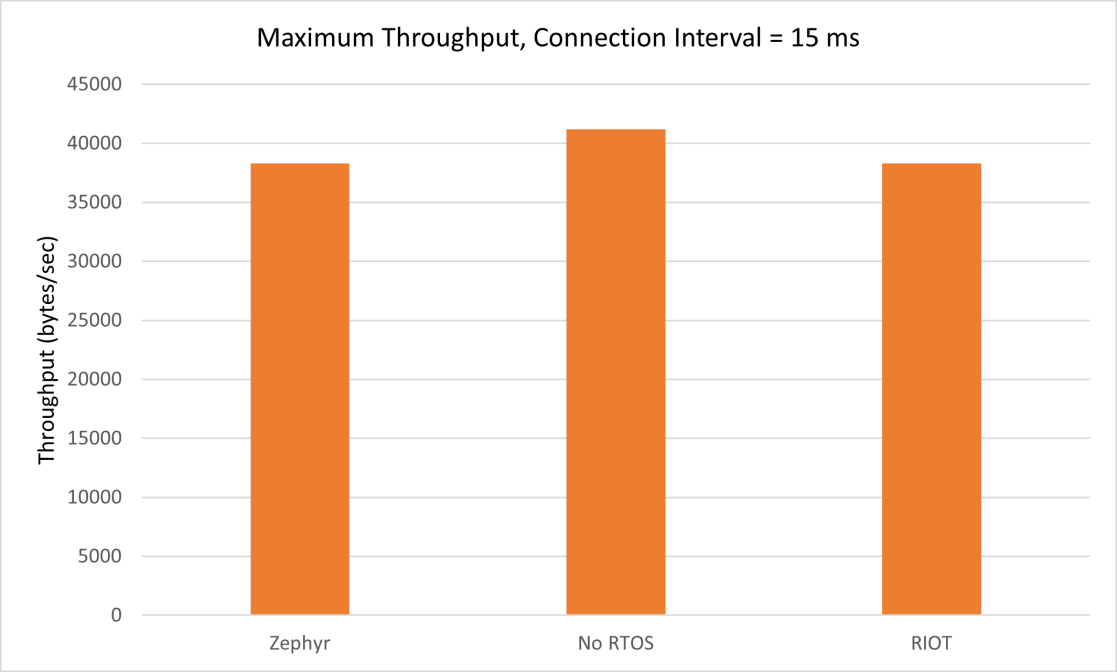

Figure 5.12: Maximum throughput measurements.

5.3.3 Latency

In [13], the average latency for one-way ATT communications and round-trip ATT dialogues between master and slave are both computed as function of connInterval.

The round-trip dialogues that are considered are the following:

- a master executes the read ATT operation and the slave replies with a notification or a command. In this case there is the packet for the read command and then the packet for the response. The test is explained in Figure 5.13.

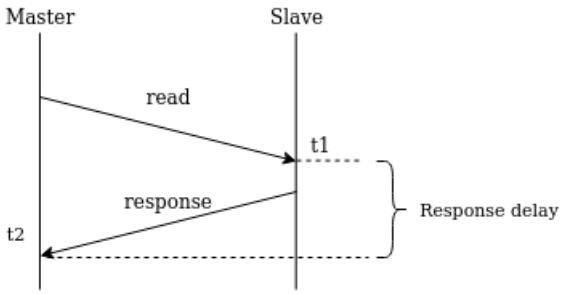

Figure 5.13: Read delay.

- the master sends a notification or a command and the slave acknowledges. This can be done with a simple write command. The test is explained in Figure 5.14.
- the slave sends an indication packet when the value of a specific characteristic changes and when the master receives it, it sends an acknowledgement packet back to the slave.

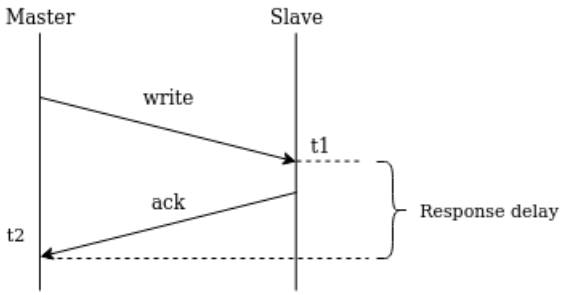

Figure 5.14: Write delay.

The one-way communications that are considered are the following:

- the master sends a notification or a command but there is no acknowledge packet sent back by the slave.
- the slave sends a notification to the master when the value of a specific characteristic changes and the master does not send any acknowledgement packet.

These measurements will always be limited by the connection interval. In the round-trip communications, the second packet (either the answer from the peripheral or the acknowledge to the indication from the master) will always happen in the next connection event. As it was previously explained, a connection event will finish when there are no more packets to be sent, which will be the case. Thus, this latency will most likely be approximately the value of the connection interval, unless the time for the peripheral to process the packet is larger than the connection interval. This test must be performed for short values of connection interval.

The conditions of the test are the following:

- 1 Mbps PHY
- Connection interval of 15 ms and 30 ms (respectively for Figures 5.15 and 5.16)
- 27 bytes of data per packet, which is the maximum PDU supported without the DLE

From observing the results in Figure 5.15 we can see that the latency is relative similar in all cases, corresponding to approximately 15ms (1 connection interval), except for writing with RIOT. In this case, the ACK packet is not returned in the next connection interval, as common with the other experiments, but two intervals later, leading to a latency around 30ms. This difference disappears when using a connection interval of 30ms, prompting us to consider that the previous difference is indeed caused by extra overhead in the RIOT stack.

5.3.4 Connection Event Messages

The measurement of the timestamps of every packet in a connection event is important for characterising the behaviour of the BLE, as this behaviour can potentially be a bottleneck in the applications. To do so, we measured the time offset of the reception of each of the seven packets in

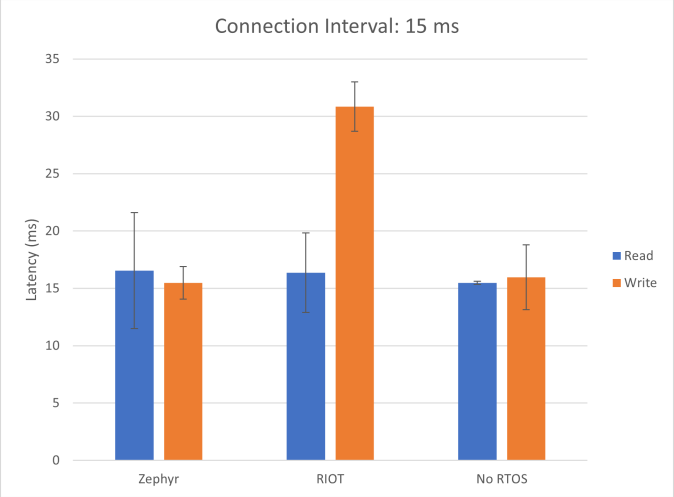

Figure 5.15: Latency for a connection interval of 15 ms.

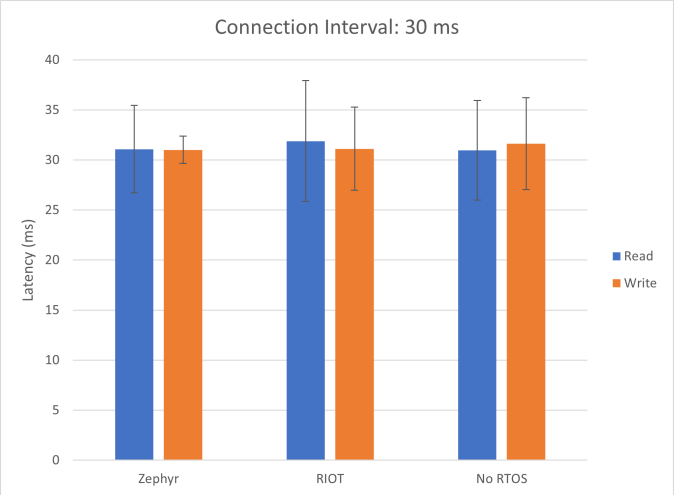

Figure 5.16: Latency for a connection interval of 30 ms.

every connection event, with respect to the beginning of that connection event, i.e. the packet sent by the master. These measurements exhibit predictable values that we can calculate by knowing the transmission time of a packet and the interval between packets. In this test, in some cases we observed a decrease later packets in the connEvent, meaning that there were some errors that caused the connection event to finish before it was supposed to.

The conditions of the test are the following:

- 1 Mbps PHY
- Connection interval of 15 ms, which is the minimum value supported by Android devices
- 27 bytes of data per packet, which is the maximum PDU supported without the DLE
- ATT_MTU is 517, which is the maximum value supported by Android
- Data payload of 512 bytes

The observations when measuring the connection event and their time offsets inside the connection interval are compatible with what we observed when measuring throughput. As we can see in Figure 5.19, in the baremetal case there are less packets transferred. In fact, this is not due to the transmission time of the packets: this will be proportional to the amount of data so the time for transferring 20 bytes or two packets of 10 bytes each should be the same. However, this comes with one more inter-frame space, so we must add 150 us.

It is also curious to verify that in the three cases, there is a change in the behaviour. What happens is not that the number of packets decrease because there was some error that led to the end of the connection event. If we add the small lines next to the main ones, we conclude that some messages are sent a bit earlier. The reason is that instead of the packets being transmitted between $1520-1540\mu$s, and thus fitting in the interval $[1500-1600]\mu$s, they are transmitted between $1590-1610\mu$s and thus some appear in the interval $[1500-1600]\mu$s and others in the interval $[1600-1700]\mu$s. Besides this, when looking at Figures 5.17 and 5.18, we see that something changes at the same time, around the 6.4 ms, both in RIOT and Zephyr. This happens for the same reason that leads to the differences in the connection event duration: around this time, in both RTOS, the packet sent contains less data, so it is sent faster, which will shift the transmission timestamps.

There is another interesting observation: in some cases there are three lines instead of two, which means that there is also a jitter associated. In the case of Figure 5.19, this jitter increases progressively, whilst in RIOT and Zephyr it happens around the 16th and 17th packets (around $[10800-10900]\mu$s.)

Another related measurement is the overall duration of the connection event. Beyond the same information as we just referred (existence of errors), this test also an important information for the applications as we can understand the time in which the radio is active. This is important because it impacts the system power consumption.

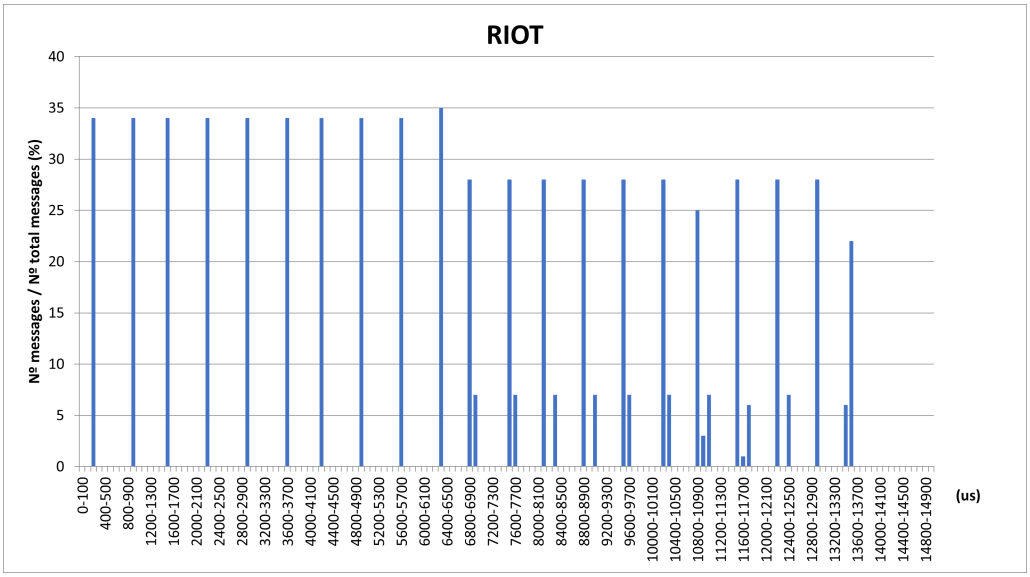

Figure 5.17: Arrival time of the packets in RIOT.

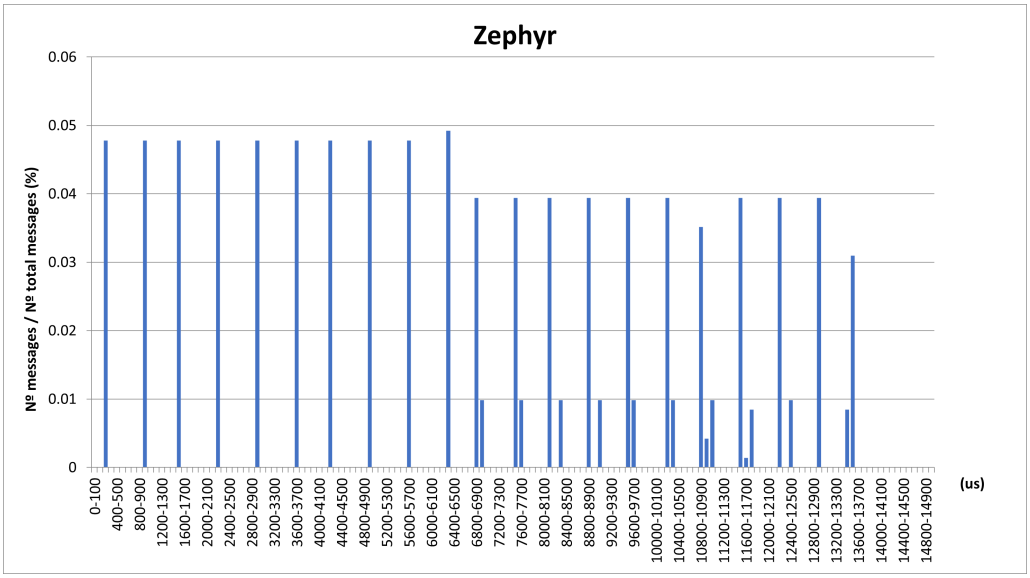

Figure 5.18: Arrival time of the packets in Zephyr.

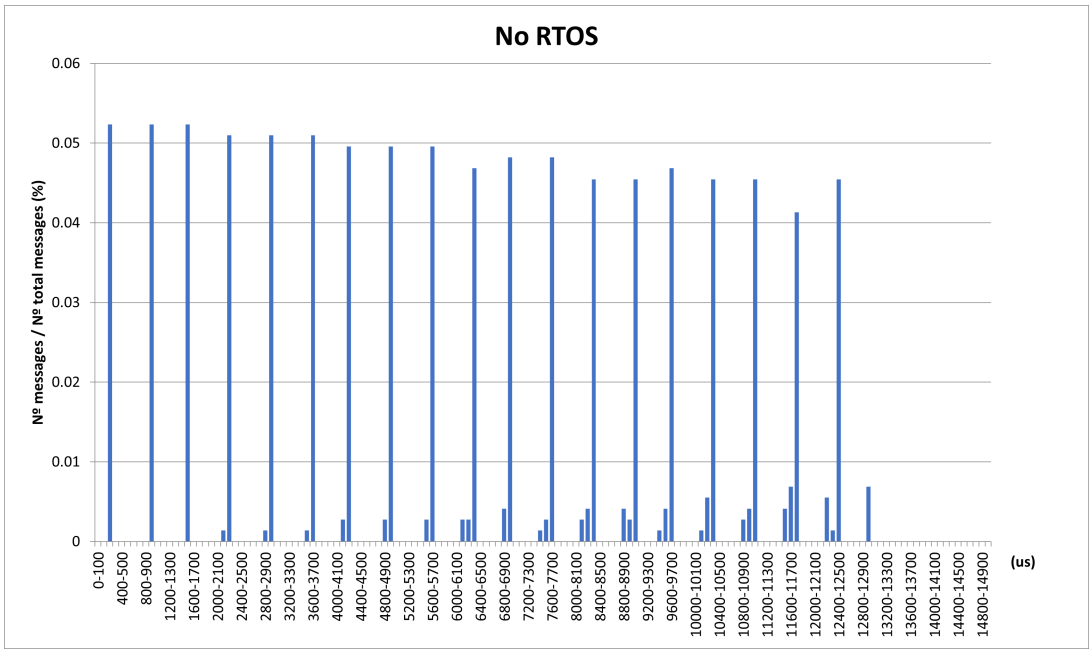

Figure 5.19: Arrival time of the packets in baremetal.

One clear observation from Figure 5.20 is precisely the duration of the connection event in the baremetal implementation, when compared to the other cases, which we already talked about. The variation was omitted because in the three cases the measurements were very precise.

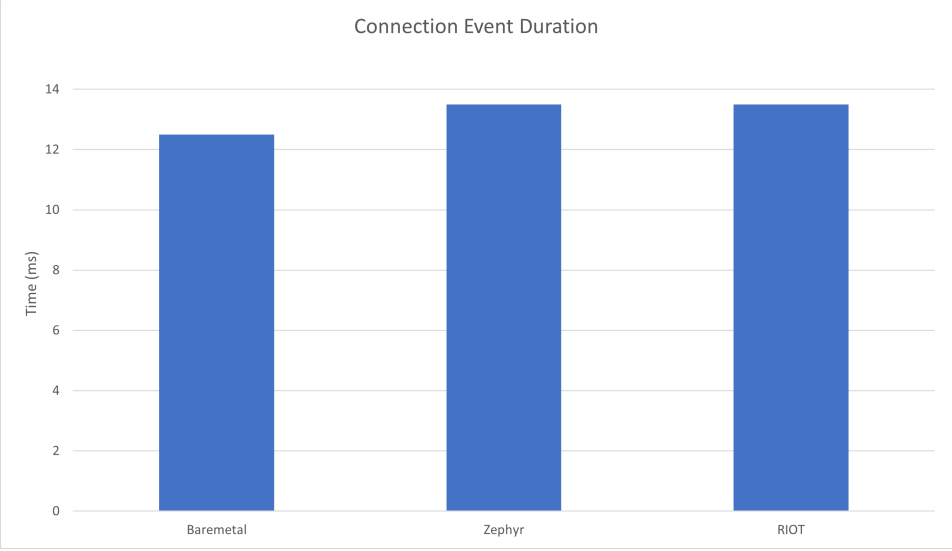

Figure 5.20: Duration of the connection event.

Finally we also measured the actual connection interval and its jitter. This does not depend on the peripheral, but only on the master. Thus, it is not a term of comparison, but an information that might also be relevant for predicting the application time behaviour. However, we can clearly see in figure 5.21 that it does not have a considerable variation.

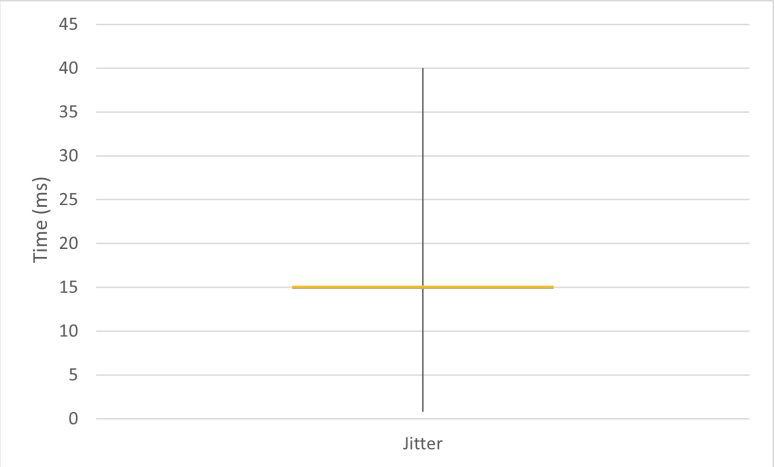

Figure 5.21: Jitter of the connection interval.

5.3.5 Conclusions

In this chapter, we presented the main features and characteristics of BLE, and then we did an experimental characterisation of the different BLE stacks, both for baremetal and the two RTOS that were previously selected (RIOT and Zephyr).

When comparing the BLE stacks, one of the most useful things is the portability to the Nordic devices. In this case, the three stacks were made for the Nordic devices, and this helps a lot in the development of BLE applications. Besides this, the SoftDevice can also be used with other RTOS. They are all compliant with LE 2M PHY and the Data Packet Length Extension, even though this is not important for Kallisto, because it does not use any of these features. Finally, it is important to mention again the memory isolation from the SoftDevice to the applications, that guarantees robustness and security, even though it comes with a memory and performance overhead.

In what concerns the tests, we did not observe significant differences in the results, meaning that the RTOS do not introduce any relevant constraint and the stack performance is essentially determined by the HW controller and by the BLE specifications. Nevertheless, it is particularly interesting the fact that the RTOS do not send always packets with 27 bytes of data, thus having some consequences on the throughput and the connection event duration. Another particular aspect is the behaviour of RIOT when sending an acknowledge to a write command from the master one connection interval later.

Chapter 6

Case Study

The previous chapters presented a comparison of multiple RTOS, their services, their performance and their support to BLE communication. In this last part of the dissertation, the goal is to observe the impact of the RTOS we have chosen in the context of a typical Kallisto application. Besides, it also allows us to get a first assessment of the easiness of development, which one of the main problem of the baremetal implementation. This chapter starts by describing the case study application, then it presents and explains the benchmark with the tests that were performed and their relevance and finally it presents the results of each test.

6.1 Application

The application we used is based on the reference firmware of Kallisto. The target applications of Kallisto are in the scope of Wireless Wearable Technology and wireless connected sensors for healthcare, sport, fitness and industrial applications. This means that its main goal is to collect data from the environment and to send it to through BLE to a central device, which is usually a smartphone. The Android application for the smartphone is called Kallisto Toolbox.

The reference firmware of Kallisto includes plenty of sensors. To simplify the comparison of the operating systems, we decided to focus only in one sensor, the accelerometer. This sensor was chosen because it can reach a higher sampling rate than the others. For the sake of fairness in the comparison, we implemented the application as similarly as possible on the different RTOS as well as on bare metal, particularly in terms of the software architecture.

6.1.1 BMI160 Accelerometer

The accelerometer is the BMI160 by Bosch. Zephyr already offers the driver for the sensor, but RIOT does not, thus it was necessary to write it. In fact, in the Chapter 3, we mentioned that RIOT does not include vendor-specific drivers and it offers less services and sensor drivers than Zephyr.

The BMI160 is a small, low-power inertial measurement unit (IMU) that provides acceleration and angular rate (gyroscopic) measurement. It includes a 16 bit digital, triaxial accelerometer and a 16 bit digital, triaxial gyroscope but the test application uses the accelerometer, only. It includes

a digital primary interface to connect to a host over I2C or SPI. Kallisto uses SPI to communicate with the microcontroller, so the case study application uses the same. Our purpose was to make the three drivers as similar as possible, with similar functions. The main functions are the one for initialising the sensor and the one for acquiring data.

The function for initialising the sensor starts by initialising the SPI interface by setting the chip select pin. Then it must perform a software reset of the sensor and it must read its chip id, while also testing the connection. After, it must set the ODR (Output Data Rate), which is the rate at which a sensor obtains new samples. We set it to the maximum, which is 1600 Hz. At last, it is necessary to enable the accelerometer. The function for acquiring a sample from the sensor is simple, as it just consists of reading the accelerometer data registers of the sensor.

Zephyr offers a hardware abstraction layer with sensor handlers, which can be used instead of calling directly the functions from the sensor API. By analysing its code, it is clear that it does not introduce a large overhead in what comes to performance, because it consists mainly in direct calls to the driver functions. Besides, it helps when developing an application, as there is no need of analysing the driver of the sensor itself. For this reason, we decided to use this layer in the Zephyr application.

6.1.2 BLE profile

The Kallisto BLE profile includes several services: one for each type of sensors and one for the system data (such as battery, firmware version, etc.). Each service includes several characteristics and their respective attributes, which, according to the BLE specification, are *read*, *write* or *notify*.

From those services, our application only implements the ones that are needed for the accelerometer, namely:

- General Access

  This is a standard BLE service that includes three characteristics that can be *read*, only, which are:

  - Device Name, which is the device name in ASCII
  - Peripheral Preferred Connection Parameters, which consists in the connection interval preferred by the peripheral.

- Device Information

  This is a standard BLE service that can be customised. The following characteristics can only be *read*. It includes:

  - Manufacturer Name String, which is the device manufacturer name in ASCII
  - Software Revision String, which is the SW version in ASCII
  - Hardware Revision String, which is the HW board name in ASCII
  - Firmware Revision String, which is the FW version in ASCII

- Motion

  In the Kallisto application, this service includes the characteristics associated to the sensors that are related to motion, such as the accelerometer, the gyroscope, magnetometer, etc. There are three characteristics associated to each of these sensors: the data, the configuration and the calibration.

  In this application, the only sensor is the accelerometer, so there are three characteristics in this service, which are:

  – Accelerometer Data

    Its attributes are *read* and *notify*, because the purpose of the application is to send notifications with the accelerometer data.

    The original Kallisto application sends 2 samples of data per notification. Each sample is 10 bytes long, and it includes the values of the X axis, Y axis and Z axis, as well as the timestamp of the sampling in microseconds. Each of the values are 2 bytes long and the timestamp is a variable of type `uint32_t`, which corresponds to 4 bytes. The characteristic includes two samples, so it is 20 bytes long, organised as in Figure 6.1.

Z axis		Y axis		X axis		Time				Z axis		Y axis		X axis		Time			
D_MSB	D_LSB	D_MSB	D_LSB	D_MSB	D_LSB	T_MSB	T_B3	T_B2	T_LSB	D_MSB	D_LSB	D_MSB	D_LSB	D_MSB	D_LSB	T_MSB	T_B3	T_B2	T_LSB

Figure 6.1: Accelerometer Data Characteristic.

    In our tests we are interested in testing the maximum throughput, which would happen when using the maximum MTU supported by BLE, which is 517 bytes. The maximum number of samples that can be sent with this MTU is 51 samples (510 bytes). This corresponds to 19 LL packets. By using the BLE Sniffer, we can see on Wireshark that the connection events take approximately 13.7 ms, out of the 15 ms that is the duration of the connection interval.

  – Accelerometer Configuration

    This characteristic has the attributes *read*, *write* and *notify*. It consists of one byte, only, encoding the information shown in Fig. 6.2.

Rate Bits									Don't care				Enable bit
R			R			R			X	X	X	X	E
Values	000	001	010	011	100	101	110	111					Write 1 to enable sensor
Rate (Hz)	12.5	12.5	25	50	100	200	400	800					

Figure 6.2: Accelerometer Configuration Characteristic.

6.1.3 Application Structure

The application follows the same guidelines as the Kallisto reference firmware, summarised in Figure 6.3. It relies on the application code, which is the main file. This file also includes the initialisation of both the Hardware Abstraction Layer (HAL) and then of the BLE. Upon initialisation, the application starts advertising. When a connection is established, it propagates the BLE event to the HAL and then it updates the BLE parameters, which consists in setting the connection interval to the minimum supported by Android, 15 ms.

Another file is the Hardware Abstraction Layer (HAL). The purpose of the HAL is to abstract the behaviour of the driver, so the main application does not have to deal with its complexity. The interaction between the entities is showed in Figure 6.4. The HAL includes a function for initialisation, which initialises the driver of the sensor. Then, there is a function for handling the BLE events associated to the services. The event that we are interested in is the write command. When the Android app sends the write command for writing the configuration, the application must parse the BLE packet and handle its configuration. This consists of seeing if the accelerometer is enabled and, if so, what is the rate chosen by the user of the Android app. According to this rate, it enables or disables a timer with the respective frequency. We sample the sensor in the interrupt routine associated to this timer. If the samples buffer is full, which means that 50 samples were already taken, then it sends a notification with the data, as explained before. The Kallisto app for Android supports the rates for data sampling that are shown in Figure 6.2.

However, by using the Android app by Nordic, nRF Connect, we can test the application for higher rates. The application itself has no impact on that behaviour, which depends on the BLE stack of Android, only.

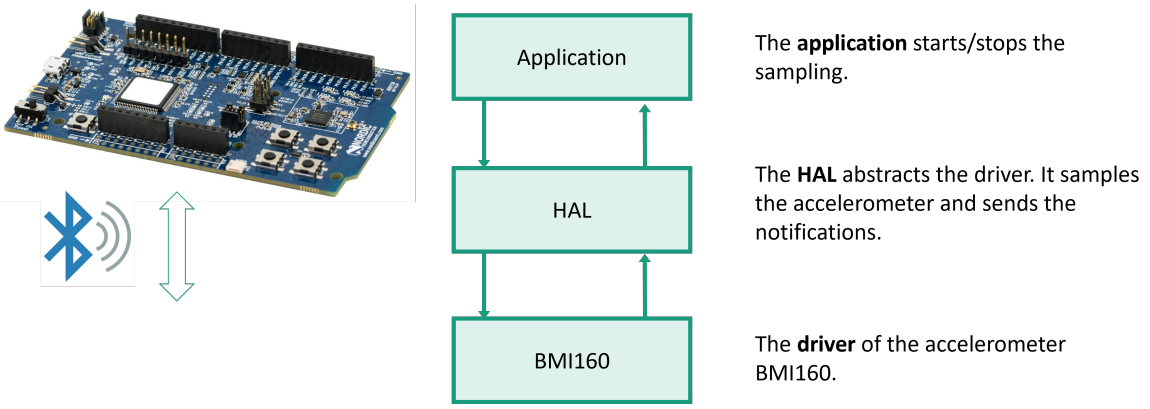

Figure 6.3: Organisation of the project entities.

6.1.4 Optimisation

When comparing the implementations, it is necessary to know their potential in what comes to the optimisation of the code in what comes to energy saving. Usually, this might also be an advantage

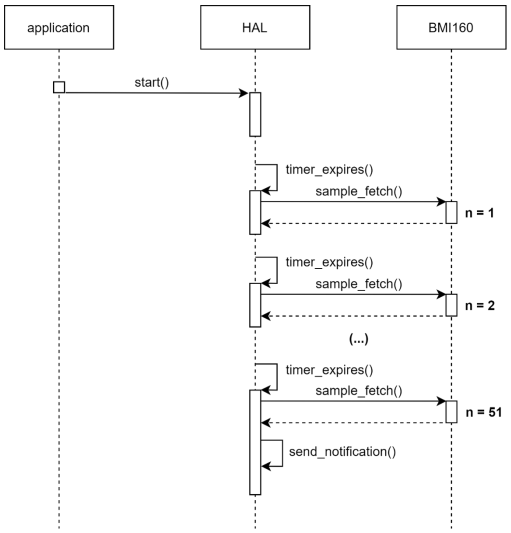

Figure 6.4: Sequence diagram explaining the interaction between application, HAL and accelerometer driver.

when using an operating system: since it may offer some useful services both for energy and memory footprint saving.

One of the first steps in what comes to memory and power saving is to disable the peripherals that are enabled by default, but are not needed. In Zephyr, this can be done by editing the file `proj.conf` and disabling those peripherals. We disabled the I2C, as we do not need it. This can also be done in the Nordic SDK, by editing the file `sdk_config.h`. We disabled the Cryptography library, the UART peripheral driver, the printf module and the logger. Note that disabling these modules and drivers not only reduces the memory footprint, but also the power consumption. Even if a peripheral is not used, the idle state of a specific pin might be "high" or it might be using the clock.

Another aspect that reduces the power consumption is to disable the higher frequency clocks that are not needed. In RIOT and Zephyr this was already implemented, but in the baremetal approach we were initially using the TIMER peripheral driver, which has a maximum frequency of 16 MHz. Then we changed to the RTC peripheral driver, and we disabled the timer. The RTC has a maximum frequency of 32768 Hz, which corresponds to a resolution of $51\mu s$. This timer is used as the trigger for sampling the accelerometer. The maximum frequency reached was around 1600 Hz. This resolution set a trade-off with the power consumption.

Besides the implemented operations, Zephyr offers other services for power saving that were not implemented, but are still important to refer:

1. Tickless Power

2. System Power Management: the kernel enters the idle state when it has nothing to schedule. This idle state is configurable, i.e. it is possible for the developer to specify the power mode, stating which peripherals should sleep and which devices should not. It is an application responsibility to set up a wake up event.

3. Device Power Management. This method is similar to the previous one, except for the fact that the CPU remains active. It works by simply turning off and putting in power saving mode some devices that are not in use. The implementation is more complex, because the components that use these peripherals need to be aware of the power consumption of the device so they know whether they are on or off.

Also in RIOT there are other techniques for power saving, more specifically it includes a module called pm_layered module that introduces a different number of hierarchical power modes that can be blocked or unblocked, and in which a lower power level means less power consumption. If power level N is blocked, then all power levels M <= N are also blocked.

The core design element in the CPU's power management system in RIOT is the so-called idle thread, that will call a function that triggers the CPU to go into a specific power mode. Peripheral drivers or application code can each independently (un)block power modes. For instance, the UART driver will keep the deep sleep power state blocked while the UART peripheral is enabled, and unblock the power state once the peripheral is disabled. Independently, when scheduled, the idle thread switches the CPU to the lowest power mode currently unblocked in pm_layered.

6.2 Benchmark

We performed four experiments to characterise the performance of the three implementations: the maximum rate for sending data, the idle time of the application, the memory footprint and, finally, the power consumption.

6.2.1 Maximum rate for sending data

In this test, the goal is to measure the maximum rate achieved when sending the data from the accelerometer. This measurement is correlated to the previous BLE tests, since the BLE behaviour will always be a bottleneck for the application. Besides this, it is also related to the implementation of the BMI160 driver and the associated overhead in what comes to the timing behaviour of the RTOS itself, measured in Chapter 2.

As it was explained before, the maximum throughput would occur when sending 510 bytes, i.e. 51 samples, each 15 ms, and this corresponds to a throughput of 3400 samples per second, or 34000 bytes/s.

In the baremetal approach, according to the SoftDevice by Nordic, a notification will use the whole application buffer, and once this buffer is full the function for sending notifications (sd_ble_gatts_hvx) will return the error BLE_ERROR_NO_TX_PACKETS. This means that there are no available application packets for this connection. When this happens, the next attempts of sending notifications will not actually update the buffer, overwriting older values that were not sent yet, but instead the values to be sent will simply be discarded and lost.

On the other hand, RIOT, as soon as the buffer is full, starts rewriting the oldest unsent values with the new ones in a circular buffer fashion. Thus, RIOT's approach would be more suitable

for an on-line sampling scenario, in which we would always be interested in the most recent data. This is in fact the most frequent situation when talking about IoT applications, even though there are some Kallisto applications in which it is important not losing data at all.

In Zephyr, once the BLE buffer is full, the function to send the notification will block, waiting until there is free space in the buffer. This means that the interrupt routine associated to the timer will take longer to execute when a higher sampling frequency fills up the buffer. This implementation may result in unbounded delays to send data, which might not be compliant with real-time applications. In fact, Zephyr added the function bt_gatt_notify_cb(), in which a callback is triggered when the notification happens. Note that this does not improve the throughput, and there is no way of flushing the buffer, it simply allows the developer to avoid sending notifications when knowing that the buffer is still full, and therefore avoid the blocking situation.

Taking these different approaches into account, we must define well what we mean by the maximum sampling rate: this would be the rate at which the central device is receiving the current data that is being sampled and without losses, because it is the most frequent scenario in what comes to IoT context. In the baremetal approach, this would be the maximum frequency for which the function does not return any error. After this, data will be lost at least according to the current implementation of Kallisto. In RIOT, this is the maximum rate at which RIOT can sample and send, without overwriting the buffer. Finally, in Zephyr, this is the maximum rate at which it can store in the buffer a new value, without blocking. In this case, some blocking time can be tolerated as long as the blocking time plus the sampling interrupt routine execution time are shorter than the cycle time.

Figure 6.5 shows the maximum rate for each of the three implementations.

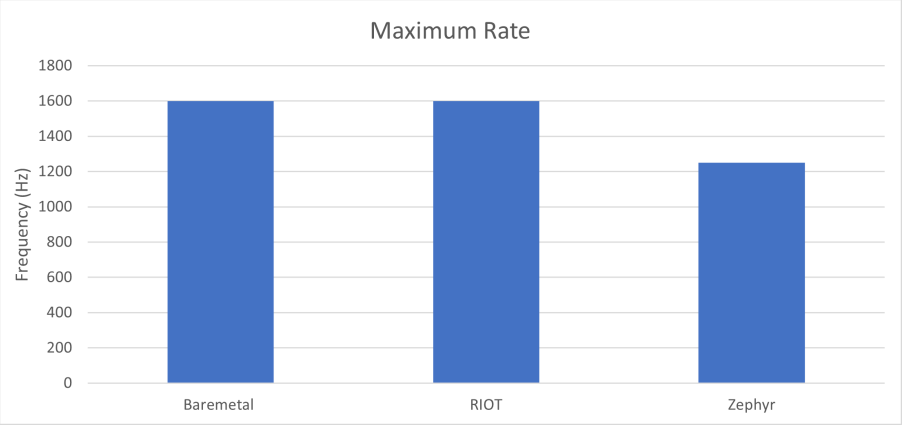

Figure 6.5: Maximum rate for data sampling for the three implementations.

After a certain frequency, the function for sending notification in the Nordic's SoftDevice starts returning an error, which means the buffer is full.

On RIOT, the function for sending notifications returns an error once the buffer is full but, as explained above, it overwrites the old data. However, the maximum rate will actually depend on the execution time of the task that samples the data and the timings associated to context switches and task activation, and not on the fact that the buffer if full.

On Zephyr, it also depends on the execution time of the task, but once the buffer is full this execution time will start to rise. However, at the maximum rate presented, which is 1250 Hz, the buffer was not full yet. What happens is the same as in RIOT: the maximum rate cannot reach higher values because of the execution time of the task and the timings associated to context switches and task activation.

6.2.2 Idle Time

The idle time of the application is an important measurement for two main reasons: firstly, it is deeply correlated to the maximum throughput we can achieve with a certain RTOS, specially when that throughput depends not on the BLE behaviour but on the execution time. Then, it is also related to the power consumption because less idle time will mean more power consumption. This is not straightforward because we are not taking into account the radio activity.

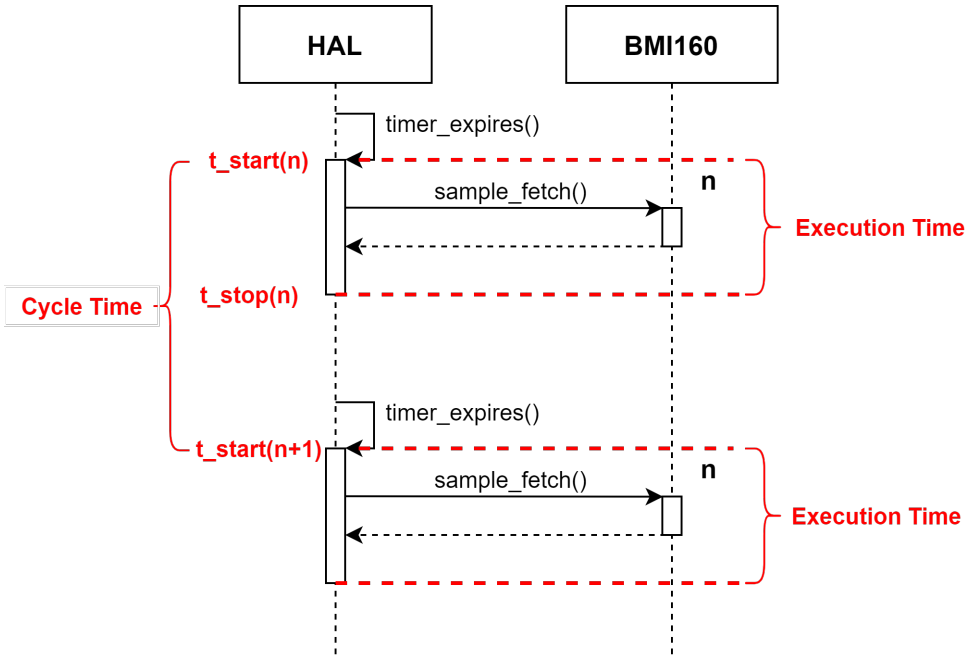

Figure 6.6: Sequence diagram representing the idle time calculation.

As explained in Figure 6.6, the cycle time is calculated by taking a timestamp in the beginning of the interrupt function associated to the sampling timer. The execution time is calculated by doing the difference between start and end time of that function. The start timestamp is taken at the beginning of the interrupt routine and the end timestamp at the end of that same routine. It is important to clarify that this will not actually allow us to calculate the idle time accurately, because it will include the time for context switches and the time for running the routine associated to the timer. However, these timings are very small, as we have seen in the first part of this thesis, and they will not influence the behaviour of this application in specific. The same thing happens for the BLE overhead: the cycle time only goes as far as writing the information in the BLE buffer.

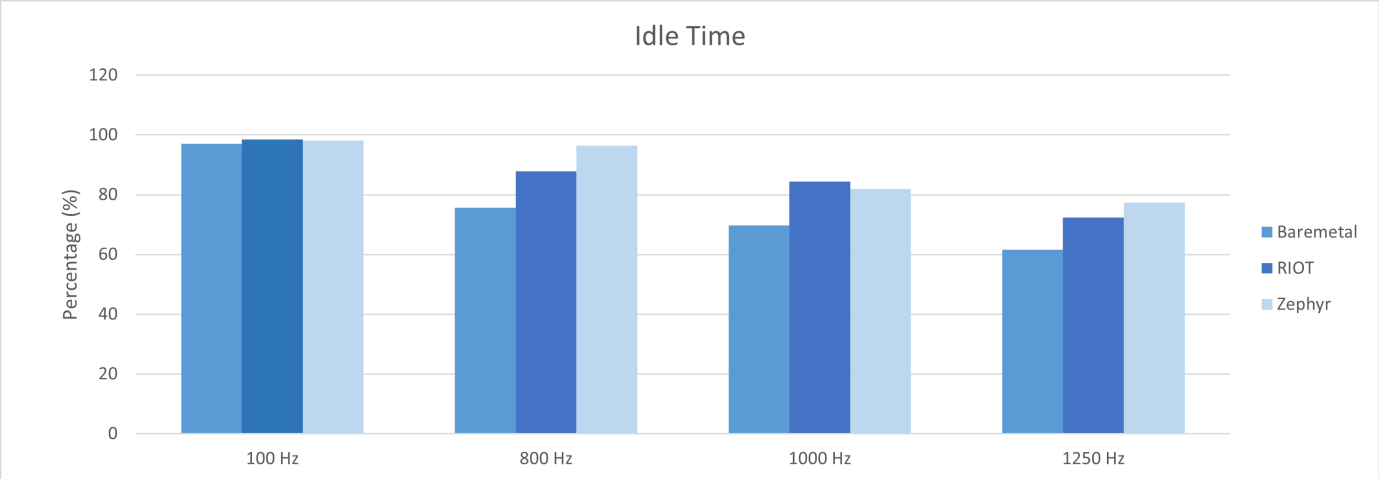

Figure 6.7: Percentage of idle time for different sampling rates and for the three implementations.

As it can be seen on Figure 6.7, the idle time percentage decreases with the increase in frequency, as it would be expected, because the cycle time decreases and the execution time remains the same. One of the observations is that the idle time in Zephyr is not particularly low, which means that the function for notifying is not blocking, and the buffer is not full. However, this is percentage of idle time is not including the time for context switches, task activation and other tasks that might be running in the background. These are responsible for the fact that Zephyr does not reach a higher rate.

6.2.3 Power Consumption

As it was explained in the first part, the power consumption is very important in an embedded system. Thus, it is another important parameter to compare.

The goal is to power the Kallisto board with a stable voltage source and then measure the current. We used a Power Analyser that is able to simultaneously provide a stable voltage source of 4V and measure the current taken by the SoC. It samples and saves the value of the current with a resolution of $0,5\mu$s and 430 times per second.

One of the clearest observations is that Zephyr has a larger power consumption, after RIOT and then the baremetal implementation. These results meet our expectations, when comparing them to the idle time results and thus, to the CPU utilisation.

Another important thing is to know how the power consumption changes when the idle time decreases. It was expected to be higher, and indeed the results show it, but we can see that on Zephyr it is constant, and in RIOT it increases when sampling at higher frequencies.

In what comes to power consumption, the measurements were taken when the board was connected to J-Link. This consumes more current: by measuring it, we understood that the current overhead when debugging it was around 0.8mA. However, this value of current is constant, depending only on the board, so it is correct to compare the values when debugging.

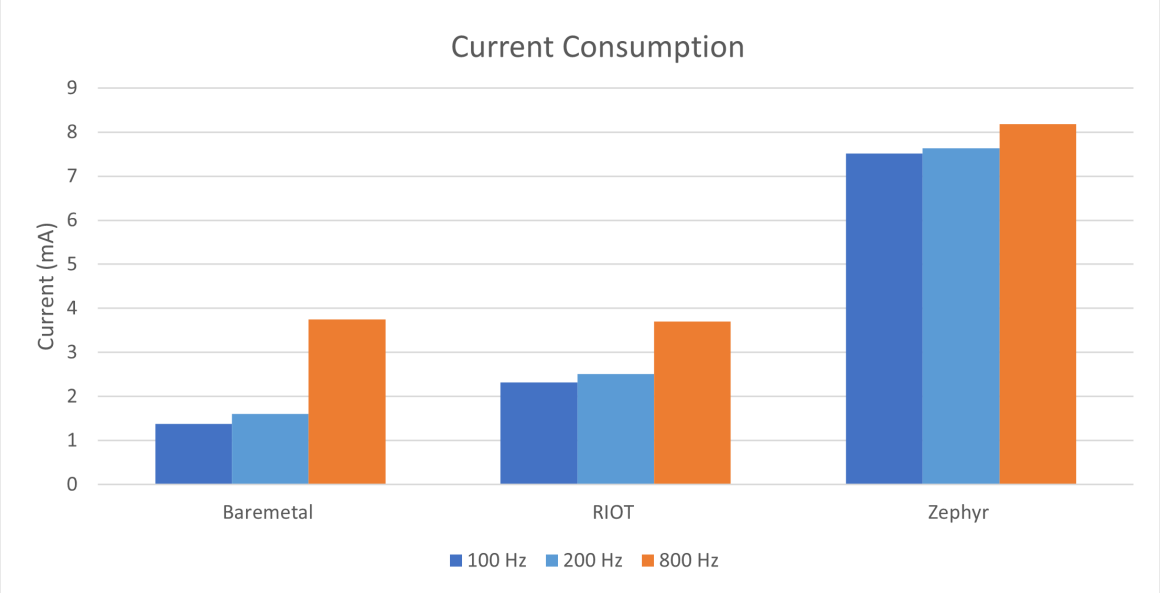

Figure 6.8: Power consumption for the sampling rates of 100 Hz, 200 Hz and 800 Hz.

The values for Zephyr are much higher than the values for the other two implementations. We expected them to be higher, because it is the consequence of the abstraction that Zephyr adds and that also explains the larger memory footprint when compared to RIOT. In similar experiences like [31] they reach the same conclusion: that the power consumption for Zephyr is higher than for RIOT, but the difference is not so accentuated: in the article the difference is about 20%, and in our case it is almost 50%. This might be happening because of other tasks running in the background, or possibly the CPU is not entering the idle state as it would be expected.

6.2.4 Memory Consumption

At last, the memory consumption is also very important in embedded systems.

After building the project for Zephyr and baremetal, we can see the footprint in what comes to Flash and RAM. After building the RIOT project, gcc make provides access to the text, data and bss sizes, which we mapped into Flash and RAM in the following way: ‘text’ is mapped in the Flash, ’data’ is for initialised variables and so it counts both for Flash and RAM and finally ’bss’ is for not initialised data that is mapped in RAM.

The results on the memory footprint might seem odd, because we would expect the baremetal implementation to have a lower memory footprint. In fact, this does not happen because we must include the SoftDevice, which is the BLE stack from Nordic. This stack is not configurable, unlike the operating system’s BLE. In fact, the memory requirements for the flash is 120 kB and for the RAM is, at least, around 6 kB. As to RIOT and Zephyr, this was the expected results: in fact, they make sense when compared to the results for the power consumption too and with the fact that, as it was explained, Zephyr introduces more abstraction than RIOT when dealing with the sensor and the SPI communication.

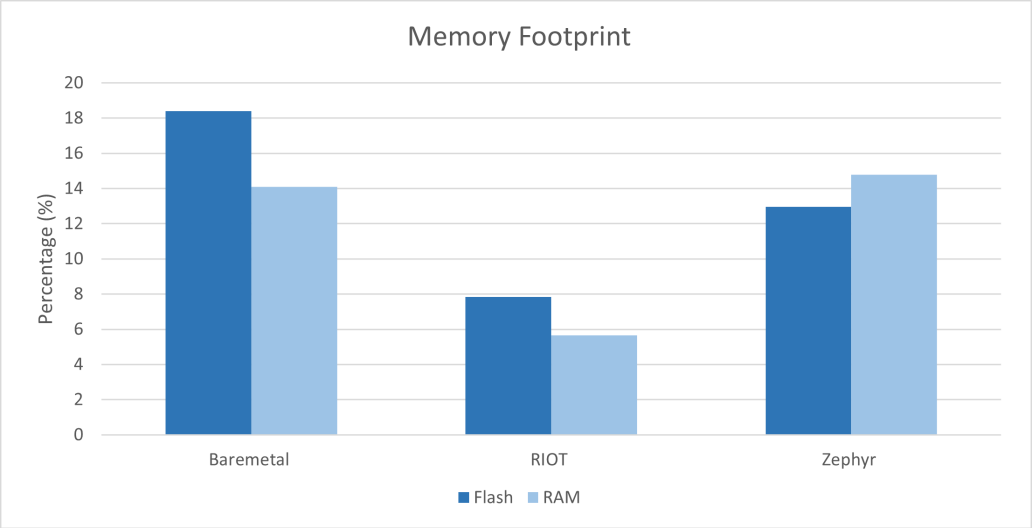

Figure 6.9: Memory footprint in Flash and RAM for the three implementations, as a percentage of the total amount of memory avaibale in nRF52.

6.3 Summary

The results of the baremetal met our expectations: there is very little overhead in what comes to the performance (no drivers or abstractions) and there are no tasks running in the background, which means that the performance was better than with the RTOS. Note that, however, the idle time percentage is lower. This can be explained by the fact that, even though the execution time of the task might be higher (therefore the idle time percentage is lower), we do not have to add to this the impact of context switching or task activation overheads. The same thing happens with the power consumption, as it is clearly lower than with an RTOS. The only drawback is the memory footprint, that is clearly higher than the others. However, this happens because there is a tradeoff between the memory isolation that we want to guarantee and the memory footprint of the stack.

RIOT showed good results too. The maximum throughput achieved was greater than Zephyr's, and its behaviour guarantees the determinism that is expected from an IoT application, by overwriting the old data and delivering always the most recent. The power consumption is considerably smaller than Zephyr, and almost as low as in the baremetal approach. Finally, also the memory footprint is smaller by a significant amount.

Zephyr's overhead comes from the abstraction of the drivers and kernel tasks. The current consumption is higher, even though it stays stable when the idle time decreases, which leads us to conclude that there are some services enabled in the background that are responsible for this current consumption.

Chapter 7

Conclusions

This dissertation aimed at finding whether an embedded RTOS for Kallisto that deals with the complexity required by modern IoT applicaions, including their real-time constrains, could cope with its limited resources, in what comes to memory footprint and energy consumption.

We started by selecting some RTOS based on the characteristics that were more relevant for Kallisto's applications. The three RTOS chosen rely on the multi-threading programming model, meaning that tasks are directly triggered by asynchronous events, and these threads execute concurrently, scheduled by the OS. The alternative of using a time-triggered system is not suitable to manage aperiodic and sporadic events, as it is necessary to poll them and this will increase the delay on serving such events.

There were some features that ended up being common to most RTOS, and that are present in the ones that were selected. In what comes to scheduling, we understood that most RTOS use a preemptive priority based algorithm, and this priority is usually static. Even though it is more deterministic, no RTOS allows to configure a static schedule before execution, because that reduces dramatically the flexibility.

In what comes to memory management, we have verified that virtual memory is not used in small embedded systems, because of its overhead. In what comes to memory allocation, they all support static memory allocation. The chosen RTOS also support dynamic memory allocation, using different algorithms, but they do not guarantee determinism when using them: there is the risk of fragmentation and inefficient memory usage. Static memory allocation introduces some memory overhead due to over-provisioning and results in less flexible systems, while dynamic memory allocation leads to a more complex system and may conflict with real-time requirements.

We will now present the most important conclusions on each of the possible implementations that were taken into account: the usage of FreeRTOS along with the Nordic SDK, Zephyr and RIOT.

7.1 FreeRTOS and Baremetal

The choice for using FreeRTOS comes from mainly two aspects: its maturity and its portability. Its simplicity of use and robustness of the Kernel makes it suitable to be used with diverse software platforms, such as Nordic's SDK. In what comes to Nordic devices in specific, they guarantee support when working with FreeRTOS, but the number of available examples and API documentation is not particularly extensive, even though this support clearly exists in the forums.

FreeRTOS was used along with the nRF5 SDK and SEGGER Embedded Studio (SES) and the availability of an IDE prepared for ARM devices facilitates the development. It includes full debug support including Real Time Terminal (RTT) output and it is easy to use. The SDK configuration header file (sdk_config.h) helps to manage the static configuration of an application that is built on top of nRF5 SDK and although there are some issues with legacy drivers and the overwriting of other files, it is reasonably easy to use.

However, this ends up being a close solution to the current one, a baremetal implementation. Even though it decreases the time and memory management complexity, it will not offer any drivers nor any MCU peripheral abstraction, which means that the developers will still have to worry about these issues. This is less concerning when taking into account the large Nordic community, because many drivers are available, even if they do not follow any API standard. The same thing will happen for the BLE stack: the developers will still have to manage Nordic's SoftDevice as they do currently. This BLE stack, however, is very complete in what comes to the features it offers and the easiness of configuration. The documentation and examples provided are also very helpful, since they cover a considerable amount of different services from the BLE specification that are commonly used in IoT nodes and also by Kallisto. Besides, the memory isolation between the application and the BLE stack is said to be robust and secure, even though it is not managed by the RTOS. Interrupts are used to signal events from the SoftDevice to the application, which then can handle them, but they cannot be directly managed by the application itself. Finally, there is another disadvantage: using FreeRTOS along with nRF SDK does not allow porting Kallisto to another SoC, because the SDK can only be used with Nordic devices.

7.2 RIOT

RIOT is an operating system that successfully guarantees real-time behaviour, by achieving very efficient and well-defined times for context switching and other kernel tasks and guaranteeing determinism in its services (for instance, by only using static memory methods within the kernel or by using a circular buffer on BLE that does not cause any blocking). Its good performance was clear both in Chapter 2, when evaluating the performance of the Kernel services, and in Chapter 6, since it showed a higher maximum throughput than Zephyr.

Its maturity is the main lack: RIOT strives to provide a comprehensive documentation, both on the API level as well as on the architectural level. While the interface and documentation of the code itself has already achieved a good standard (using Doxygen), example code and high-level

descriptions are clearly lacking, and this makes the development much harder. According to Hahm et al. in [14], the kernel can be considered mature, as there haven't been any major reported bugs in the last years. However, other parts of RIOT (e.g., the network stack) are comparably younger and still subject to changes, e.g., as a result of co-evolving with new IoT protocol standards. RIOT is not associated to any IDE or debugger, which clearly makes the development of the code less easy.

Besides this, RIOT clearly offers less services than the other operating systems. This is clear when looking at the variety and amount of inter-task synchronisation and communication services and the number of drivers it offers. In most cases, this will not have any impact apart from the easiness of development, but the abstraction is one of the key points that justify the usage of an operating system. Its lack of abstraction in what comes to these services is one of the reasons why it shows lower memory footprint than Zephyr.

The configuration is done mainly by including additional modules, such as a particular device driver or a system library. It is important to mention that, even though RIOT's community is not particularly active in giving support during the development, and there are not many forums or example applications, it is not particularly difficult to develop the applications for RIOT. However, this might still be a disadvantage when working with some specific technologies or when facing some issues.

In what comes to the BLE stack, it uses Nimble, but RIOT can also be used with Nordic's SoftDevice, even though neither RIOT nor Nordic give direct support. It supports all the features that are currently used, such as the LE 2M PHY and the Data Packet Length Extension, and the parameters are easily configurable, both statically or at runtime.

7.3 Zephyr

Finally, Zephyr is a very mature operating system whose main advantage is the flexibility it provides and the easiness of usage. This is clear in the number of inter-task communication services it offers, and the number of drivers and network stacks. Regarding the scheduling algorithms, it offers other options besides the preemptive priority-based scheduling, with support for dynamic priorities and cooperative tasks. In what comes to memory management, it also offers a dynamic allocation mode that gives the developer more flexibility and control.

The drivers and network stacks are used along with layers of abstraction that facilitate its usage, and the configuration file allows to configure most features in an easy and clear way. The strong community also contributes to this easiness of development: the documentation is complete, well-structured, and clear, and a considerable number of example applications is provided. Also the community is very active, specially the forums, with a good and fast support from Zephyr's developers.

Zephyr offers a large variety of tools. In our case, we used the IDE that was recommended on the Zephyr documentation, which is VSCode, along with PlatformIO. This includes project management tools, editor and debugger. PlatformIO also fetches the source files, so there is no

need of managing them. In what comes to the development itself, an application typically provides a configuration file that specifies application-specific values for one or more kernel configuration options, making it easier to be used.

However, both the flexibility and the abstraction come at the price of runtime performance, power consumption and memory footprint. According to Belleza and De Freitas in [8], the high times associated to Zephyr's services can be explained by its "rich API", which has many layers of abstraction. The overhead on the performance is easily observed when testing its runtime performance, in Chapter 3, and in Chapter 7, in which we can also observe the impact it has on the memory footprint and the power consumption. The power consumption is a relevant issue to analyse in a future work, to understand if it can be reduced.

Regarding the BLE stack, Zephyr's BLE shows the same kind of support in what comes to documentation and examples, and the same easiness of usage. The fact that it provides several options for combinations of Host and Controller is another good example of the flexibility that Zephyr offers.

7.4 Final Considerations

FreeRTOS gives the guarantee of real-time, and its performance is the proof of it, but it is not better than RIOT's behaviour in many aspects. The lack of support for the development of the applications does not pay off for its good performance or the good performance of the SoftDevice, specially when considering that the SoftDevice has a larger memory footprint.

RIOT has a particularly good performance, both in what comes to the Kernel and to the BLE stack Nimble. It is mentioned in several articles for the concern that the developers have in real-time and deterministic behaviour. It also shows a very low memory footprint and a low current consumption, almost as low as the baremetal implementation. Its only disadvantage is that, besides having a great support for the development in Nordic devices, Nordic does not give support for development with RIOT, and the RIOT community is not as active as the Nordic's.

Zephyr is in many ways the opposite of RIOT: the development is easier, even though with less flexibility and a considerable overhead in terms of both memory, power and performance. Zephyr focuses more on guaranteeing flexibility than robustness, even though in terms of run-time performance it was still to guarantee determinism.

Overall, our recommendation is to use RIOT when using a device that is more constrained in what comes to memory resources, or if the application is more demanding in terms of safety and fast-response (besides the determinism). In fact, RIOT supports 8-bit devices, and Zephyr does not. On the other hand, Zephyr appears to be the most suitable choice when the device can easily cope with the overhead, which is the case of the nRF52840, and the application can deal with the differences in the performance, because the support that Nordic and Zephyr give to each other makes the development much easier and faster.

References

[1] (2012). Free RTOS.

[2] (2016). RIOT in a nutshell.

[3] Aroca, R. V. and Caurin, G. (2009). A Real Time Operating Systems (RTOS) Comparison. *WSO - Workshop de Sistemas Operacionais*, page 12.

[4] Baccelli, E., Gundogan, C., Hahm, O., Kietzmann, P., Lenders, M. S., Petersen, H., Schleiser, K., Schmidt, T. C., and Wahlisch, M. (2018). RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE Internet of Things Journal*, 5(6):4428–4440.

[5] Baccelli, E., Hahm, O., Gunes, M., Wahlisch, M., and Schmidt, T. (2014). RIOT OS: Towards an OS for the Internet of Things. pages 79–80.

[6] Barr, M. (2009). Real men program in c.

[7] Barry, R. (2013). Freertos's tick suppression saves power.

[8] Belleza, R. R. and De Freitas, E. P. (2018). Performance study of real-time operating systems for internet of things devices. *IET Software*, 12(3):176–182.

[9] D., J. and M., R. (2012). Real-Time Operating Systems and Programming Languages for Embedded Systems. *Embedded Systems - Theory and Design Methodology*.

[10] Duffy, C., Roedig, U., Herbert, J., and Sreenan, C. J. (2006). A Performance Analysis of MANTIS and TinyOS.

[11] Furst, J., Chen, K., Kim, H.-S., and Bonnet, P. (2018). Evaluating bluetooth low energy for iot. *1st Workshop on Benchmarking Cyber-Physical Networks and Systems*.

[12] Golatowski, F. and Timmermann, D. (1998). Using hartstone uniprocessor benchmark in a real-time systems course. *Proceedings - Real-Time Systems Education III, RTSE 1998*, 1998-November:77–84.

[13] Gomez, C., Oller, J., and Paradells, J. (2012). Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors (Switzerland)*, 12(9):11734–11753.

[14] Hahm, O., Baccelli, E., Petersen, H., and Tsiftes, N. (2016). Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal*, 3(5):720–734.

[15] Hambarde, P., Varma, R., and Jha, S. (2014). The survey of real time operating system: RTOS. *Proceedings - International Conference on Electronic Systems, Signal Processing, and Computing Technologies, ICESC 2014*, pages 34–39.

[16] Harbour, M. G. (2006). Programming real-time systems with C / C ++ and POSIX. pages 1–9.

[17] Linhares, A. G. (2016). Performance measurements and analysis of bluetooth low energy. Master's thesis, Federal University of Pernambuco, Recife.

[18] Loveless, T. (2019). How to choose a real-time operating system.

[19] Mathane, V. and Lakshmi, P. V. (2018). Deterministic Real Time Kernel for Dependable WSN. *2018 4th International Conference for Convergence in Technology, I2CT 2018*, pages 9–12.

[20] Mitra, A. (1974). Getting maximum mileage out of mass media. *Media Asia*, 1(3):14–15.

[21] Mumolo, E., Nolich, M., and Lenac, K. (2008). A real-time embedded kernel for nonvisual robotic sensors. *Eurasip Journal on Embedded Systems*, 2008(1).

[22] Nordic Semiconductors (2019). Nordic nRF52840 Product Specification v.1.1.

[23] Obenland, K. (2000). The use of POSIX in real-time systems, assessing its effectiveness and performance. *The MITRE Corporation*.

[24] Ojo, M. O., Giordano, S., Procissi, G., and Seitanidis, I. N. (2018). A Review of Low-End, Middle-End, and High-End Iot Devices. *IEEE Access*, 6(710583):70528–70554.

[25] Pinto, M. L., Oliveira, S. D., and Aur, M. (2021). Real-Time Performance Evaluation for Robotics An Approach using the Robotstone Benchmark.

[26] Qutqut, M. H., Al-Sakran, A., Almasalha, F., and Hassanein, H. S. (2018). Comprehensive survey of the IoT opensource OSs. *IET Wireless Sensor Systems*, 8(6):323–339.

[27] Sabri, C., Kriaa, L., and Azzouz, S. L. (2018). Comparison of IoT constrained devices operating systems: A survey. *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, 2017-October:369–375.

[28] Scheler, F. and Schroder-preikschat, W. (2010). Time-Triggered vs. Event-Triggered: A matter of configuration?

[29] Shah, V. H. and Shah, A. (2016). An analysis and review on memory management algorithms for real time operating system. *International Journal of Computer Science and Information Security*, 14(5):236.

[30] Silberschatz, A. (2019). *Operating system concepts*. John Wiley &; Sons, Inc.

[31] Silva, M., Cerdeira, D., Pinto, S., and Gomes, T. (2019). Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking. *IEEE Internet of Things Journal*, 6(6):10375–10383.

[32] Silva, P., Almeida, L., and Marau, R. (2010). Temporal behavior of Ethernet communications : Impact of the operating system and protocol stack. *International Conference on Models of Information and Communication Systems*.

[33] Stankovic, J. A., Spuri, M., Ramamritham, K., and Buttazzo, G. C. (2013). *Deadline scheduling for real-time systems: edf and related algorithms*. Springer.

[34] Stewart, D. B. (2002). Measuring Execution Time and Real-Time Performance. *Embedded Systems Conference (ESC SF)*, (September):1–15.

[35] Stewart, K. (2020). Zephyr Project : Unlocking Innovation with an Open Source RTOS.

[36] Tanenbaum, A. and Boschung, H. T. (2018). *Modern operating systems*. Pearson.

[37] Tosi, J., Taffoni, F., Santacatterina, M., Sannino, R., and Formica, D. (2017). Performance evaluation of bluetooth low energy: A systematic review. *Sensors (Switzerland)*, 17(12):1–34.

[38] Townsend, K., Carles, C., Akiba, and Davidson, R. (2015). *Getting started with Bluetooth low energy*. SPD.

[39] Ungurean, I. (2020). Timing Comparison of the Real-Time Operating.

[40] Weiderman, N. H. and Kamenoff, N. I. (1992). Hartstone Uniprocessor Benchmark: Definitions and experiments for real-time systems. *Real-Time Systems*, 4(4):353–382.

[41] Welsh, M., Culler, D., and Brewer, E. (2001). SEDA: An architecture for well-conditioned, scalable internet services. *Operating Systems Review (ACM)*, 35(5):230–243.