UCLouvain

École polytechnique de Louvain

Benchmarking Framework for Real-Time Operating System Applications

Study and implementation with Contiki and RIOT

Authors: **Julien Gomez, Trong-Vu Tran**
Supervisor: **Ramin Sadre**
Readers: **Lionel Metongnon, Ludovic Moreau**
Academic year 2018–2019
Master [120] in Computer Science

Abstract

The purpose of this thesis is to assess the possibility to create a benchmarking framework aimed at real-time operating systems running on constrained devices. This benchmarking framework will be used to evaluate the performances of an application on top of different RTOS. The performance metric used is the context switching time.

To do so, we started with a theoretical analysis of real-time operating systems. This allows us to formalize the concepts needed for the rest of the thesis. After that, we compared different RTOS in terms of features and characteristics. This comparison helps in understanding and explaining the operations we performed and the results we obtained. For the development of the benchmarking framework, we explored three different approaches.

Our results showed that using the internal real-time clock to compute the context switching time is not suited for the benchmarking framework. Instead, the use of an external board such as the Pocket Science Lab is the best option to retrieve precise measurements.

Future improvements include the use of a cache inside the RTOS to retrieve interesting data such as memory usage or task utilization. Moreover, the external board can be used to monitor the power consumption of the benchmarked board.

Acknowledgements

We would like to express our deepest gratitude to Professor Ramin Sadre, our supervisor which oversaw our work for more than a year. Without his guidance, help and logistical support this thesis would not have been possible.

We would like to thank Lionel Metongnon and Ludovic Moreau for accepting the difficult and time-consuming task to read the present document. We would like to thank Ludovic Moreau a second time for reviewing our thesis before our submission.

In order for us to write this thesis, many other people helped us. We would like to thank Pascal Simon, Aditi Hilbert, Dr. Emmanuel Baccelli, Prof. Thomas C. Schmidt, Daniele Lacamera, Gilles Doffe, Rahul Arvind Jadhav, Rodrigo Peixoto, Vicente J.P. Amorim, Jukka Rissanen, Victor Hansen and the communities from RIOT, Contiki, FreeRTOS, Apache Mynewt and mBed.

For their support, we would like to thank our fellow friends: Arnaud Gellens, Robin Hormaux, Romain Hubert, Charline Outters, Kevin Stoffler, Corentin Surquin and Carolina Unriza Salamanca.

A special thanks to our friend Antoine Van Grootenbrulle whom we wish was here to witness our work. You will be missed dearly.

Julien Gomez *I would like to thank Manon Biver and my parents for their strong support through this adventure and Trong-Vu Tran for being the perfect thesis partner and without whom it would have not been possible to realize this work.*

Trong-Vu Tran *I would like to thank my parents and my lifelong friend Corentin Surquin for their indefectible faith and support and also Julien Gomez for bearing with me through this journey.*

Content

I Comparing RTOS from a theoretical point of view 4

1 Theory of RTOS 5
- 1.1 System architecture . . . 5
  - 1.1.1 Kernel . . . 5
  - 1.1.2 User space and kernel space . . . 6
  - 1.1.3 CPU modes . . . 6
  - 1.1.4 Interprocess communication . . . 7
  - 1.1.5 Interrupt . . . 7
  - 1.1.6 Kernel architecture . . . 7
- 1.2 Scheduling . . . 9
  - 1.2.1 Base concepts . . . 9
  - 1.2.2 Scheduling policies . . . 10
- 1.3 Memory management . . . 10
  - 1.3.1 Static memory management . . . 11
  - 1.3.2 Dynamic memory management . . . 11
- 1.4 Power management . . . 13
  - 1.4.1 Hardware power management . . . 13
  - 1.4.2 Operating system power management . . . 14
- 1.5 Programming model . . . 15
  - 1.5.1 Event-driven model . . . 15
  - 1.5.2 Multithreaded model . . . 15
- 1.6 Hardware support . . . 15
  - 1.6.1 Constrained device classes . . . 16
  - 1.6.2 Hardware abstraction layer . . . 17

2 RTOS outline 18
- 2.1 RIOT . . . 18
  - 2.1.1 Historic . . . 18
  - 2.1.2 Characteristics and features . . . 19
  - 2.1.3 Specificities . . . 19
- 2.2 Contiki . . . 20

2.2.1 Historic . . . 20
2.2.2 Characteristics and features . . . 20
2.2.3 Specificities . . . 21
2.3 FreeRTOS . . . 22
2.3.1 Historic . . . 22
2.3.2 Characteristics and features . . . 22
2.4 Comparison table . . . 24

II Implementing a benchmarking framework in the context of a RTOS 25

1 Objective 26
1.1 Implementing a benchmarking framework . . . 26
1.1.1 Motivations . . . 26
1.1.2 Criteria . . . 27
1.1.3 Approaches . . . 28
1.2 Narrowed objective and limitations . . . 28
1.2.1 Context-switching time centric framework . . . 29
1.2.2 Scheduler limitations . . . 29
1.3 Reference value . . . 29
1.3.1 Simple application . . . 30
1.3.2 Methodology . . . 30
1.3.3 Measurements setup . . . 32
1.3.4 Measurements results . . . 34

2 Experiment 39
2.1 Integrating the framework in the kernel of the RTOS . . . 40
2.1.1 Motivations . . . 40
2.1.2 Limitations . . . 40
2.2 Implementing the framework as an extension of the RTOS . . . 41
2.2.1 Definition of RTOS extension . . . 41
2.2.2 Framework utilization . . . 41
2.2.3 Using the framework with our simple application . . . 42
2.2.4 Framework implementation . . . 43
2.3 Using external devices . . . 45
2.3.1 Choice of the external device . . . 45
2.3.2 Role of the different devices . . . 47
2.3.3 Usage of the Pocket Science Lab . . . 48
2.3.4 Using the framework with our simple application . . . 48
2.3.5 Framework implementation . . . 49

3 Results and framework measurements **52**
3.1 Overhead . . . 52
3.1.1 Extension approach overhead . . . 52
3.1.2 Devices approach overhead . . . 53
3.2 Framework measurements . . . 53
3.2.1 Extension approach measurements . . . 53
3.2.2 Devices approach measurement . . . 55
3.2.3 Summary . . . 57

4 Discussion and comparison of the approaches **60**
4.1 Extension approach overhead issue . . . 60
4.1.1 Causes . . . 60
4.1.2 Solutions and improvements . . . 61
4.1.3 Overhead of the devices approach . . . 61
4.2 Assessment of the different approaches . . . 61
4.2.1 Extension approach performance . . . 61
4.2.2 Devices approach performance . . . 62
4.3 Contiki measurements distribution . . . 64
4.4 Summary . . . 65

Introduction

Context

Embedded Systems, Real-Time Operating Systems and Internet of Things

Embedded Systems have been around for a while now. One of the first well-known modern embedded systems was the Apollo Guidance Computer (AGC) designed in the 60's which contributed to the monumental success of sending a man on the Moon. By its design, the software that was developed for it would become a precursor of what is now known as Real-Time Operating Systems. Although not necessarily, Real-Time Operating Systems (RTOS) are well often used inside embedded systems and countless examples exist to prove it. We already mentioned the aerospace industry which is certainly in high demand for embedded systems, but we can also mention the automotive, medical or military industries. From a more consumer point of view we can mention home automation, digital cameras or even washing machines.

In the past two decades, embedded devices benefited from the explosion of the Internet which brought them the ability to communicate and interact with each other and more globally with anything connected to the Internet. The age of the Internet of Things (IoT) as it is called brought countless of new applications for embedded systems. Billions of those connected devices are now in use as of today with no sign of getting out of the trend.

Benchmarking of Real-Time Operating Systems

A benchmark is a tool designed to assess the performances of a system. In computer science, softwares are used to benchmark the performances of computer hardware but also softwares. Compilers or database management systems (dbms) are a good example; many tools are available to benchmark them such as HammerDB.

Preliminary research we made for this thesis revealed that solutions already exist in order to benchmark real-time operating systems. We can cite `MiBench`[17], an embedded benchmark suite and also `ParMiBench`[20], its equivalent for multiprocessor systems.

We then came up with the idea of building a benchmarking tool not to benchmark the operating systems but the applications built on top of them.

RIOT Summit 2018

In September 2018, we went to the RIOT Summit 2018 in Amsterdam in order to meet developers and researchers specialized in the RTOS area.

This was the third summit of the RIOT community. Every year the members of the community gather and talk about their projects and the future work for RIOT. The summit was divided into two days. During the first day, 12 speakers presented their work. On the second day, tutorials were given and breakout groups ended the summit.

By talking to developers that were present at the summit, we learned that the STM32F4 series microcontroller is a good choice to perform a benchmark on RTOS. With its ARM Cortex-M4 based MCU, it compiles a large variety of RTOS.

Additionally, we discussed our idea to use a logical analyzer to perform time analysis. Gilles Doffe from Savoir-faire Linux confirmed our opinion about using this kind of devices for our framework.

During this summit, we discovered a large number of applications of RIOT and RTOS in general. We talked to some of the maintainers of RIOT and other developers. With their expertise and their advice, we got references to hardwares and softwares that could be useful for our work.

Objective

The objective of the present work is dual.

First, we wanted to gather information in order to present a theoretical state of the art about RTOS. The literature of RTOS is really sparse and technical. We wanted to summarize and give an introduction of what makes the specificities of real-time operating systems.

Next, we implemented a proof-of-concept and explored multiple ways to build a benchmarking tool targeting RTOS applications.

We did not want to build a new benchmarking tool for RTOS as multiple tools are already available. We then developed the idea for the RTOS application benchmarking framework and discussed it with the RIOT community at the RIOT Summit. The feedback was positive and many found the idea really interesting to explore. We believe that a complete benchmarking tool for RTOS applications can benefit the industry and help it in designing better fitted applications in constrained environments.

Outline

The present document is divided in two parts.

Chapter 1 of the first part is a summary of RTOS theory. It provides information needed in order to understand the second part. It can also be read independently from the rest of the document and gives a good glimpse at RTOS design.

Chapter 2 provides a comparison of different RTOS from a theoretical point of view. We chose to study and analyze RIOT, Contiki and FreeRTOS. They give a good example of a real implementation of the theory developed in the first chapter. We also developed our proof-of-concept framework for these OS, so they are then presented beforehand.

Chapter 1 of the second part presents the different objectives we followed during our development and research. The objectives shifted as we progressed and limitations were discovered. This chapter also describes the methodology we followed to get reference measurements to compare our framework.

Chapter 2 describes the experiments we performed. We tried different approaches and explained their operations.

Chapter 3 and chapter 4 present and discuss our results. We performed comparisons with our different approaches and the reference measurements we obtained in order to determine the best fit for our framework.

Part I

Comparing RTOS from a theoretical point of view

Chapter 1

Theory of RTOS

In order to understand how a real-time operating system (RTOS) works compared to a general purpose operating system (OS), it is essential to define some recurrent characteristics we would expect to find in a real-time architecture.
This chapter is non-exhaustive and we could debate about the importance of some characteristics compared to some others. We decided to choose those characteristics because they are commonly used in the literature and we think that they will provide a good glance at what we can expect from an RTOS.

The first part of this thesis is the result of an effort from ours to summarize the knowledge we collected during our research. We spent countless hours collecting data, reading papers, documentation and technical datasheets from vendors, researchers and developers. We also spent time contacting qualified people to review, read and advise us on very specific topics.

1.1 System architecture

Operating systems for embedded devices started appearing in the 00's. Since then design choices have evolved with the technology, research and trends.
In the subsection 1.1.6, we will explain the different kernel architectures commonly found in RTOS and their impact on the operating system. But beforehand, we will introduce some concepts that will help explain the kernel architectures.

1.1.1 Kernel

The kernel is the center piece of an operating system. The role of the kernel is to manage every critical function of an operating system. The kernel could technically run by itself but it would not be very practical. An operating system is built around its kernel and in the case of an RTOS, kernel design is critical for ensuring

strict deadlines. The tasks a kernel usually manages consist of:

- Memory management
- Process and CPU scheduler
- Device drivers
- System calls

1.1.2 User space and kernel space

Almost every operating system makes the distinction between user space and kernel space. The memory allowed to the operating system is divided into two distinct segments. The goal is to provide memory and hardware protection against malicious or faulty behaviors.

The first one, the kernel space, is dedicated to operations the most related to the hardware. It is reserved for the most sensitive operations of the operating system. Due to its central position in the operating system, any bug in the kernel can lead to system failure. In order for a user process to use the hardware, it must ask permission from the kernel through an API so that the kernel performs the operation for the process.

The second one, the user space is dedicated to all the code running outside of the kernel space such as applications, programs and libraries. Each user-space process has its own memory space and cannot access the memory of another process (unless they use shared memory)[1].

In RTOS, the boundary between user space and kernel space may be only symbolic. Many of them (Contiki, RIOT, freeRTOS...) can run without memory management unit (MMU) which is responsible for the memory protection and therefore the separation between user and kernel spaces. An application could then potentially access kernel space even when it is not supposed to.

1.1.3 CPU modes

CPU run on different operating modes (also called states or levels). In kernel mode (also called privilege mode or unrestricted mode), the CPU may perform any operation without restriction. In user mode, direct access to hardware is prohibited.

Those modes prevent the application software from accessing the hardware directly. Applications are restricted to their address space and should use the operating system's abstract services in order to perform operations on the hardware[31].

1.1.4 Interprocess communication

Interprocess communication (IPC) designates the mechanisms used to share data between processes in an operating system. Those mechanisms can differ from OS to OS but are generally the same. We can notably cite[11]:

- Pipes
- Names pipes
- Message queuing
- Semaphores
- Shared memory
- Sockets

1.1.5 Interrupt

An interrupt is a signal emitted by hardware or software to the CPU signaling an event. When an interrupt happens, the state of the current program is saved and the routine related to the interrupt is executed.

The origin of an interrupt can be of all sorts, from changing running thread in a single core processor to pressing a key of a keyboard.

1.1.6 Kernel architecture

Monolithic architecture

A monolithic kernel is composed of a single block of code running a single large process. Thanks to its simplicity, fast execution time and low memory footprint, it served as the norm for early RTOS. In monolithic systems, the operating system runs as a whole in privilege mode. The application built upon it requests services by using *system call* instructions.

Interrupt handling is performed directly in the kernel for the most part and interrupt handlers are not full-fledged processes. Consequently, the interrupt handling overhead is very small because there is no full task switching when triggered. Nonetheless, the handling code cannot invoke most system services like blocking synchronization primitives. Instead of that, the scheduler is disabled during the Interrupt service routine (ISR) and only hardware prioritization is in effect. Hence, the ISR is implicitly executed at higher priority than all the other tasks of the system.

The monolithic kernel architecture is comparatively quite fast, since everything is implemented under the same address space. Nonetheless, due to its design, it is prompt to critical failures, difficult to understand, maintain and update[15][29].

Microkernel architecture

In a microkernel architecture, the kernel is broken down into separate processes; the `microkernel`, a minimalistic kernel and the `servers`, extending the functionalities of the said microkernel[29].

The servers functionalities are often features of the kernel that can run in the user space rather than the kernel space (such as the file system, network features or device access)[24]. A message-passing communication mechanism is used to communicate between the kernel and the servers. The main purpose of the microkernel is to handle the communication between the application and the servers and to perform critical operations, such as accessing input/output (I/O) device registers, that would be difficult or inefficient to perform in user space.

Interrupt requests are handled by transforming them into messages to the appropriate handling task. The interrupt handler runs in interrupt service mode and performs the work required by the hardware, then sends a message to an interrupt service task. Interrupt service tasks operate like any other task, including the blocking primitives. The overhead of interrupt handling is higher than with the monolithic architecture since it implies a full task switch.

A microkernel architecture is considered more secure, as kernel-space functionalities and user-space functionalities are dissociated. It is also more reliable and resilient as if a server crashes, it does not stop the microkernel from running nor the other servers. The memory footprint can also be minimized since we can choose which server we want to use and only boot with those. From a maintainer point of view, it is also less complex, easier to understand and update[15][29]. The microkernel architecture needs an IPC. A message-passing mechanism must be used and is inheritantly slower than a direct function calls.

Of course there are more architectures than what is listed above. But in the context of this thesis, we considered that explaining each one in detail would not be relevant. The two designs presented are the most common and provide a good preview of what a kernel looks like. Also, the following parts of this thesis will expand on this and we will see in practice what each RTOS uses from the theory.

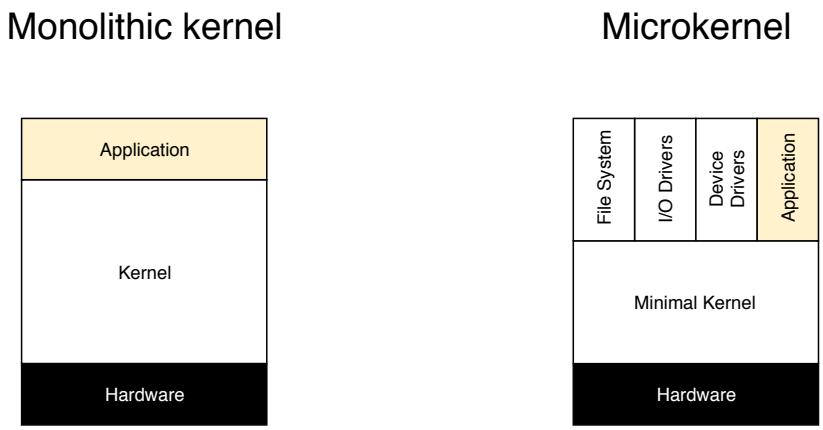

Figure 1.1: different kernel designs

1.2 Scheduling

The scheduler plays an important role in the design of a(n) (RT)OS. The performances of an OS are deeply impacted by the scheduling algorithm it uses. As we will see further in the thesis, it can also affect the energy consumption of the device.

1.2.1 Base concepts

A scheduler is a process designed to choose which task will run at a certain time in the system. Different strategies can be used, each one with multiple advantages and drawbacks. Some commonly used are presented below in the Subsection 1.2.2.

A scheduler is said to be *preemptive* if tasks in the system can preempt each other. When a higher-priority task wants to execute, the scheduler can interrupt a lower-priority task and run the higher-priority one. On the other hand, a scheduler is said *non-preemptive* or *cooperative* if a task that has been allowed to start will execute until it is complete.

Another distinction we can make is to divide schedulers into *online* and *offline* schedulers. Online schedulers decide of the ordering of the different tasks during runtime based on various parameters (such as task priority for example). A scheduler based on task priority is also called *priority-based* scheduler. On the contrary, offline schedulers (also known as *table-driven* schedulers) perform their scheduling decisions at the start of the system.

1.2.2 Scheduling policies

First in, first out (FIFO)

Also known as first come, first serve (FCFS), FIFO is one of the simplest scheduling algorithms. Processes are queued into a data structure in their order of arrival. The first process to be enqueued is the first that will be executed.

Round-robin

The round-robin scheduling algorithm divides the time allocated to each process into fixed time slices. If a process does not terminate when his allocated time slice is expired, the scheduler switches to another task. If a task terminates within its time slice, the scheduler simply switches to the next task.

Earliest deadline first (EDF)

Earliest deadline first scheduling algorithm dynamically assigns a priority to each enqueued process (based on its deadline or an estimation of it) into a priority queue data structure. The scheduler then executes the process the closest to its deadline at each scheduling event.

Fixed-priority preemptive scheduling

Priority for each task is pre-assigned by the operating system. The scheduler arranges the tasks by order of priority. Higher-priority tasks can interrupt lower-priority tasks[19].

1.3 Memory management

In modern computer systems, memory management has evolved since early days techniques which were limited by early computer systems where each memory location was specified in the program. This led to critical errors and/or unpredictability when an incorrect location was specified.

Nowadays, the memory management of (almost) every computer system follows the same principle. The memory of a computer system can be divided into 5 distinct sections[25]:

- The `text` segment or code segment consists of the compiled program code. It is read-only and initialized from the executable file.
- The `data` segment contains the global and static variables, allocated and initialized before the start of the program.

- The `bss` segment contains all global and static variables that are not initialized.
- The `heap` is used for the dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
- The `stack` is used for local variables. Space on the stack is reserved for local variables when they are declared.

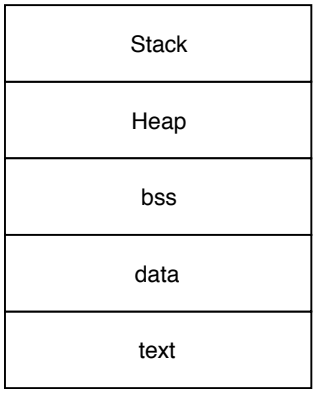

Figure 1.2: memory organization

1.3.1 Static memory management

By the time a program begins to execute, there must be some specific blocks of memory reserved for its use. This includes, for instance, the memory containing the program's own code. Moreover, every static variable must have a specific memory reserved.

The static memory allocation is predetermined by the compiler and will always be reserved for the program in the same manner at the beginning of every run.

This part of the memory operates as a *stack* or last-in first-out (LIFO) queue. The area of memory available for the use of the program will shrink and grow following the execution of the program, which makes it very fast and efficient with no fragmentation.

1.3.2 Dynamic memory management

Sometimes, fixed memory size can be a problem. Static memory does not allow allocation of memory beyond what is declared initially. The *heap* serves this purpose. It is a large pool of memory which must be explicitly managed by the

programmer. It has no guarantee of efficient use of space, memory may become fragmented over time as blocks of memory are reallocated.

It may be tedious for the inexperienced programmer to manage the heap but it allows a more flexible and shareable pool of memory for an efficient programming. To allocate memory on the heap, one must use *malloc()* or *calloc()* from the C code library stdlib.h (when available). There are multiple algorithms to allocate memory when calling these functions. The most common ones are presented below[32].

Sequential fit algorithm

For this memory management algorithm, a single linked list contains the unallocated blocks of memory. When needed, they are allocated using different policies:

- First fit: returns the first block large enough from the list.
- Next fit: similar to first fit but starts where the pointer was left off at the previous iteration.
- Best fit: research through the whole list and returns the smallest block large enough to meet the request.
- Worst fit: returns the largest block from the list.

Buddy allocator algorithm

This algorithm makes use of an array of linked lists. Each list of the array owns blocks of a distinct size. When requested, the buddy allocator algorithm finds the smallest but large enough block to meet the requirement from the array. It then picks one of the blocks from this position in the array.

If the list is empty at the position where the best fitting block is located, it goes to the next position in the array and splits a block from this list to fill the empty position. The opposite can also be applied: two smaller blocks can be merged to obtain a bigger one.

Indexed fit algorithm

This algorithm makes use of an indexed data structure to implement the desired fit. Some common data structures used for this algorithm are trees or hash tables.

Bitmapped fit algorithm

A bitmap representing the usage of the heap is created. Each bit of the map corresponds to a part of the heap. If a part is used, the bitmap is set to 1 otherwise it is set to 0. Allocation is done by searching the bitmap for clear bits.

1.4 Power management

The focus on energy management is something fairly recent in the field of information technology (IT). With the rise of the Internet of Things (IoT), advancements have been made to allow operating systems to manage power consumption more efficiently.

Numerous communication stacks focused on IoT and low energy consumption have been developed in the last decade.

Unfortunately less attention has been paid to the design of energy-efficient OS for resource-constrained devices. Traditional hardware is limited in terms of power management and the progress in this field required both software and hardware to evolve. We will present below some of the advancements made and the technologies developed in recent years[27].

1.4.1 Hardware power management

In order to implement advanced techniques of power management, certain hardware features have been developed. The purpose is to give more control from the software over the hardware.

Clock gating

Clock gating is a technique consisting of turning off the clocks of unused peripherals in order to save energy. Those peripherals enter what is called *idle state* or *sleep*. The clocks are physically switched off from the circuit with the addition of a logical gate and do not consume energy until reactivation.

CPU low-power modes

The recent advancements in CPU have introduced power-saving modes. This feature stops the CPU clocks so that it is put to sleep unless a scheduling event or interrupt is triggered and wakes up the CPU, with the help of a real-time clock (RTC) for example.

Real-time clock wake up support

When a CPU is in sleep mode, there are two possibilities to wake it up with an on-chip RTC or by an external event. The on-chip RTC is a low-frequency clock (usually around 32kHz) that does not drain a lot of battery life. RTC can include alarm functions: timers that when reached, trigger the RTC to wake up the processor.

Supported CPU frequencies / dynamic frequency scaling

In modern CPU, many options are available to switch between frequency ranges depending on the resources needed. This feature can be used to minimize power consumption when we do not need much computational power.

Adaptive voltage scaling / dynamic voltage scaling

Similarly to the CPU frequency, voltage can be regulated based on the actual state of the chip. The voltage is continuously monitored and adjusted during the runtime.

1.4.2 Operating system power management

Peripherals state control

Peripherals state control makes use of the clock-gating feature provided by the hardware. Thanks to this feature, only the peripheral clocks required by the application at a certain point in time are active. The other clocks are gated and do not consume energy.

Sleep mode

The idea is to allow the system to switch off certain components of the microprocessor.

The sleep mode of a microprocessor takes advantage of multiple hardware features such as adaptative voltage scaling, CPU low-power modes and dynamic frequency scaling. The RTC wake up support serves to wake up the CPU when in sleep mode and no other source is active.

Tick suppression

Tick suppression defines the principle of providing tick-less support for the scheduler.

In a regular scheduler, a periodic timer (tick interrupt) is used to track time. This tick interrupt wakes up the CPU to perform a scheduling cycle. Such a mechanism, even if punctual, is frequent and then depletes power in a non-negligible way by entering and exiting sleep mode frequently[3].

In a tick-less scheduler, the tick interrupt is disabled when the idle task is running. Stopping the tick interrupt allows the CPU to remain in a deep power-saving state until either an interrupt occurs or it is time for the kernel to switch task.

1.5 Programming model

The programming model of RTOS can be seen as "the way to program" an application using a specific RTOS. Different "ways to program" or paradigms are predominant in the world of RTOS. In this section, we will present the two main different paradigms used[29].

1.5.1 Event-driven model

In an event-driven programming model, a program generally consists of a main loop which listens for events. Events can be generated by interrupts, sensors or user input. When an event is detected, a callback function is triggered.

The developer has to manually maintain state across tasks which can be tedious. Thus, the individual tasks do not have to maintain their own stack and they use a shared stack. The memory footprint is then reduced since a single stack is used across the application.

1.5.2 Multithreaded model

The multithreaded model allows an application to run different tasks in their own thread context, and communicate between them using an IPC API.

Each thread has its own memory stack and does not require manual management by the programmer. The stack is managed automatically by the thread scheduler. The memory requirements of threaded applications are often larger than their event-driven counterparts.

1.6 Hardware support

The embedded world is responsible for the majority of the world's microcontrollers. The diversity of CPU keep increasing every year. Working with this diversity of constrained devices makes the developer's work harder as he needs to adapt and learn to use specific libraries for each device. A source code on a particular device will not be easily reusable for another device.

With RTOS, developers can more easily make their code portable on many devices.

This section first describes the three classes of constrained devices and how this is related to RTOS. Then, it explains what a hardware abstraction layer (HAL) is and how RTOS uses it in order to be compatible with the largest number of devices.

1.6.1 Constrained device classes

Constrained devices have been classified into 3 classes by the Internet Engineering Task Force (IETF) with the request for comments RFC7228[4] in May 2014. IETF is a standards organization aiming at improving and maintaining the usability of the Internet. Therefore they play a certain role in the evolution of IoT technologies. We can argue that IETF is a competent authority over constrained devices since its domain of competence is the Internet. Nonetheless, the boundary between constrained embedded devices and the Internet is permeable. For the purpose of this thesis, the classification they made with RFC7228 is used. RFCs are informational or experimental documents published by engineers and computer scientists. Some of them become standards and some others are *de facto* standards.

The distinction between the three classes are made with the random-access memory (RAM) and read-only memory (ROM) capabilities of each device. Table 1.1 resumes the different constrained device classes.

	Data size (e.g., RAM)	Code size (e.g., flash)
Class 0 (C0)	$<<$ 10 kB	$<<$ 100 kB
Class 1 (C1)	$\sim$ 10 kB	$\sim$ 100 kB
Class 2 (C2)	$\sim$ 50 kB	$\sim$ 250 kB

Table 1.1: classes of constrained devices

Class 0. These devices are the most constrained. They are typically sensor nodes (or motes) in a sensor network. There are so constrained that they cannot access the Internet without the help of a larger device. The source code of a RTOS is often too heavy to fit in such devices. Instead Class-0 devices are usually used bare metal. In the embedded world, bare-metal programming is writing code that runs directly on the hardware without any abstraction such as an OS.

Class 1. These devices are able to talk to each other but via constrained protocols. The use of security protocols are often too heavy for that class. RTOS generally require at least a few kB for the strict minimal functionalities to run. Therefore, they are not practical for class 0 devices (even if they could technically run) and mainly target class 1 or more powerful devices.

Class 2. These devices are the less constrained and can use the same stacks of protocols used in personal computers and servers. General-purpose OS can be used for this kind of device but the class-2 devices can benefit from lightweight and energy-efficient protocols.

1.6.2 Hardware abstraction layer

Due to the variety of CPU, vendors provide a set of libraries used to develop applications on their architectures. This set of libraries and tools are called the hardware abstraction layer.

Definition of HAL A hardware abstraction layer defines a set of routines, protocols and tools to access underlying hardware. It provides abstract and high-level functions to interact with the hardware. The hardware, drivers and board supports are considered as black boxes.

RTOS and HAL The HAL is strongly dependent on the architecture of the CPU. In order for the RTOS to support multiple boards and architectures, it has to implement and use the different HAL provided by the vendors.

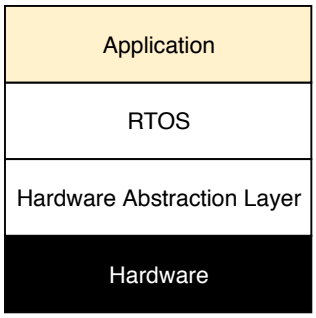

Figure 1.3: layers around the hardware abstraction layer

Figure 1.3 shows that the application and RTOS layers are placed above the HAL, itself just above the device layer. The application layer talks to the RTOS and the RTOS layer talks to the corresponding HAL depending on the device used.

Pro and cons Developers can switch hardware and perform cross-platform testing more easily. But there is some limitation: the HAL is tied to the hardware and change heavily with it. Also, there is some limitation using a hardware abstraction layer. Not all the functionalities from the hardware are available.

Chapter 2

RTOS outline

In this chapter, we will cover the specificities of each OS we planned to work with. We chose to write about technical and non-technical characteristics of the chosen OS and even their respective projects as a whole.

A table summarizing the comparison is provided at the end of the chapter.

2.1 RIOT

2.1.1 Historic

The RIOT project started privately in 2008. It began as part of the FeuerWhere project[12], where firefighters would be tracked with embedded devices during an intervention. The goal was to design an ad hoc self-configurating network of sensors used to monitor vital state and environmental parameters of rescue forces inside a building.

In 2010, a fork from the FeuerWare software developed for the FeuerWhere project was made. Development continued for μkleos[18], a microkernel-based operating system for embedded devices. The focus of this system was modularity and Internet compliance with the integration of IETF protocols such as 6LoWPAN, RPL and TCP.

In 2013, RIOT went public. μkleos was rebranded to avoid problems with spelling and pronunciation. Since then, more than 200 people contributed to the project with more than 20 000 commits and 75 releases. The project is currently supported by Freie Universität Berlin, INRIA and Hamburg University of Applied Sciences.

2.1.2 Characteristics and features

RIOT is an open-source real-time operating system. The OS provides a lightweight operating system with multithreading and real-time features running across a wide range of resources constrained devices. It runs on 8-bit, 16-bit and 32-bit platforms and only needs a minimum of 1,5 kB of RAM. A focus has been made on making RIOT standard-C and C++ compliant.

License RIOT is distributed under the LGPL2.1 license which provides unrestricted commercial or private use, modification and distribution if it is not a derivative work. With LGPL license, RIOT is also provided "as is", without any warranty of any kind.

Technicals The RIOT project aims to bridge the gap between operating systems for Wireless Sensor Networks (WSN) and traditional fully fledge operating systems. RIOT follows some design objectives such as:

- Energy efficiency
- Small memory footprint
- Modularity
- Uniform API, platform agnostic

Community The RIOT community can be characterized in 3 categories[28]:

- Contributors are people who contributed to the project, from code contribution to technical or non-technical discussions.
- Maintainers who contributed noticeably on the project. Maintainers can propose to give the maintainer status to contributors that have been noticed. The status comes with rights (merge rights) and duties (code review duties).
- Coordinator status is acquired following the same principle as the maintainer. The role covers non-technical aspects of the project such as bringing up topics, debates and deal with the overhead.

2.1.3 Specificities

RIOT enforces constant time for kernel tasks. The kernel exclusively uses static memory which guarantees runtimes of $O(1)$ whereas dynamic memory is used for applications.

The scheduler cycle is also performed in constant runtime thanks to a circular linked list of threads[2].

The main specificity of RIOT resides in its scheduler. The operating system includes a tickless scheduler. When there is no pending task, the operating system will switch to the *idle* task which is the thread with the lowest priority. The CPU will then put itself in its deepest possible sleep mode (which depends on the device in use). In this state, the device can only be woken up by an interrupt (external or kernel-generated).

2.2 Contiki

2.2.1 Historic

Contiki[6] was created in 2003 by Adam Dunkels[7]. At the time, it was the first operating system to provide IP connectivity to sensor networks. It did so thanks to μIP[8], a lightweight TCP/IP stack intended for tiny microcontrollers, also developed by Adam Dunkels at the Swedish Institute of Computer Science (now RISE SICS).

Further development of Contiki has been supported by various industries and research institutes such as Texas Instruments, Atmel, Cisco, ENEA, ETH Zurich, Redwire, RWTH Aachen University, Oxford University, SAP, Sensinode, Swedish Institute of Computer Science, ST Microelectronics and Zolertia[9].

2.2.2 Characteristics and features

The development of Contiki started with the need to bring connectivity to small sensor devices with limited capabilities. By definition, Contiki is not a real-time operating system and has not been designed as such.

License Contiki is distributed under a 3-clauses BSD license (also known as Revised BSD license). The advertising clause from the original BSD license which stated that "all advertising materials mentioning features or use of the software must display an acknowledgement to the organization" is removed. This 3-clauses license gives the right to redistribute source code and binaries with or without modifications and the software is provided "as is".

Technicals Contiki operates with the event-driven model. Processes are implemented as event handlers that run to completion. An event handler can be defined as a procedure that performs a specific action in response to an event. Because event handlers cannot block, the same stack can be shared among them. Also, locking

mechanisms are not needed as event handlers do not run concurrently. Instead of having multiple stacks for multiple threads (which must be over-provisioned or they risk running out of memory), a single stack can then be used.

This model does not come without problems, as it is sometimes tedious for programmers to manage. For example, a time-consuming process such as a cryptographic operation can monopolize the CPU for quite some time and make the system unable to respond to external events. In a preemptive multithreading system, the computation can be preempted to react to an external event and continue the process later. To fix this issue, Contiki makes use of an application library that implements preemptible threads. This library is optional an can be used with programs that explicitly require it.

In order to implement its event-driven system, Contiki comes with a mechanism called *protothread*[10]. The concept of protothreads was also developed by Adam Dunkels with the help of other researchers. The goal of the protothread programming abstraction is to make it easier for developers to develop, debug and maintain code written in the event-driven model.

Protothreads can be defined as lightweight stackless threads providing a blocking context on top of an event-driven system. They can be used to perform non-preemptive concurrency or cooperative multitasking. Programs written for an event-driven model typically have to be implemented as explicit state machines. With protothreads, programs can be written in a sequential fashion without designing explicit state machines.

Protothreads do not serve the same purpose as threads for the event-driven model. The concept has been developed to bring a number of benefits of the multi-threaded programming model into Contiki and ease the development of applications.

Contiki does not provide any specific kernel energy saving mechanism. Instead, it lets the application-specific parts of the system implement such mechanisms.

Community From a community perspective, Contiki revolves around a group of core members (a.k.a. the merge team)[26]. Every contribution has to be reviewed by a certain number (1 or 2 depending on the topic) of core members before it can be merged. This organization left the project stalled for years now. Pull requests are not merged anymore into the project.

2.2.3 Specificities

Contiki integrates a tool called Cooja. Cooja is a network simulator which is designed to simulate a network using Contiki motes. Motes can be simulated at

the hardware level or with a standard setup allowing faster response but with less precise system behavior. It is an extremely powerful tool which makes developing and debugging tremendously easier even with a large-scale network.

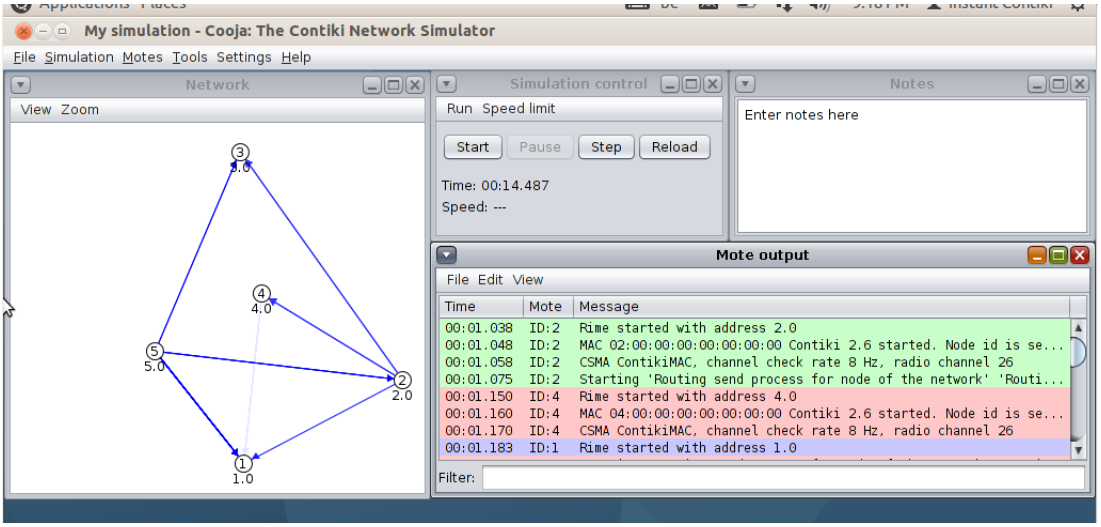

Figure 2.1: Cooja interface

2.3 FreeRTOS

2.3.1 Historic

The FreeRTOS kernel has been developed in 2003 by Richard Barry. He founded a company called Real Time Engineers Ltd to develop and maintain FreeRTOS until the stewardship of the project was passed to Amazon Web Services in 2017. Since then, we can distinguish the FreeRTOS kernel and Amazon FreeRTOS which includes the aforementioned kernel with a set of libraries extending the functionalities of the RTOS.

2.3.2 Characteristics and features

Similarly to RIOT, FreeRTOS is designed to be a real-time preemptive operating system aimed at embedded devices. Its strength comes from the fact that it is highly and easily configurable.

License Both the FreeRTOS kernel and Amazon FreeRTOS are provided under the MIT license[14]. This license gives pretty much no restriction on reusability of the software.

Technicals For example, FreeRTOS supports preemptive or cooperative scheduling[16]. It also uses dynamic priority scheduling which means that the priorities are defined during runtime. The user can choose which scheduling policy he wants by modifying the `configUSE_PREEMPTION` variable in the `FreeRTOSConfig.h` file. In the cooperative case, the only action the scheduler performs is to increase the tick count. In the preemptive case, the scheduler increments the tick count and then checks if a task is in the unblocked state. If it is indeed the case, it checks the priority level of the task and compares it with the current task to see if a switch is required.

Since FreeRTOS uses the multithreading paradigm, the code can be structured as a set of tasks. As explained in the first chapter, each task uses its own stack and therefore has its own context.

In addition to tasks, FreeRTOS includes another mechanism called `co-routine`[13, 34]. Co-routines are similar to tasks but have some fundamental differences which justify their usage:

- They share the same stack between them which greatly decrease the memory wasted in provisioning.
- They use a cooperative scheduling between them but they can be used in an application using preemptive tasks.
- The shared stack comes with more restrictions on how co-routines can be structured compared to regular tasks.

Co-routine have been implemented for very RAM constrained devices and are very rarely used these days. Nonetheless, they stay a relevant alternative to tasks for specific usages.
FreeRTOS also supports tickless idle mode, similarly to RIOT. Since FreeRTOS is highly configurable, the developer can activate this feature by setting the variable `configUSE_TICKLESS_IDLE` as 1 in `FreeRTOSConfig.h`. A custom-built tickless idle mode can also be provided by the developer and used by setting the same variable to 2. This feature can be used when a default implementation in not provided for a certain board.

Community From a community standpoint, FreeRTOS makes the distinction between official and supported code base and user-contributed code. The first is

developed by Amazon Web Services when the second is available on a distinct website.The project is very Amazon-centric and makes it difficult to federate a community.

2.4 Comparison table

The table 2.1 summarize the comparison between RIOT, Contiki and FreeRTOS.

	RAM size	ROM size	Scheduler type	Programming model	License	Communities
RIOT	∼1.5kB	∼5kB	preemptive	mutli-thread	LGPL2.1	++
Contiki	<2kB	<30kB	cooperative	event-driven	Revised BSD	+
FreeRTOS	<1kB	<10kB	configurable	mutli-thread	MIT	-

Table 2.1: RTOS comparisons

Part II

Implementing a benchmarking framework in the context of a RTOS

Chapter 1

Objective

1.1 Implementing a benchmarking framework

The first question one can ask is "What does benchmarking an embedded system means?". A benchmark measures the performances of a defined system, in our case, an embedded system. Those performances can be measured in terms of latency, memory usage or even power consumption. But we can also see benchmarks that include reliability, interoperability or stability measurements.

As part of our work, we want to answer two questions.

- Is it possible to implement a benchmarking framework able to measure precise performance metrics, such as the context switching time, without using an oscilloscope?
- Is it possible to implement a benchmarking framework aware of the RTOS that could give useful metrics such as the memory usage, the power consumption or the CPU utilization?

1.1.1 Motivations

In the RTOS world, benchmarks already exist and some of them are even commercialized. [17, 20, 33] However, those benchmarks are just workloads. They use a defined set of routines that is used to measure the performances of a RTOS. This does not reflect the performances of a real-world application. A benchmark must be a tool that a developer can use to make a fair comparison of two systems.

Developers find themselves using other tools to assess the performances of their applications. Let's take for example, the case of a developer who wants to measure the latency of its application. A benchmark will only provide a latency measurement based on predefined tasks and within a controlled environment. If

our developer wants to measure the real latency of its application, he will need to use an oscilloscope and take the time to do the measurements.

However, these devices are expensive and some developers cannot afford them. Furthermore, working with external devices takes time and can dramatically slow down the development process.

For those reasons, we wanted to build a benchmarking framework that the developer could run on any RTOS with any application.

Ideally, the use case for any developer using our benchmarking framework would be the following. This scenario is hypothetical and does not represent the state of our work. The developer implements an application with multiple tasks on a specific device. The developer wants to measure the performance of its application and, to do so, set a flag in the Makefile of its application to turn on the benchmarking framework. The developer flashes its application on the device and the framework outputs continuous performance measurement of the application. With this framework, the developer can optimize its application or change the device and check if the measurements are improved.

1.1.2 Criteria

Ideally, this framework would respect the following criteria in order to be useful for a real-world application.

Widely distributed By making our framework accessible on a large variety of RTOS, we allow developers not to limit themselves to a specific set of RTOS. With this framework, they would be able to assess the performance of their applications regardless of the platform.

Providing a large variety of metrics Our framework should be able to provide performance measurements on interrupt latency, context switching time. Those metrics are important for a real-time context applications. Also, using constrained devices requires to monitor the memory usage and the energy consumption of the applications. Our framework should be able to provide those metrics as well.

Hidden for the developer The developer should not need to worry about integrating our framework in its application. Using it or not, our framework should not change the source code of the application and allow the developer to directly deploy it in a production environment.

Easy to use and to configure Using an oscilloscope or any external device to benchmark an embedded application takes time and effort for the developer. Our

framework should be able to improve the benchmarking process by being easy to use and easy to configure.

1.1.3 Approaches

In order to develop this framework, we explored three different approaches. We introduced them here so that the reader has already in mind our three experiments but each one of them is described later in the chapter 2.

Integrating the framework in the kernel of the RTOS Our first approach was to integrate the framework in the heart of the RTOS. By focusing our implementation in the kernel space, we would be able to retrieve performance measurements without altering the user space and the application. This approach is detailed in the section 2.1.

Implementing the framework as an extension of the RTOS The majority of RTOS allow users to develop their own extension without altering the kernel source code. We chose to investigate this opportunity with our second approach. The framework would be an extension of RTOS. This implementation is explained in the section 2.2.

Using external devices Finally, our last approach was to use external devices such as a computer or an external board to benchmark our real-world application. This last idea is described in the section 2.3.

1.2 Narrowed objective and limitations

Building a complete benchmarking framework is an ambitious work that requires a lot of time. In the scope of our thesis, we chose to narrow down our objective and produce only a proof-of-concept. By implementing this proof-of-concept framework, we want to show that it is possible to build a large benchmarking framework given more time and effort.

From the criteria previously described, we decided to reduce the first two by focusing only on a specific set of RTOS and a specific metric. We reduced our framework implementation to a benchmarking of the context switching time and to RTOS that use a cooperative scheduler. For the last two criteria, we kept them untouched in order to build a benchmarking framework the most hidden and easy-to-use for the developer.

By implementing a proof-of-concept, we want to show that it is possible to build a complete framework capable of benchmarking the interrupt latency, the memory usage and the energy consumption of any RTOS.

1.2.1 Context-switching time centric framework

We chose to focus on the context switching time for the metric of our benchmarking framework because, in the context of embedded devices, the time spent between two tasks must be the smallest possible. Benchmarking the context switching time can help the developer reduce the time lost between two tasks. Moreover, by considering the interrupt handler as a task, it is possible to compute the interrupt latency with the context switching time.

1.2.2 Scheduler limitations

We chose to only use RTOS with cooperative scheduler because this kind of scheduler allows us to know when a task is in the foreground and when it goes to the background. With a preemptive scheduler, on the other hand, we do not know when the task is preempted.

For example, in Contiki, the developer can use macros like `PROCESS_PAUSE()` or `PROCESS_WAIT_EVENT()` to let other tasks run in a cooperative fashion. The complete list of macros and more information can be found in the Contiki processes documentation [5].

RIOT, on the other hand, uses a preemptive scheduler. But it is possible to implement an application in a cooperative fashion. By defining the task priorities at the same value, it is possible to call `thread_yield()` to provoke a context switch. More information can be found in the scheduler documentation [30] of RIOT.

For those reasons, we chose to focus our work on these RTOS: Contiki and RIOT.

1.3 Reference value

Our benchmarking framework will compute the context switching time but we need to know how well or how bad our framework performs. To assess the performance of our measurements, we first need to compute the real context switching time. This real context switching time will be our reference value.

```
1  PROCESS_THREAD(task, ev, data)
2  {
3      PROCESS_BEGIN();
4
5      while (1)
6      {
7          clock_delay_usec(1000);
8          PROCESS_PAUSE();
9      }
10
11     PROCESS_END();
12 }
```

Listing 1.1: source code of a task implemented in Contiki for the simple application

1.3.1 Simple application

In order to experiment and test our benchmarking framework, we need to have a simple application for every tested RTOS.

The application contains two identical tasks. The tasks wait for 1 ms in the foreground and then goes to the background letting the scheduler run the next task. The tasks run in an infinite loop.

Implementation in Contiki

The source code of one of the two tasks is shown in the listing 1.1. It uses the `clock_delay_usec()` call to make the task wait for 1 ms. Finally, we use the `PROCESS_PAUSE()` macro to pause the task and let the other task do its iteration.

Implementation in RIOT

The source code of one of the two tasks is shown in the listing 1.2. We cannot use the `xtimer_usleep()` method as it will lead to a context switch. Instead, we used a for loop. The `thread_yield()` call will perform a context switch and let the other task run on iteration.

1.3.2 Methodology

In order to compute the real context switching time, we use an oscilloscope and two GPIOs. Each task of our simple application is responsible of one GPIO. In order for the task to use the GPIO, they need to be updated. We describe this change in the subsection 1.3.3. Every time a task is run and goes to the foreground,

```
1  void *threadA(void *arg)
2  {
3      (void)arg;
4
5      while (1)
6      {
7          for(int i = 0; i < 1000; i++) {}
8          thread_yield();
9      }
10     return NULL;
11 }
```

Listing 1.2: source code of a task implemented in RIOT for the simple application

it sets its GPIO up. Once the task is finished and goes to the background, it resets its GPIO down.

With the oscilloscope, we can measure the voltage of the two GPIOs and compute the real context switching time from those measurements. The figure 1.1 shows the different steps of our methodology. On this schema, the execution time of the two tasks is represented above the voltage measurements of the two GPIOs. The context switching time happens between the execution of the two tasks and is bordered by dotted lines.

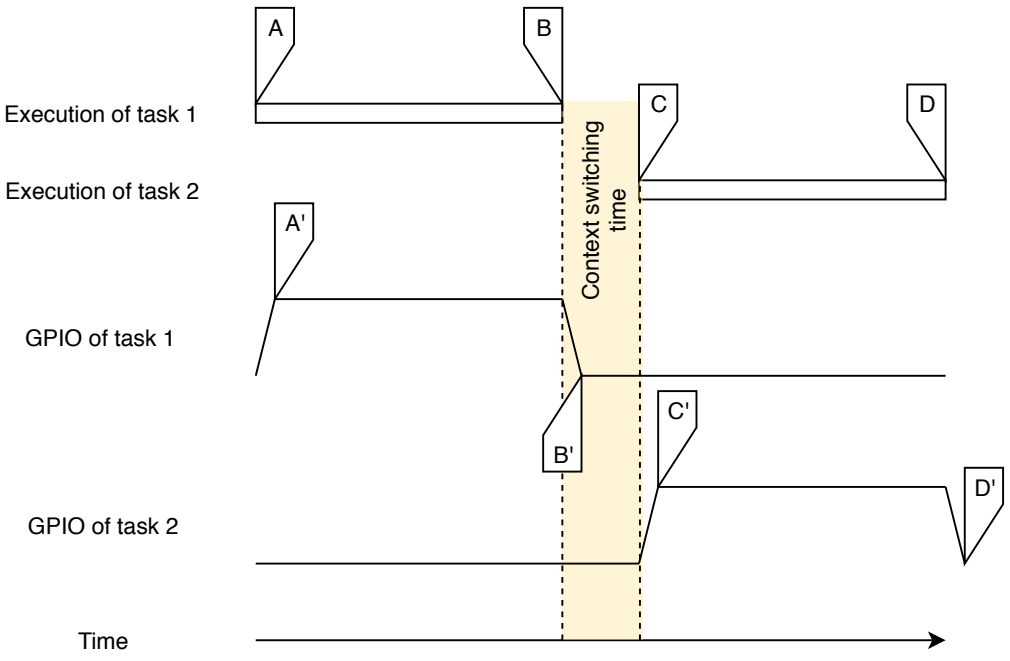

Figure 1.1: steps in the methodology to compute the real context switching time

The steps are the following: the first task sets up its GPIO on step A. The GPIO is in high position few nanoseconds later on step A'. Once the first task is over, it resets its GPIO on step B which goes in low position on step B'. In the same way, the second task set up its GPIO on step C which goes in high position on step C' few nanoseconds later and reset it on step D which goes to low position on step D'. The oscilloscope measures the context switching time between the step B' and the step C'. The time for the GPIOs to rise up or down is around 10 nanoseconds and can be omitted.

1.3.3 Measurements setup

The oscilloscope used for the measurements was the Tektronix MSO 56[22] available at the Welcome Lab at UCLouvain. We used two channels to measure the voltage of the two GPIOs used by the application. The interface of the oscilloscope, shown in the figure 1.2, allows us to directly see in real time our measurements. A table resume the measurements displayed in the graph below while we can check the voltage of our two GPIOs in the bottom-right window. Once we reached 1000 measurements, we exported our data to a flash thumb.

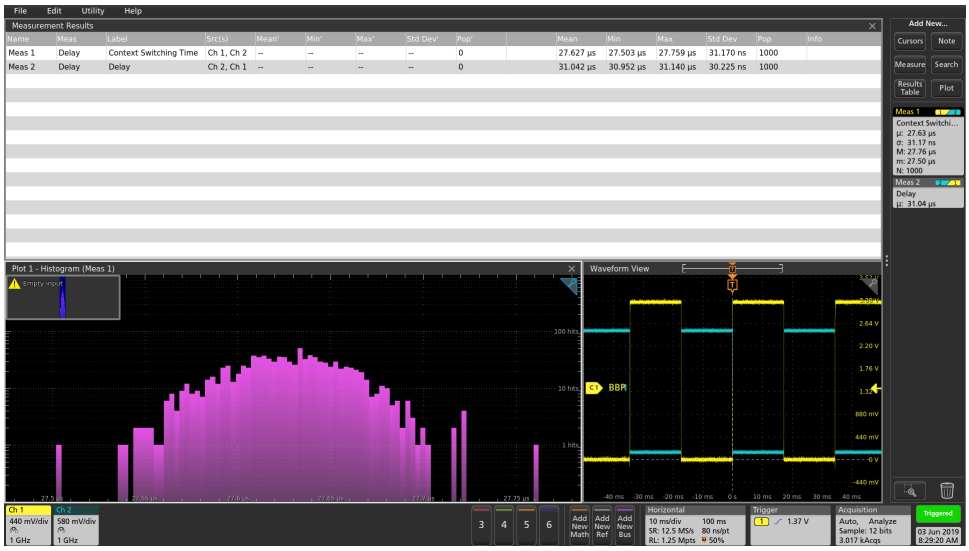

Figure 1.2: interface of the oscilloscope

We updated our simple task by adding GPIO calls in order for the oscilloscope to detect it. The task will set up a GPIO, wait for 1 ms, and then clear the GPIO. The two tasks use different GPIO in order to differentiate them with the oscilloscope.

```
1  PROCESS_THREAD(task, ev, data)
2  {
3      PROCESS_BEGIN();
4
5      while (1)
6      {
7          GPIO_SET_PIN(
8              GPIO_PORT_TO_BASE(GPIO_C_NUM),
9              GPIO_PIN_MASK(3));
10
11         clock_delay_usec(1000);
12
13         GPIO_CLR_PIN(
14             GPIO_PORT_TO_BASE(GPIO_C_NUM),
15             GPIO_PIN_MASK(3));
16
17         PROCESS_PAUSE();
18     }
19
20     PROCESS_END();
21 }
```

Listing 1.3: source code of the task with GPIO calls

Implementation in Contiki

The source code of the updated task is shown in the listing 1.3. `GPIO_SET_PIN()` and `GPIO_CLR_PIN()` are used to respectively set and clear the GPIO used by the task.

Implementation in RIOT

The source code of the updated task is shown in the listing 1.4. `gpio_set()` and `gpio_clear()` are used to respectively set and clear the GPIO used by the task.

Boards used

To perform our measurements, we used two devices from Zolertia. The RE-Mote board[35] and the Z1 board[36]. The CPU of the RE-Mote is a ARM Cortex-M3[21] 32 MHz clock speed with 512 kB flash and 32 kB RAM. The Z1 has a MSP430F2617[23] 16 MHz clock speed with 92 kB flash and a 8 kB RAM.

With those specifications, the Zolertia RE-Mote categorizes itself as a Class-2 device and the Zolertia Z1 as a Class-1 device. Moreover, both Contiki and RIOT support those boards.

```
1   void *thread(void *arg)
2   {
3       (void)arg;
4
5       while (1)
6       {
7           gpio_set(GPIO_PIN(PORT_C, 2));
8
9           for(int i = 0; i < 1000; i++) {}
10
11          gpio_clear(GPIO_PIN(PORT_C, 2));
12
13          thread_yield();
14      }
15      return NULL;
16  }
```

Listing 1.4: source code of a task implemented in RIOT for the simple application

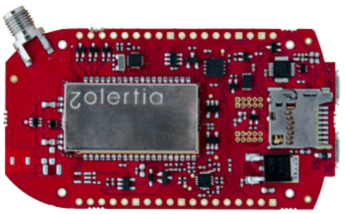

Figure 1.3: Zolerta RE-Mote board

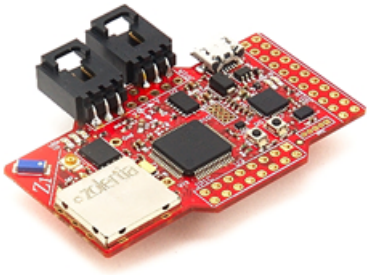

Figure 1.4: Zolerta Z1 board

1.3.4 Measurements results

In the end, we have four different reference values. One for each board and for each RTOS. The results are divided by boards and the measurements from Contiki are compared side-by-side with the measurements from RIOT for more clarity.

RE-Mote measurements

With the RE-Mote board, we measured an average context switching time of 18.505 μs for Contiki and 12.626 μs for RIOT. The table 1.1 shows the statistics of the measurements.

The figure 1.5 and the figure 1.6 show the distribution of the reference value with the RE-Mote board. Note that the y-axis is logarithmic. For the reference value of Contiki, we can see that the majority of the measurements is around 14 μs.

	Contiki	RIOT
Mean (μs)	18.505	12.626
Min (μs)	14.312	12.549
Max (μs)	72.753	12.656

Table 1.1: reference value for Contiki and RIOT on the RE-Mote

Some measurements are above 35 μs with extrema around 70 μs. For RIOT, we find two distributions. One around 12.56 μs and the other around 12.64 μs.

Z1 measurements

With the Z1 board, we measured a higher context switching time than with the RE-Mote board. The average context switching time for Contiki is 54.99 μs and 30.6971 μs for RIOT. It is not a surprise that the RE-Mote board has a smaller context switching time than the Z1 board as the latter is a Class-1 device while the former is a Class-2 device. The table 1.2 shows the statistics of the measurements made with the Z1 board.

	Contiki	RIOT
Mean (μs)	55.077	27.699
Min (μs)	33.061	27.580
Max (μs)	547.95	27.827

Table 1.2: reference value for Contiki and RIOT on the Z1

The figure 1.7 and the figure 1.8 show the distribution of the reference value with the Z1 board. Note that the y-axis is logarithmic. This time, RIOT has a strong distribution around 27.625 μs. For Contiki, on the other hand, we have the majority of our measurements around 32 μs but some measurements are above 100 μs.

Resume

The table 1.3 summarizes our reference values with the two boards, the RE-Mote and the Z1, and the two RTOS, Contiki and RIOT.

Those values will be used during our experiments as the real context switching time on the two boards and with the two RTOS. To assess the performance of our benchmarking framework, we will compare the framework measurements with the reference measurements.

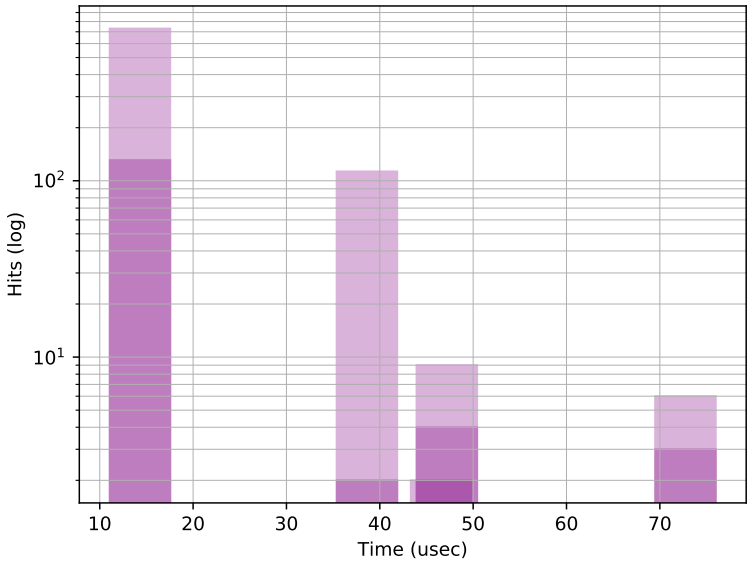

Figure 1.5: reference measurements distribution with Contiki on the RE-Mote board

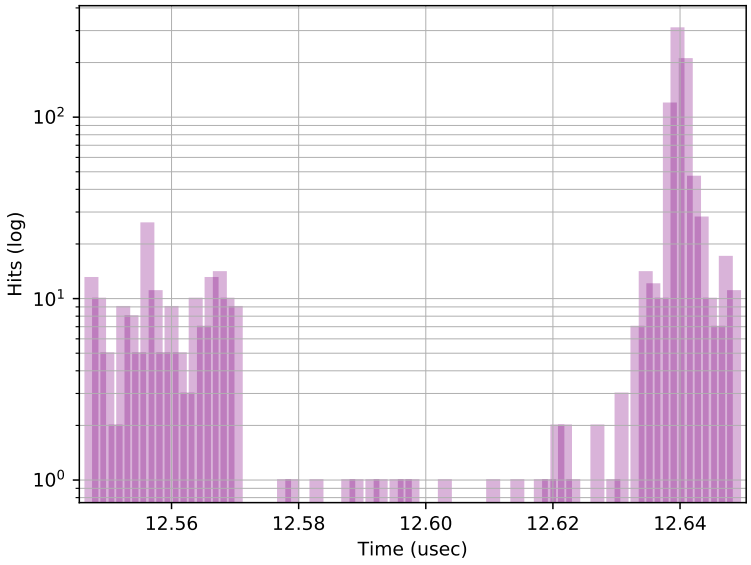

Figure 1.6: reference measurements distribution with RIOT on the RE-Mote board

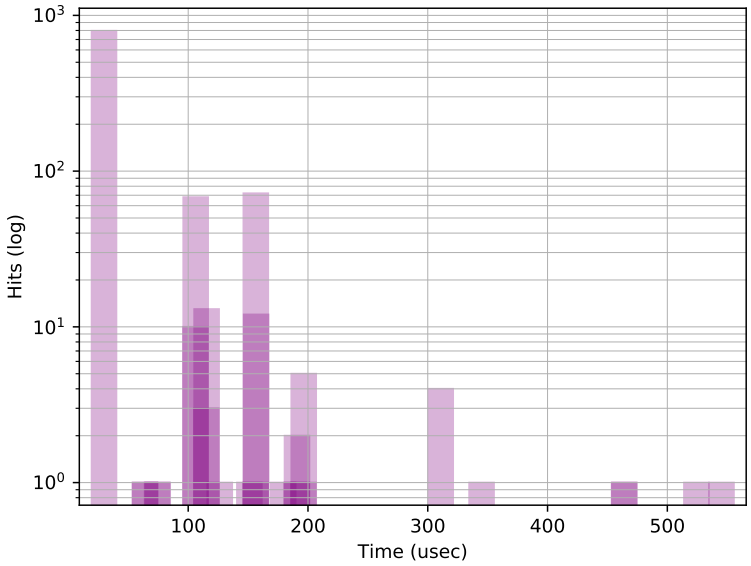

Figure 1.7: reference measurements distribution with Contiki on the Z1 board

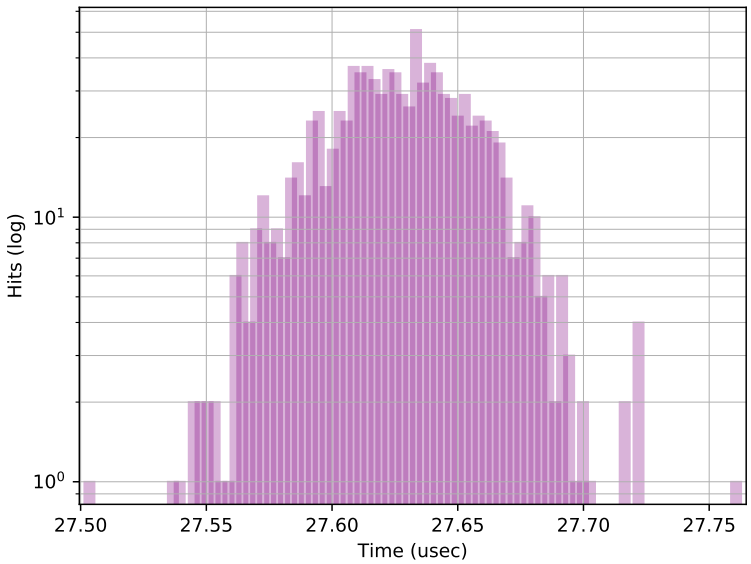

Figure 1.8: reference measurements distribution with RIOT on the Z1 board

	Contiki	RIOT
RE-Mote (μs)	18.505	12.626
Z1 (μs)	55.077	27.699

Table 1.3: resume of the reference values

Chapter 2

Experiment

In order to measure the context switching time, we imagined building our framework as a proxy between the interrupts and the tasks. In this way, our framework would be able to intercept interrupts and context switches in order to measure them as shown in the figure 2.1.

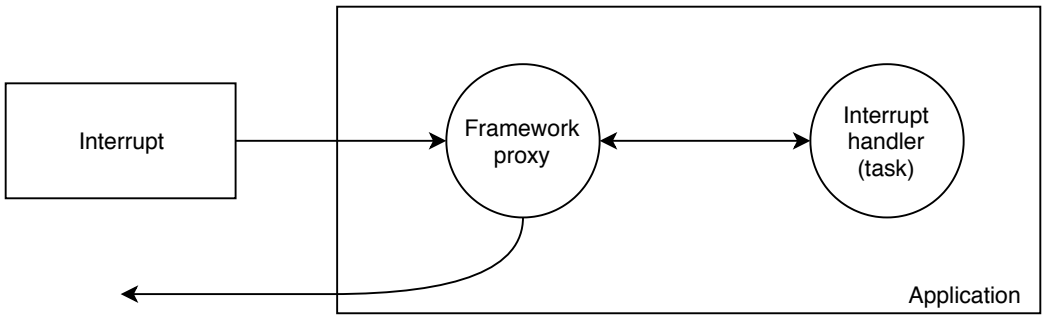

Figure 2.1: benchmarking framework proxy

However, our main concern was to implement a framework that measures the context switching time without altering it or impacting the application. Indeed, we do not want our framework to slow down the application by adding an overhead. To discuss this issue, we contacted and discussed with some IoT communities. Via mailing lists, we communicated with communities from RIOT, Contiki, FreeRTOS, mBed OS and Apache Mynewt. We also exchanged ideas with IoT developers through forums on the web. The different communities gave us some advice and helped us elaborate our three following approaches. These three approaches are described in this chapter.

2.1 Integrating the framework in the kernel of the RTOS

Our first approach was to implement the framework inside the kernel of the RTOS. Using this method, we can compute the time it takes for a context switch to occur directly in the scheduler. In RTOS, when a task goes to the background, the scheduler goes in foreground and resume the next task. The framework would be able to compute the time between the first and the last call of the scheduler computing this way the context switching time.

2.1.1 Motivations

This approach has some advantages. First, it makes the framework completely hidden for the developer. This is great because it matches one of our criteria for the framework. The developer could implement its application regardless of the framework operation. In results, no source code in the user space should be altered making it possible to use the framework on previously implemented applications. Then, one could use the framework on applications already running in production only by updating the RTOS.

2.1.2 Limitations

Unfortunately, we chose to abandon the approach of integrating the framework inside the kernel of RTOS for the following reasons. First, we were not sure that if we used this methodology we would actually measure the real context switching time. It is possible that some calls or functions are executed before the scheduler and that the framework will not take them into account.

Finally, the scheduler implementation is strongly platform-dependent meaning that every platform has its own scheduler source code. It is impossible for us to integrate our framework for each existing platforms. Some of them are even written in assembly code. For example, the listing 2.1 shows a truncated list of all the supported platforms by RIOT.

```
cpu/
|---- arm7_common
|---- atmega1281
|---- atmega1284p
(...)
|---- stm32l1
+---- stm32l4
```

Listing 2.1: truncated list of platforms supported by RIOT

2.2 Implementing the framework as an extension of the RTOS

For this second approach, the idea comes from our discussions with the communities. Instead of building the framework inside the kernel, we would implement the framework as a RTOS extension. The extension would provide calls that would be used in the user space of the application. This approach is later mentioned as the extension approach.

2.2.1 Definition of RTOS extension

Many RTOS allow developers to create their own extensions that will be used by other developers to enhance and add functionalities to their applications. In these extensions, we can find libraries that integrates FTP, COAP or even a Shell. To use those extensions, the developer needs to specify them in the `Makefile`. Those libraries are called 'apps' for Contiki and 'modules' for RIOT.

```
1 CONTIKI_PROJECT = example
2 all: $(CONTIKI_PROJECT)
3 CONTIKI = ../contiki
4
5 # Using the shell app
6 APPS += shell
7
8 include $(CONTIKI)/Makefile.
      include
```

Listing 2.2: example of Makefile using the app `shell` with Contiki

```
1 APPLICATION = example
2 BOARD ?= native
3 RIOTBASE ?= $(CURDIR)/../riot
4
5 # Using the shell module
6 USEMODULE += shell
7
8 include $(RIOTBASE)/Makefile.
      include
```

Listing 2.3: example of Makefile using the module `shell` with RIOT

2.2.2 Framework utilization

The idea of this second approach is to be able to measure the context switching time without impacting the application. With this in mind, we will make sure that our framework is called only twice per task iteration. From these calls, we can deduce the context switching time. The figure 2.2 shows an example with two tasks. The first task starts at the step A and ends at the step A'. The context switch occurs between the step A' and the step B, this latter step being the start of the second task. The step B' is the end of the second task. The idea is to call the framework at each step, A, A', B and B', providing a unique ID. The context switching time is measured between the step A' and B. The next sections describe the implementation of this framework and how it is used with our simple application.

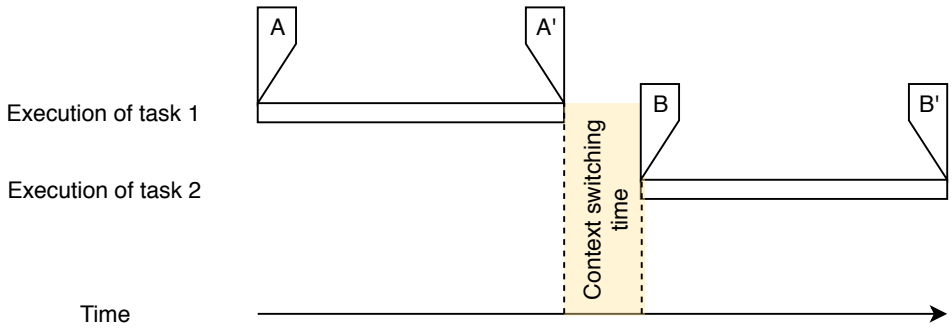

Figure 2.2: pings to the framework

2.2.3 Using the framework with our simple application

To understand the implementation of the framework, we need to understand what happens when our simple application boots. Then, we explain the implementation of the framework in the next section.

In the startup of our application, the first task is launched. It will call the method `bench_ping(TASK_1)` where `TASK_1` is the ID of the first task. At this point, we enter in the framework space like shown in the figure 2.3. The framework will check if the received ID from the `bench_ping()` call is already stored in its context. If the received ID match the one stored in the context or if no ID is stored in the context, no context switch occcured and the framework will reset its internal timer. It also stores the received ID in its context. The stored ID is now `TASK_1`. Once the first task ends its iteration and let the other task run, it calls the same method `bench_ping(TASK_1)` with its ID. Once again, the framework checks the received ID and no change is detected. It resets its internal timer. The stored ID is still `TASK_1`.

Now, the second task starts and calls the framework with `bench_ping(TASK_2)` where `TASK_2` is the ID of the second task. The framework will detect an ID change by comparing the received ID, `TASK_2`, with the stored ID in its context, `TASK_1`. This change means that a context switch occurred. We are between the step A' and B in the figure 2.2. The framework will measure the context switching time by computing the difference between the actual timer and its internal timer. The measure is then written on the serial port to be read by an external source like a computer. The framework resets once again its timer and store the `TASK_2` ID in its context. In this way, we have computed the context switching time between the first and the second task.

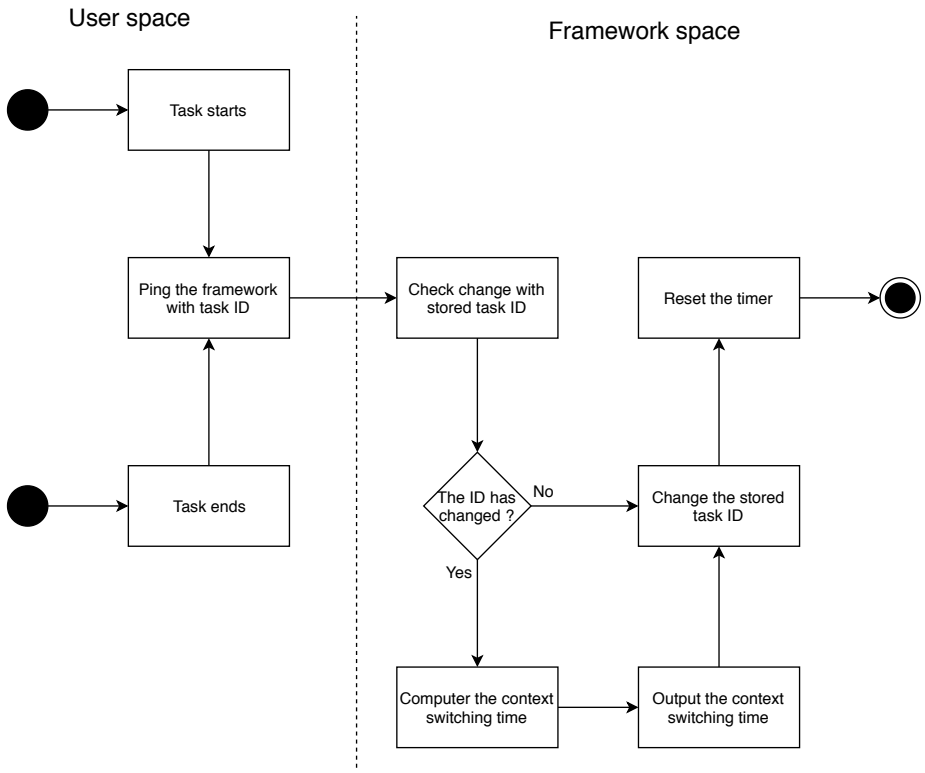

Figure 2.3: activity flow of the framework

2.2.4 Framework implementation

Application source code

For our simple application source code, we just added the call `bench_ping()` at the start and the end of each task. The listing 2.4 shows how the task source code is impacted in Contiki.

Framework source code

In the framework context, we store the received ID in the `new_id` variable and then use the `previous_id` variable to check if a context switch occurred. The `current_time` variable is the interval timer. The code of the framework context is shown in the listing 2.5.

The `bench_ping()` method will just saved the received ID in the `new_id` variable and then check if a change is detected like shown in the listing 2.6.

When a change is detected, the framework will compute the context switching time by retrieving the current time and computing its difference with the `current_time` variable. The time is computed in ticks for a better precision. To re-

```
PROCESS_THREAD(task, ev, data)
{
    PROCESS_BEGIN();

    while (1)
    {
      bench_ping(TASK_ID); // Ping the framework
      // Wait for 1ms
      clock_delay_usec(1000);
      bench_ping(TASK_ID); // Ping the framework
      PROCESS_PAUSE();
    }

    PROCESS_END();
}
```

Listing 2.4: source code of the application task with `bench_ping()` calls

```
struct BContext {
  uint32_t previous_id;
  uint32_t new_id;
  clock_time_t current_time;
} bench_context;
```

Listing 2.5: framework context implementation

```
void bench_ping(uint32_t id)
{
  bench_context.new_id = id;
  if (!check_change())
  {
    bench_context.current_time = RTIMER_NOW();
  }
}
```

Listing 2.6: `bench_ping()` implementation

```
// Compute the difference
clock_time_t previous = bench_context.current_time;
clock_time_t current = RTIMER_NOW();
clock_time_t result = current - previous;

// Keep the previous id for log
uint32_t previous_id = bench_context.previous_id;
// Change previous_id to new_id
bench_context.previous_id = bench_context.new_id;

bench_context.current_time = RTIMER_NOW(); // Ticks

printf("[BENCH_CONTEXT_SWITCHING] %lu %lu %lu\n", previous_id,
    bench_context.new_id, result);
```

Listing 2.7: source code of the benchmarking framework implemented in Contiki

trieve the time in ticks, we used `RTIMER_NOW()` in Contiki and `xtimer_now().ticks32` in RIOT. The internal timer is then reset just before writing the measure into the serial port. Then the measure is written to the serial port using the `printf()` call. The code of this measurement is shown in the listing 2.7.

All the source codes can be found in the Github repository[1].

2.3 Using external devices

Our third and last approach was to use an external device to compute the context switching time. In this way, all the heavy computational processes are done outside of the benchmarked board. This approach is later mentioned as the devices approach.

2.3.1 Choice of the external device

Our first idea was to implement it with Python3.7 on a desktop computer. Using the Universal Asynchronous Receiver-Transmitter (UART) protocol over USB, the board sends a single byte containing the thread ID that is read by a Python script on the computer.

The motivations for using this alternative are:

- Sending one byte of data over USB with UART have a smaller impact than computing the context switching time locally on the board;

[1] `https://github.com/bench-os/bench-os`

- Time consuming computational tasks of the framework are done on the computer and not on the board;
- Using Python3.7, we can achieve a time precision at the nanosecond.

However, after discussing with the embedded community, we abandoned this idea for the following reasons:

- There is buffering happening on the USB-serial chip on the board, on the PC's USB hardware, in the PC USB-serial driver and also in the desktop operating systems;
- Those buffering will add delay in our measurements;
- Context switches will occur on the desktop computer that will invalidate any timing value.

Instead, we decided to use a device called the Pocket Science Lab (PSLab) to perform our experiments.

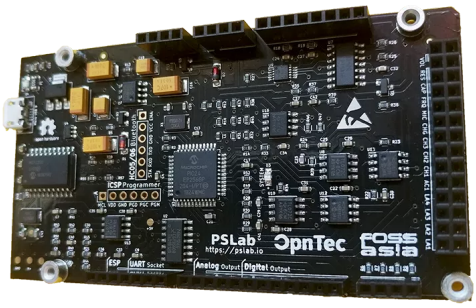

Figure 2.4: Pocket Science Lab device from PSLab.io

The Pocket Science Lab device from PSLab.io[2] comes with a built-in 4-channel up to 2MSPS oscilloscope, multimeter, 4-channel, 4 MHz logic analyzer, and other digital instruments. Using the Python library `pslab-python`[3], we can communicate with the board and experiment with it. With a device like the PSLab, we can leave the benchmarked board that contains our simple application untouched in terms of heavy computations.

[2] `https://pslab.io`
[3] `https://github.com/fossasia/pslab-python`

2.3.2 Role of the different devices

With the benchmarked board, the PSLab and the computer, we have three devices that we use to compute the context switching time. The figure 2.5 shows the connections between the different devices. To make some clarity, we defined a specific role to each device.

The benchmarked board

The benchmarked board runs the RTOS of our choice with the benchmarking framework and the simple application. It communicates with the computer through UART and with the PSLab using a single GPIO. The GPIO channel is used for timing computation while the UART channel is used to flash and read any serial output from the board by the computer. Its role is to simply run the application and change the state of the GPIO depending on the tasks status.

The computer

The computer is the brain of the benchmarking framework. It is responsible for communicating with both the benchmarked board and the PSLab through the UART protocol. It coordinates the benchmarked board and the PSLab in order to retrieve the context switching time. The computer is also the place where all the data are gathered.

The PSLab

Connected with a single GPIO to the benchmarked board, the PSLab watches for any context switch. It measures the context switching time using its logical analyzer. The board receives its instructions and sends the measures through UART with the computer.

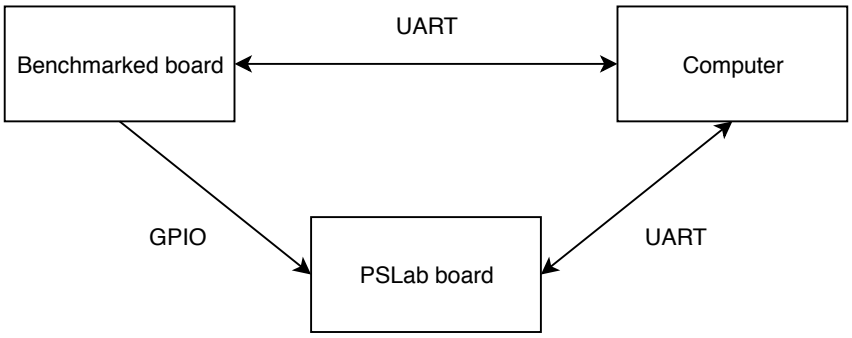

Figure 2.5: interaction schema of the devices used in the framework

2.3.3 Usage of the Pocket Science Lab

The PSLab monitors a single GPIO and measures the context switching time from it. The figure 2.6 shows the steps in the measurement with the PSLab. Each task will set the GPIO up at the start of its execution and then reset the GPIO once it ended. In our example, the task 1 set up the GPIO at the step A and the GPIO is in high position at step A'. Once the task 1 is finished, it resets the GPIO at the step B that will be in low position at step B'. The same process occurs for the task 2. The task 2 set up the GPIO at step C. The GPIO is in high position at step C'. Finally, the task 2 reset the GPIO at the end of its execution at step D and the GPIO will be in position low at step D'. From our measurements made with the reference value, we know that the rising and falling times of the GPIO is around 10 nanoseconds so it can be omitted.

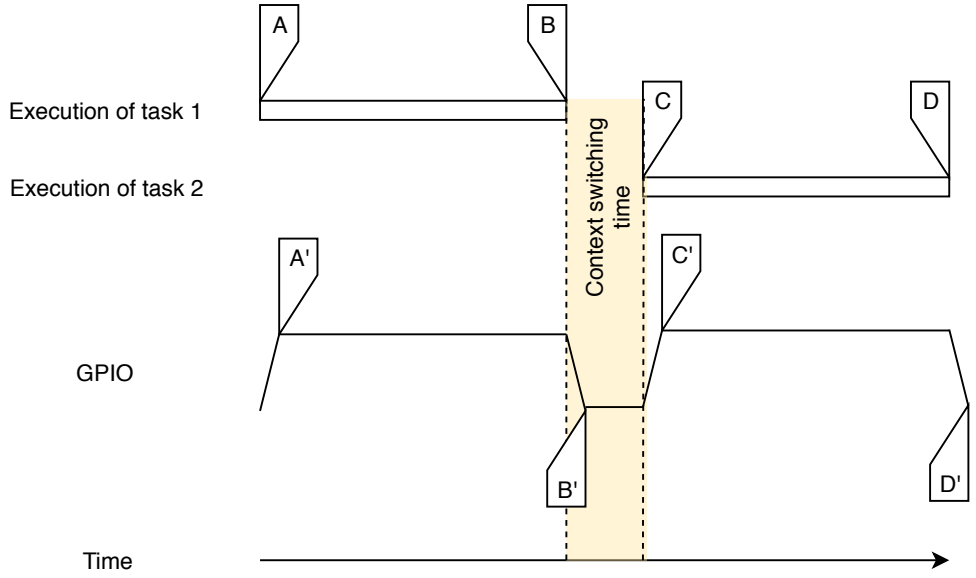

Figure 2.6: measurement of the context switching time with a single GPIO

2.3.4 Using the framework with our simple application

To understand the framework implementation, it is important to understand what happens when our simple application boots, both from the point of view of the benchmarked board and from the point of view of the PSLab and the computer.

Point of view of the benchmarked board

When the application boots, it will initialize the framework. An initialization is required to use the GPIO. Once the initialization is done, the first task starts. It will call `bench_on()` from the framework. In the framework space, `bench_on()` will have for effect to set the GPIO up. When the task is done, it will call `bench_off()` that will reset the GPIO. The second task goes in the foreground and will also call `bench_on()` and, later, `bench_off()`.

Point of view of the PSLab and the computer

The PSLab will listen for an interval between a falling and a rising edge. In the figure 2.6, this interval happens between the step B' and C'. For reminder, the rising time can be omitted. Once the PSLab detects an interval, it computes its duration. This interval time is the context switching time. This measure is sent to the computer that will store it.

2.3.5 Framework implementation

The interaction between the benchmarked board and the PSLab is done through a GPIO that is controlled by the benchmarked board. The code in the benchmarked board is quite straightforward. First, our simple application needs to initialize the framework and start its tasks. Then, we need to add `bench_on()` and `bench_off()` respectively at the start and at the end of the two tasks. The code of the updated application is shown in the listing 2.8.

During the initialization, the framework will setup the GPIO as an output port. The listing 2.9 shows this process. The `bench_on()` and `bench_off()` will just set or reset the GPIO like shown in the listing 2.10.

```
1 void bench_init()
2 {
3     GPIO_SOFTWARE_CONTROL(GPIO_PORT_TO_BASE(GPIO_C_NUM),
    GPIO_PIN_MASK(2));
4     GPIO_SET_OUTPUT(GPIO_PORT_TO_BASE(GPIO_C_NUM), GPIO_PIN_MASK(2));
5 }
```

Listing 2.9: initialization of the framework in Contiki

```
1 void bench_on(uint32_t pid)
2 {
3     GPIO_SET_PIN(GPIO_PORT_TO_BASE(GPIO_C_NUM), GPIO_PIN_MASK(2));
4 }
5
6 void bench_off()
7 {
```

```c
#include "contiki.h"
#include "sys/clock.h"
#include "bench-context-switching.h"

#include <stdio.h>

PROCESS(init_task, "Init task");
PROCESS(task_1, "First task");
PROCESS(task_2, "Second task");
AUTOSTART_PROCESSES(&init_task);

PROCESS_THREAD(init_task, ev, data)
{
    PROCESS_BEGIN();

    bench_init();

    process_start(&task_1, NULL);
    process_start(&task_2, NULL);

    PROCESS_END();
}

PROCESS_THREAD(task_1, ev, data)
{
    PROCESS_BEGIN();

    while (1)
    {
        bench_on();
        clock_delay_usec(1000);
        bench_off();
        PROCESS_PAUSE();
    }

    PROCESS_END();
}

// ...
// task_2 is identical to task_1
```

Listing 2.8: source code of the simple application in Contiki

```
8       GPIO_CLR_PIN(GPIO_PORT_TO_BASE(GPIO_C_NUM), GPIO_PIN_MASK(2));
9   }
```

Listing 2.10: `bench_on()` and `bench_off()` implementation in Contiki

Finally, from the computer, we need to retrieve and store the interval measurements from the PSLab. To do so, we use a Python script shown in listing 2.11 that connects to the PSLab and retrieves the context switching time between a falling and a rising edge.

The complete source code can be found on the Github repository[4].

```
1 from PSL import sciencelab
2 I = sciencelab.connect()
3
4 VALUES = []
5
6 while True:
7     CS_TIME = I.MeasureInterval('ID1', 'ID1', 'falling', 'rising')
8     VALUES.append(CS_TIME)
```

Listing 2.11: Python script to communicate with the PSLab and retrieve the interval measurements

[4] `https://github.com/bench-os/bench-os`

Chapter 3

Results and framework measurements

This chapter gives all the measurements gathered during our experiments. Those results are given without any comment. Each presented plot has a logarithmic y-axis. The chapter 4 discusses the results and compare the two approaches. Section 3.1 gives measures of the impact of the different approaches. Section 3.2 gives the obtained measurements with the two approaches.

For each approach, 1000 measurements were made with the two RTOS, Contiki and RIOT, and on the two boards, the RE-Mote and the Z1 boards for a total of 4000 measurements. From these measurements, the statistics used were the mean, the minimum and the maximum.

3.1 Overhead

One of our first concerns was to implement a framework that does not impact our simple application. The benchmarking framework should be hidden for the developer. Enabling the framework should not alter the good use of the application.

To measure the impact of our approaches, we used the same setup as with our reference value and retrieved 1000 measurements of the context switching time. We compared those measurements with our reference value and computed the overhead of our framework. The oscilloscope measurements are shown in the figure 3.1.

3.1.1 Extension approach overhead

The extension approach has a large overhead of more than 2 ms for with RE-Mote board and more than 3 ms with the Z1 board on both Contiki and RIOT. In the figure 3.2, we can see the overhead of the first approach with the measurements

made with the oscilloscope. We discuss later why our extension approach adds so much latency in the section 4.1.

3.1.2 Devices approach overhead

The devices approach, on the other hand, has a small overhead of less than 3 μs with either the RE-Mote or the Z1 boards on both Contiki and RIOT. In the figure 3.3, we can see the overhead of the devices approach. The devices approach adds a much smaller overhead to our simple application than the extension approach. This difference between the overhead of the two approaches is discussed in the section 4.1.

3.2 Framework measurements

We describe in this section the results gathered with the extension approach described in the section 2.2 and with the devices approach described in the section 2.3. For recall, the first approach is to implement the framework as a RTOS module used in the user space and that will compute internally the context switching time. The second approach is to measure the context switching time using an external board, the PSLab.

In the same way as with the reference value, the results are divided by boards. The measurements made with the RE-Mote boards are first given, then the one made with Z1 board. The Contiki measurements are displayed side-by-side with the RIOT one for more clarity.

3.2.1 Extension approach measurements

RE-Mote board measurements

With Contiki, the framework outputs measurements with an average of 31.6162 μs. With 0 μs as the minimum value and 457.7636 μs as the maximum value, the measurements have a large distribution. We have a difference of 13.1112 μs between the measured context switching time and the real one measured with the oscilloscope.

With RIOT, the framework outputs a constant context switching time of 17 μs. All 1000 measurements were equal to 17 μs. Thus, the difference with the real context switching time is 4.374 μs.

Both measurements were above the real context switching time given by the reference value. The table 3.1 shows the measurement of Contiki and RIOT for the RE-Mote board.

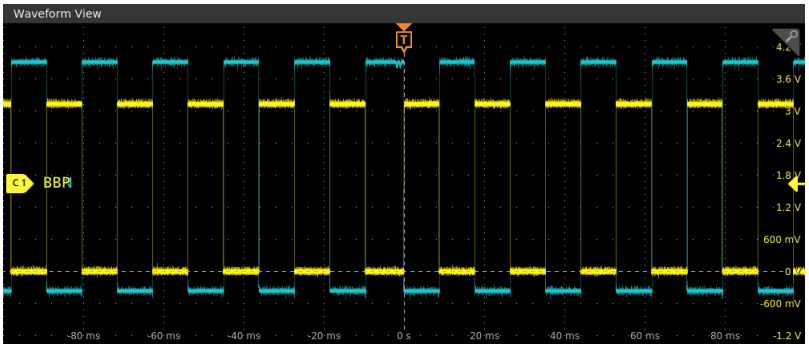

Figure 3.1: overhead reference measured by the oscilloscope

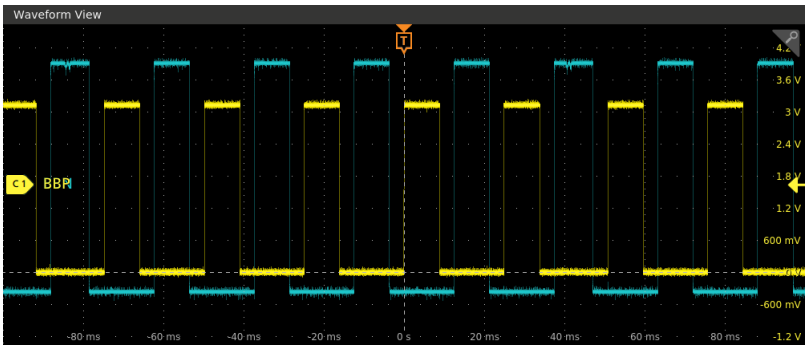

Figure 3.2: extension approach overhead measured by the oscilloscope

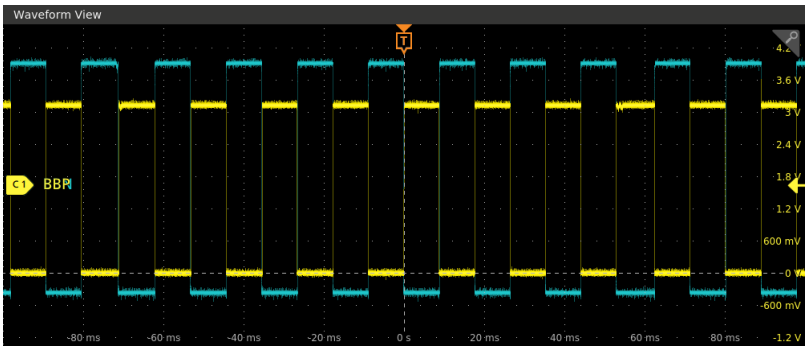

Figure 3.3: devices approach overhead measured by the oscilloscope

	Contiki	RIOT
Mean (μs)	31.6162	17
Min (μs)	0	17
Max (μs)	457.7636	17

Table 3.1: extension approach measurements for Contiki and RIOT on the RE-Mote board

Z1 board measurements

On the Class-1 Z1 board, the framework gave an average context switching time of 90.5761 μs with Contiki. Once again, on Contiki, our values were largely distributed between 30.5175 μs and 976.5625 μs. The average context switching time measured is 35.4991 μs above the real context switching time.

With RIOT, most of the values were equal to 40 μs with an average of 40.252 μs. This put the measurements 12.553 μs above the real context switching time.

The table 3.2 shows the measurement of Contiki and RIOT for the Z1 board.

	Contiki	RIOT
Mean (μs)	90.5761	40.252
Min (μs)	30.5175	40
Max (μs)	976.5625	41

Table 3.2: extension approach measurements for Contiki and RIOT on the Z1 board

3.2.2 Devices approach measurement

RE-Mote board measurements

On the RE-Mote board, the devices approach outputs an average context switching time of 19.0329 μs on Contiki. This measurement differs from the real context switching time by only 2.85%. They are 0.5279 μs above the reference measurements. The majority of the values were around 14 μs but some of them were above 40 μs. The figure 3.4 show the distribution of the measurements made with Contiki.

With RIOT, the average context switching was 12.9832 μs. For RIOT, the measurements were either around 12.95 μs or around 13 μs. The figure 3.5 shows this disparity. The measured context switching time is above the real context switching time by 0.3572 μs. This represents an offset of 2.82% from the reference measurements.

The table 3.3 shows the measurement for Contiki and RIOT for the RE-Mote board.

Z1 board measurements

With the second approach, on the Z1 board, the framework found an average context switching time of 54.99 μs with Contiki. The framework measurements were 0.15% off from the real measurements by being 0.087 μs below them. However, with Contiki the measured values are more scattered like shown in the figure 3.6.

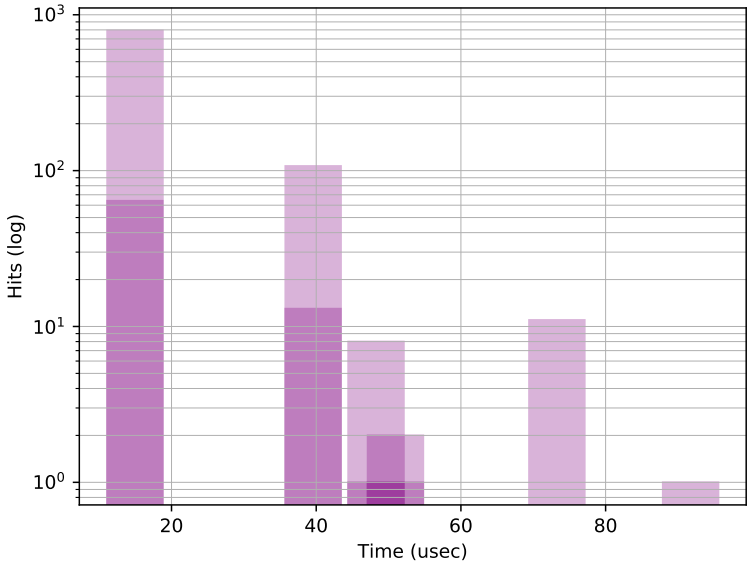

Figure 3.4: devices approach measurements distribution with Contiki on the RE-Mote board

	Contiki	RIOT
Mean (μs)	19.0329	12.9823
Min (μs)	14.9375	12.9375
Max (μs)	91.75	13.0156

Table 3.3: devices approach measurements for Contiki and RIOT on the RE-Mote board

The minimum measured context switching time is 32.5625 μs and the maximum one is 314.5 μs.

With RIOT, the framework outputted an average context switching time of 30.6971 μs. The difference with the real context switching time is 2.9981 μs making the framework measurements more than 10% above the reference measurements. On the other hand, the framework measurements gave the same distribution as the one found with the oscilloscope in the section 1.3.4 with the figure 1.8. The measured measurements distribution is shown in the figure 3.7.

The table 3.4 shows the measurement for both Contiki and RIOT on the Z1 board.

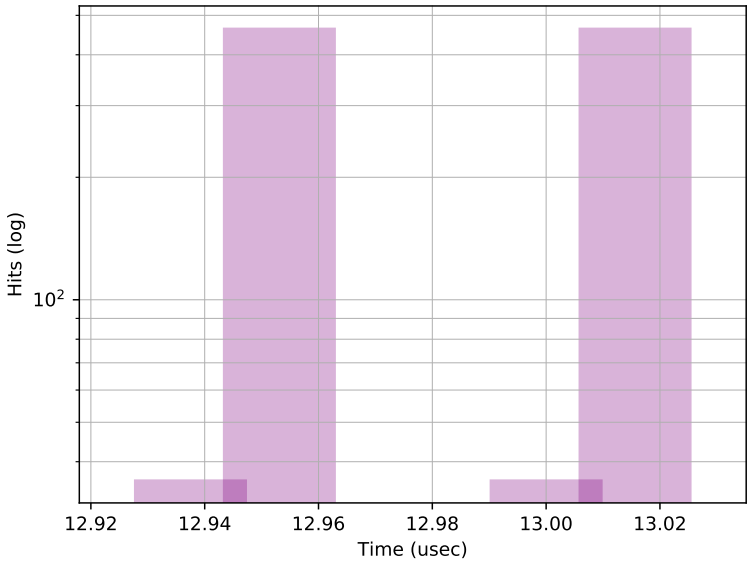

Figure 3.5: devices approach measurements distribution with RIOT on the RE-Mote board

	Contiki	RIOT
Mean (μs)	54.99	30.6971
Min (μs)	32.5625	30.5312
Max (μs)	314.5	30.8437

Table 3.4: devices approach measurements for Contiki and RIOT on the Z1 board

3.2.3 Summary

To give more clarity to the results, the following tables give the average context switching time measured with the two approaches. The table 3.5 gives the measurements made on the RE-Mote board. The table 3.6 gives the measurements made on the Z1 board. The difference between the measured context switching time and the real context switching time measured with the oscilloscope is written between parenthesis.

	Contiki	RIOT
Extension approach (μs)	31.6162 (13.1112)	17 (4.374)
Devices approach (μs)	19.0329 (0.5279)	12.9832 (0.3572)

Table 3.5: Framework measurements resume on the RE-Mote board

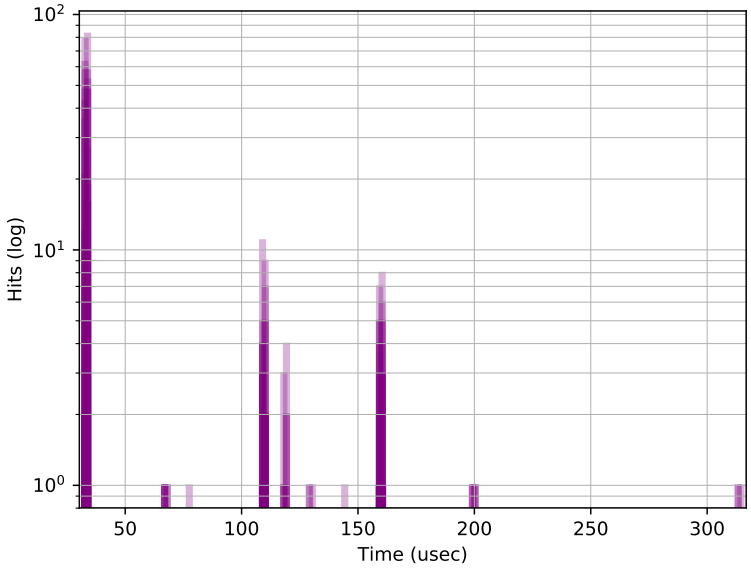

Figure 3.6: devices approach measurements distribution with Contiki on the Z1 board

	Contiki	RIOT
Extension approach (μs)	90.5761 (35.4991)	40.252 (12.553)
Devices approach (μs)	54.99 (-0.087)	30.6971 (2.9981)

Table 3.6: Framework measurements resume on the Z1 board

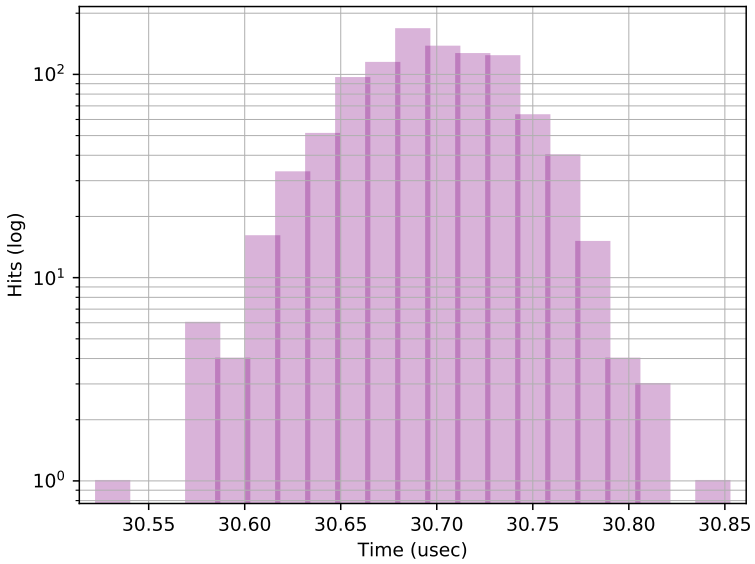

Figure 3.7: devices approach measurements distribution with RIOT on the Z1 board

Chapter 4

Discussion and comparison of the approaches

During our experiments, we tried to measure the context switching time of RTOS with two different approaches. In this chapter, we discuss and interpret the results described in the chapter 3. From those results, we detail in the following sections four observations that we have made.

4.1 Extension approach overhead issue

With our first approach, the framework adds to our simple application a large overhead of more than 2 ms for every context switch. This is an issue for our benchmarking framework as it should have a minimal impact on our application. With such an overhead, the RTOS loses its real-time capabilities. It becomes impossible for the RTOS to react in time to interrupts.

4.1.1 Causes

Sending measurements through the serial port creates this latency. Indeed, the framework sends a string that contains the measured value in order for the computer to read it. For example, the listing 4.1 shows how the framework writes the measurements on the serial port with the method `printf()` with Contiki.

```
1  printf("[BENCH_CONTEXT_SWITCHING] %lu %lu %lu\n", previous_id,
     bench_context.new_id, result);
```

Listing 4.1: writting the measurements to the serial port with Contiki

The string `[BENCH_CONTEXT_SWITCHING]` is a flag used to differentiate framework outputs from the application outputs. However, printing out at least 32 characters in every context switch on the serial port even at 250 kbit/s took 1.2 ms.

4.1.2 Solutions and improvements

To improve the extension approach, one optimization could be to reduce the number of bits send through the serial port. By sending compressed data and using only a small number of bytes, it could be possible to reduce the overhead. But even with 8 bits for the flag a 32 bits for the context switching time, 40 bits would take up to 160 μs to be transmitted.

In addition to compress the data, another solution would be to use a cache storing the measurements. This cache would be flushed and the values written on the serial port at appropriate times, either when the cache is full or with a framework method that would allow the developer to choose when to flush the cache. This way, the overhead occurs not at each context switch but only at an appropriate time.

4.1.3 Overhead of the devices approach

On the other hand, the devices approach has a small impact on the system with an overhead of less than 3 μs per context switch. Even if it is a small overhead, it is however present. Our hypothesis on this overhead is that the framework calls, `bench_on()` and `bench_off()` have an impact on the system. However, there is no fact supporting this conclusion.

4.2 Assessment of the different approaches

Based on our reference measurements from the section 1.3, we can assess the performances of our two different approaches. To determine which approach perform the best, the difference between the measured context switching time and the real one is used. Moreover, comparing the distributions of the measurements can refine the analysis.

4.2.1 Extension approach performance

For our extension approach, we can say that it performs poorly. For example, on Contiki with the RE-Mote board, the expected context switching time is 18.505

μs but our framework measured a context switching time of 31.6162 μs. This represents a difference of 70%.

The real-time clock used to measure the context switching time is not precise enough. This lack of precision explains the framework results. During our experiments, we have noticed that on the Z1 board, the real-time clock has a speed of 32768 ticks per second. In other words, a tick occurs every 30.5 μs. This is why the minimal value measured with Contiki on the Z1 board is 30.5175 μs. The same logic can be applied with RIOT or the RE-Mote board.

With a slow clock, it is not possible to have precise measures for the context switching time. Unfortunately, we did not find any solution to overcome this problem. Therefore, it is not possible to compute the context switching time internally.

4.2.2 Devices approach performance

With the devices approach, we have better results than with our first framework. First, the measurements done with the framework differ only by 2.8% with both Contiki and RIOT.

Wuth Contiki, we have a maximum difference of 0.5 μs with the reference value. When comparing the distributions of the framework measurements with the distributions of the reference measurements, we can see that they match. The figure 4.1 shows the distributions comparison with measurements made on the RE-Mote board.

The figure 4.2 shows the distributions comparison with measurements made on the Z1 board.

With RIOT, we have a maximum difference of 3 μs. With the RE-Mote board, the difference between the real context switching time and the measurements made by the framework is 0.35 μs. In the other hand, with the Z1 board, this difference is 2.99 μs. We were not able to determine from where this difference comes from. However, by comparing the measurements distributions, we see a strong correlation between the reference measurements and the framework measurements.

The figure 4.3 shows the distributions comparison with measurements made on the RE-Mote board and the figure 4.4 shows the same comparison but on the Z1 board.

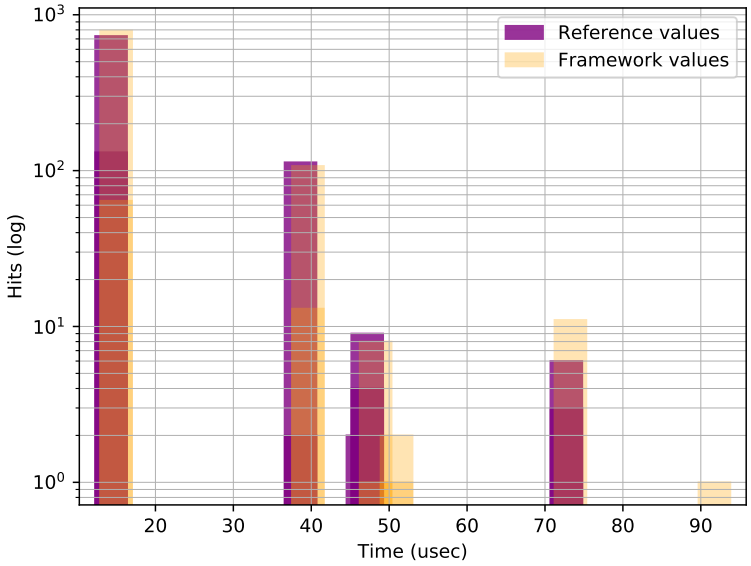

Figure 4.1: distributions comparison with Contiki on the RE-Mote board

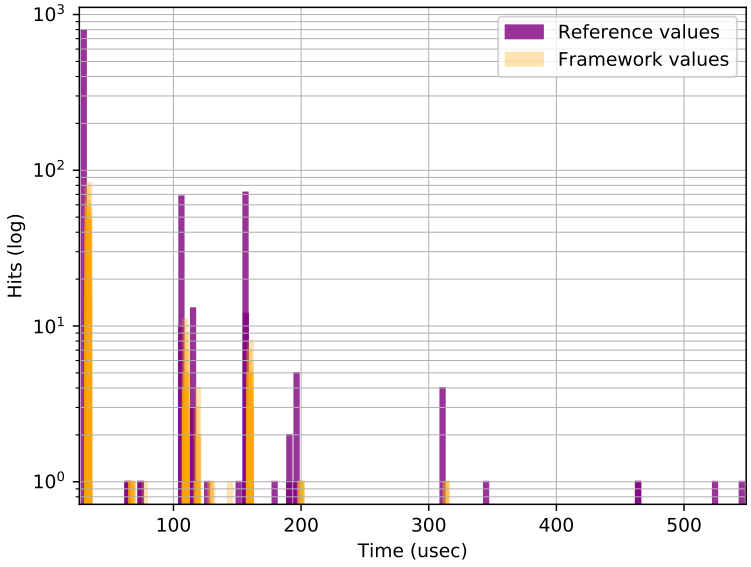

Figure 4.2: distributions comparison with Contiki on the Z1 board

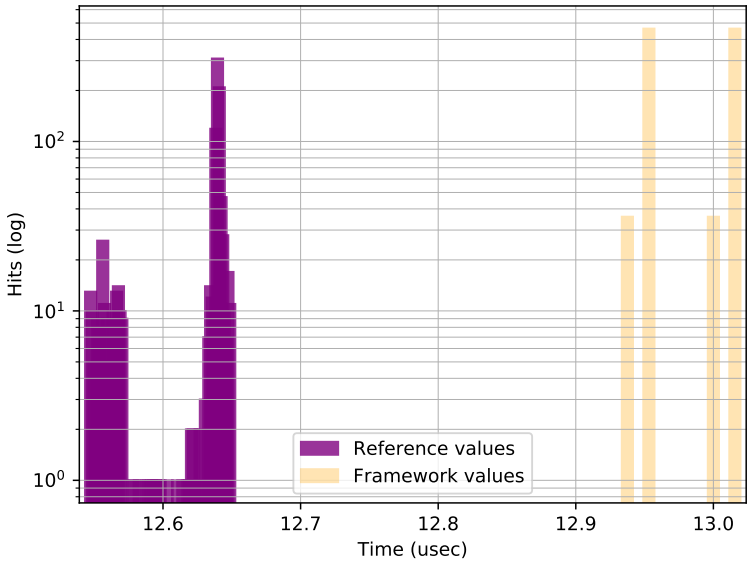

Figure 4.3: distributions comparison with RIOT on the RE-Mote board

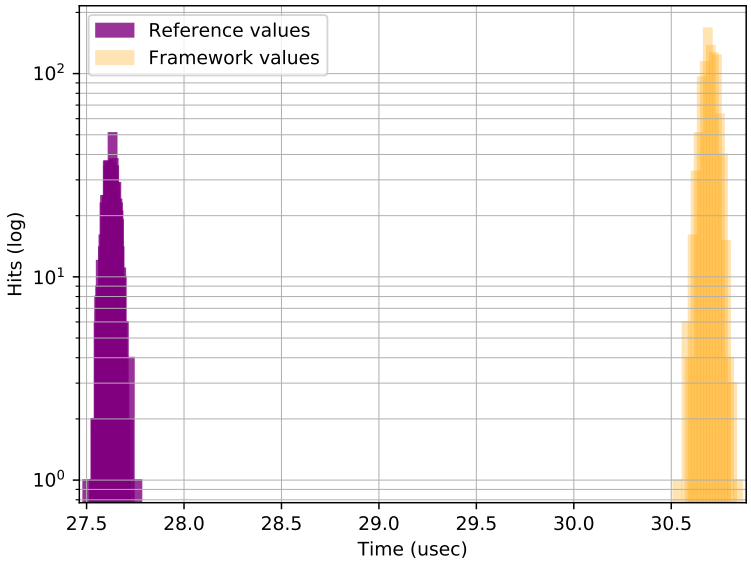

Figure 4.4: distributions comparison with RIOT on the Z1 board

4.3 Contiki measurements distribution

By comparing the measures made with Contiki to the one made with RIOt, we noticed a huge difference. With RIOT, the standard deviation is 31.16 ns with

the RE-Mote board and 27.61 ns with the Z1 board. With Contiki, the standard deviation is 10 μs with the RE-Mote board and 54 μs with the Z1 board. The figure 4.5 and the figure 4.6 clearly show the difference between the distribution of the reference value between Contiki and RIOT. At this point, we do not know where this difference comes from.

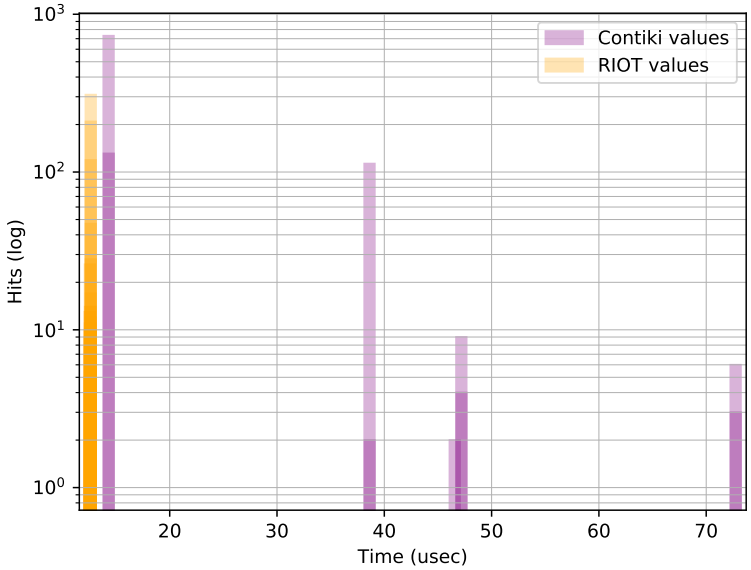

Figure 4.5: distribution comparison between RIOT and Contiki with the reference value on the RE-Mote board

4.4 Summary

With two RTOS, Contiki and RIOT, and two boards, the RE-mote and the Z1, we have collected data that will define which approach best fits our framework.

With our first approach, we discovered that using the internal real-time clock to compute the context switching time is not suited for our benchmarking framework. Moreover, outputting measurements to the serial port induce a large overhead that could prevent the benchmarked application to run correctly.

The second approach helps our framework to gather precise measurements. With the PSLab as the external devices, the framework was able to measure precisely the context switching time on both Contiki and RIOT.

When comparing the distribution of the two approaches to the real context switching time distribution, it is clear that the devices approach is the best fit for

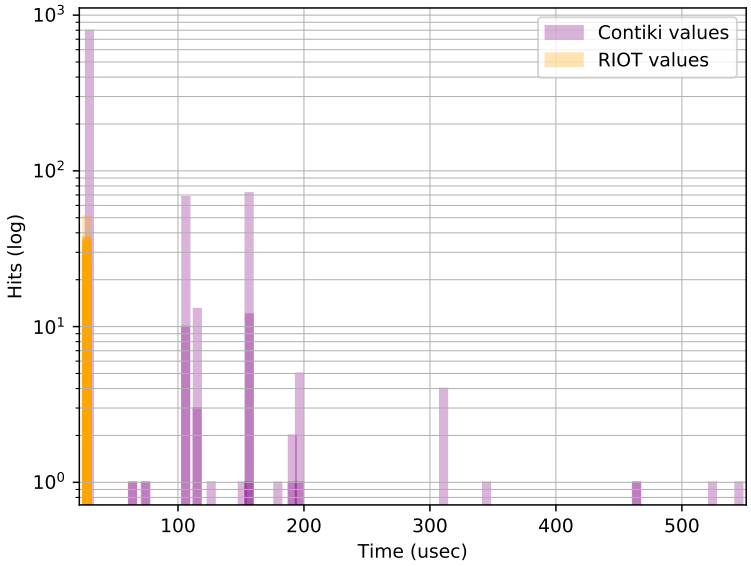

Figure 4.6: distribution comparison between RIOT and Contiki with the reference value on the Z1 board

our benchmarking framework.

The figure 4.7 and the figure 4.8 show the comparison between the first and the second approach on the RE-mote and the Z1 boards with Contiki. With a large distribution, it is difficult to see the difference between the reference distribution and the devices approach measurements distribution. However, we can see that the distribution of the extension approach measurements does not match the distribution of our reference measurements.

The figure 4.9 and the figure 4.10 show the comparison between the first and the second approach on the RE-mote and the Z1 boards with RIOT. The devices approach is the closest to the reference measurements. As the distribution measurements is narrower with RIOT than with Contiki, the difference is more visible.

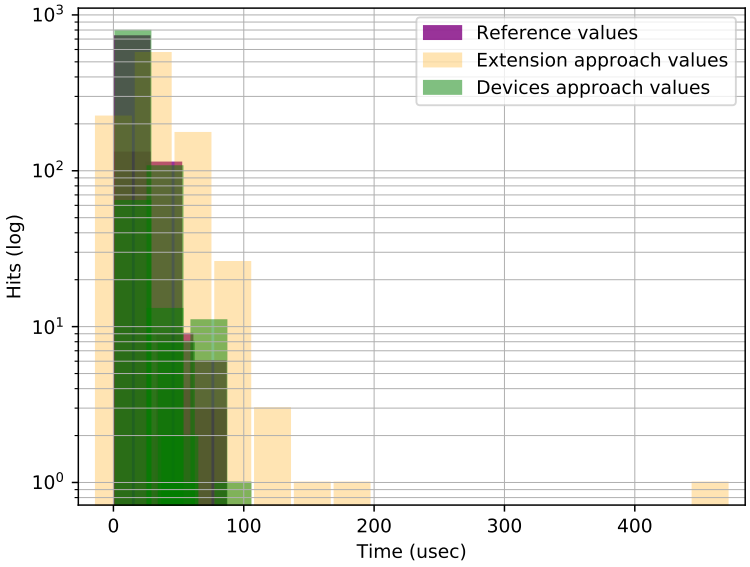

Figure 4.7: comparison of the two approaches with Contiki on the RE-Mote board

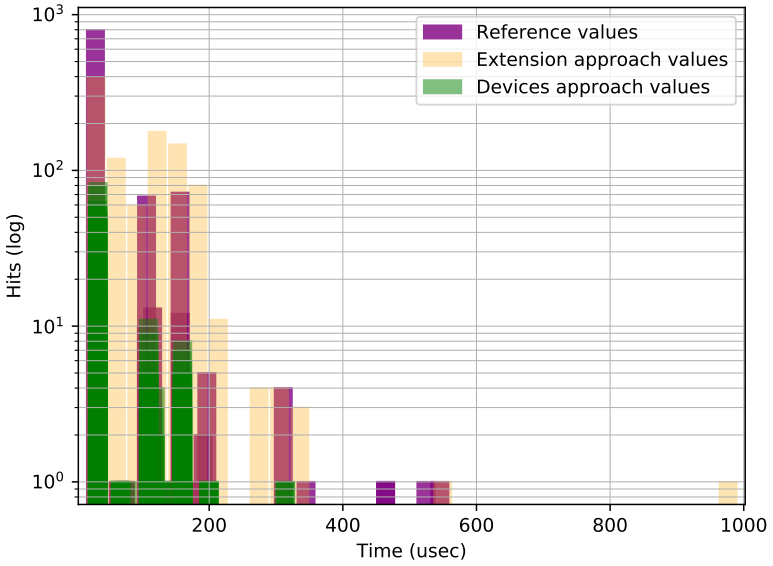

Figure 4.8: comparison of the two approaches with Contiki on the Z1 board

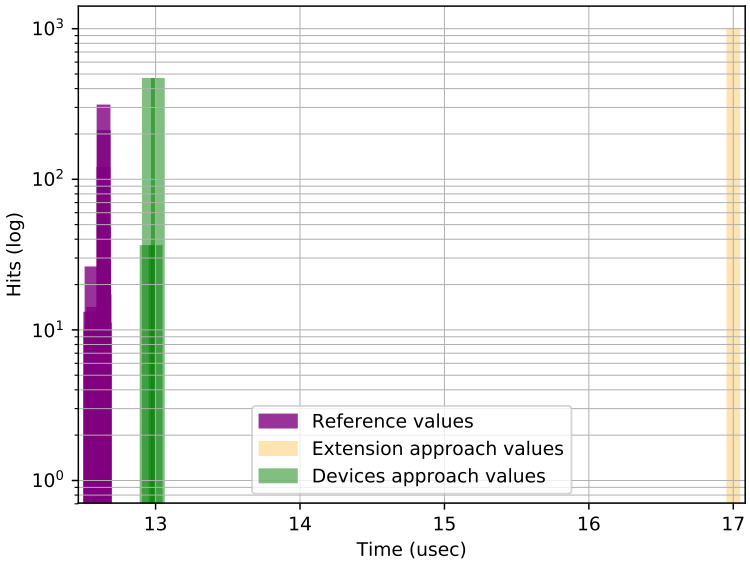

Figure 4.9: comparison of the two approaches with RIOT on the RE-Mote board

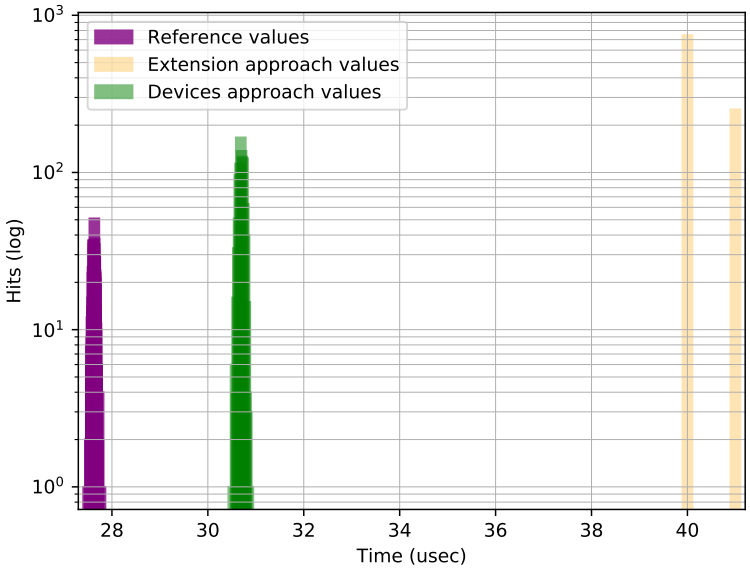

Figure 4.10: comparison of the two approaches with RIOT on the Z1 board

Conclusion and possible improvements

First objective

The first objective of the thesis was to summarize the theory underlying RTOS. We gathered what we consider the important features in RTOS. In some points they are similar to general purpose OS but they often implement specificities that are not found in a regular OS. We presented 3 different embedded OS and compared them from a theoretical point-of-view.

Second Objective

The second objective was to develop a proof-of-concept showing to implementing a framework to benchmark RTOS applications is possible. We had two questions regarding this framework. The first question was to know if it is possible to build a benchmarking framework capable of performing precise measurements without the use of an oscilloscope. The second one was to determine if such a framework could be able to retrieve information useful in the context of RTOS.

We started by developing a kernel-integrated framework but it proved more difficult than expected. This approach was not explored further.

The second approach was to develop a middleware between the RTOS and the application. This method requires modifications in the application for it to call the framework. This approach caused a large overhead due to the serial port. Moreover, it was not able to perform precise measurements.

The third approach was developed with the issues from the second approach in mind. With the help of the PSLab device, which serves as a monitoring device. The PSLab benchmarks the board using GPIO and results are collected with a computer connected to the PSLab. This method brought the best results but an overhead is still observed.

In the end, we can answer our first question in the affirmative. We were able to

implement a framework capable of measuring with enough precision the context switching time of an application running on RTOS.

Going further with the framework

For our second question, improving the framework is possible. With the extension approach, it is possible to store more data in a cache while the application runs. Such information could be the memory usage of the application but also some statistics of the CPU utilization. It would be possible to determine which task runs the most or which interrupt is called the most. Those data stored in the cache would be written on the serial port at appropriate times.

Moreover, the PSLab is a customizable tool that can be used to measure the power consumption of the boards.

List of Figures

1.1 different kernel designs 9
1.2 memory organization 11
1.3 layers around the hardware abstraction layer 17

2.1 Cooja interface 22

1.1 steps in the methodology to compute the real context switching time 31
1.2 interface of the oscilloscope 32
1.3 Zolerta RE-Mote board 34
1.4 Zolerta Z1 board 34
1.5 reference measurements distribution with Contiki on the RE-Mote board 36
1.6 reference measurements distribution with RIOT on the RE-Mote board 36
1.7 reference measurements distribution with Contiki on the Z1 board . 37
1.8 reference measurements distribution with RIOT on the Z1 board . . 37

2.1 benchmarking framework proxy 39
2.2 pings to the framework 42
2.3 activity flow of the framework 43
2.4 Pocket Science Lab device from PSLab.io 46
2.5 interaction schema of the devices used in the framework 47
2.6 measurement of the context switching time with a single GPIO . . . 48

3.1 overhead reference measured by the oscilloscope 54
3.2 extension approach overhead measured by the oscilloscope 54
3.3 devices approach overhead measured by the oscilloscope 54
3.4 devices approach measurements distribution with Contiki on the RE-Mote board 56
3.5 devices approach measurements distribution with RIOT on the RE-Mote board 57

3.6 devices approach measurements distribution with Contiki on the Z1 board . . . 58
3.7 devices approach measurements distribution with RIOT on the Z1 board . . . 59

4.1 distributions comparison with Contiki on the RE-Mote board 63
4.2 distributions comparison with Contiki on the Z1 board 63
4.3 distributions comparison with RIOT on the RE-Mote board 64
4.4 distributions comparison with RIOT on the Z1 board 64
4.5 distribution comparison between RIOT and Contiki with the reference value on the RE-Mote board 65
4.6 distribution comparison between RIOT and Contiki with the reference value on the Z1 board 66
4.7 comparison of the two approaches with Contiki on the RE-Mote board 67
4.8 comparison of the two approaches with Contiki on the Z1 board . . 67
4.9 comparison of the two approaches with RIOT on the RE-Mote board 68
4.10 comparison of the two approaches with RIOT on the Z1 board . . . 68

List of Tables

1.1 classes of constrained devices 16

2.1 RTOS comparisons 24

1.1 reference value for Contiki and RIOT on the RE-Mote 35
1.2 reference value for Contiki and RIOT on the Z1 35
1.3 resume of the reference values 38

3.1 extension approach measurements for Contiki and RIOT on the RE-Mote board 54
3.2 extension approach measurements for Contiki and RIOT on the Z1 board 55
3.3 devices approach measurements for Contiki and RIOT on the RE-Mote board 56
3.4 devices approach measurements for Contiki and RIOT on the Z1 board 57
3.5 Framework measurements resume on the RE-Mote board 57
3.6 Framework measurements resume on the Z1 board 58

Bibliography

[1] United States Naval Academy. *User Space, Kernel Space, and the System Call API*. URL: https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/11/lec.html (visited on 06/08/2019).

[2] Emmanuel Baccelli et al. "RIOT: One OS to rule them all in the IoT". PhD thesis. INRIA, 2012.

[3] Richard Barry. *FreeRTOS's tick suppression saves power | Embedded*. URL: https://www.embedded.com/electronics-blogs/industry-comment/4414162/1/FreeRTOS-s-tick-suppression-saves-power (visited on 06/08/2019).

[4] C. Bormann, M. Ersue, and A. Keranen. *Terminology for Constrained-Node Networks*. RFC 7228. http://www.rfc-editor.org/rfc/rfc7228.txt. RFC Editor, May 2014. URL: http://www.rfc-editor.org/rfc/rfc7228.txt.

[5] *contiki-os/contiki*. [Online; accessed 8. Jun. 2019]. June 2019. URL: https://github.com/contiki-os/contiki/wiki/Processes.

[6] A. Dunkels, B. Gronvall, and T. Voigt. "Contiki - a lightweight and flexible operating system for tiny networked sensors". In: *29th Annual IEEE International Conference on Local Computer Networks*. Nov. 2004, pp. 455–462. DOI: 10.1109/LCN.2004.38.

[7] Adam Dunkels. *Adam Dunkels*. URL: http://dunkels.com/adam/ (visited on 06/08/2019).

[8] Adam Dunkels. *adamdunkels/uip: The historical uIP sources*. URL: https://github.com/adamdunkels/uip (visited on 06/08/2019).

[9] Adam Dunkels. *Contiki: Bringing IP to Sensor Networks*. URL: https://ercim-news.ercim.eu/en76/rd/contiki-bringing-ip-to-sensor-networks (visited on 06/08/2019).

[10] Adam Dunkels, Oliver Schmidt, and Thiemo Voigt. "Using Protothreads for Sensor Node Programming". In: *In Proceedings of the REALWSN 2005 Workshop on RealWorld Wireless Sensor Networks*. 2005.

[11] Aniket Dusey. *Interprocess Communication: Methods*. URL: https://www.geeksforgeeks.org/interprocess-communication-methods/ (visited on 06/08/2019).

[12] *FeuerWhere*. [Online; accessed 9. Jun. 2019]. July 2011. URL: http://www.mi.fu-berlin.de/inf/groups/ag-tech/projects/feuerwhere/index.html.

[13] FreeRTOS. *https://www.freertos.org/taskandcr.html*. URL: https://www.freertos.org/taskandcr.html (visited on 06/08/2019).

[14] *FreeRTOS open source licensing, FreeRTOS license description, FreeRTOS license terms and OpenRTOS commercial licensing options*. URL: https://www.freertos.org/a00114.html (visited on 06/08/2019).

[15] Padmini Gaur and Mohit P Tahiliani. "Operating systems for IoT devices: A critical survey". In: *2015 IEEE Region 10 Symposium*. IEEE. 2015, pp. 33–36.

[16] Rich Goyette. "An analysis and description of the inner workings of the freertos kernel". In: *Carleton University* 5 (2007).

[17] Matthew R Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE. 2001, pp. 3–14.

[18] O. Hahm et al. "An Experimental Facility for Wireless Multi-hop Networks in Future Internet Scenarios". In: *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*. Oct. 2011, pp. 48–57. DOI: 10.1109/iThings/CPSCom.2011.114.

[19] University of Illinois at Chicago. *Operating Systems: CPU Scheduling*. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/5_CPU_Scheduling.html (visited on 06/08/2019).

[20] Syed Muhammad Zeeshan Iqbal, Yuchen Liang, and Hakan Grahn. "Parmibench-an open-source benchmark for embedded multiprocessor systems". In: *IEEE Computer Architecture Letters* 9.2 (2010), pp. 45–48.

[21] Arm Ltd. *Cortex-M3 – Arm Developer*. [Online; accessed 9. Jun. 2019]. June 2019. URL: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m3.

[22] *MSO56*. [Online; accessed 9. Jun. 2019]. June 2019. URL: https://www.tek.com/oscilloscope/mso56.

[23] *MSP430F2617 16-Bit Ultra-Low-Power MCU, 92KB Flash, 8KB RAM, 12-Bit ADC, Dual DAC, 2 USCI, HW Mult, DMA | TI.com*. [Online; accessed 9. Jun. 2019]. June 2019. URL: http://www.ti.com/product/MSP430F2617.

[24] Arslan Musaddiq et al. “A survey on resource management in IoT operating systems”. In: *IEEE Access* 6 (2018), pp. 8459–8482.

[25] C. Paasch. O. Bonaventure G. Detal. *Systèmes informatiques*. URL: https://sites.uclouvain.be/SystInfo/notes/Theorie/html/C/malloc.html#organisation-de-la-memoire (visited on 06/08/2019).

[26] *Pull Request Policy · contiki-os/contiki Wiki*. URL: https://github.com/contiki-os/contiki/wiki/Pull-Request-Policy (visited on 06/08/2019).

[27] Rizwan Hamid Randhawa, Adeel Ahmed, and Muhammad Ibrahim Siddiqui. “Power Management Techniques in Popular Operating Systems for IoT Devices”. In: *2018 International Conference on Frontiers of Information Technology (FIT)*. IEEE. 2018, pp. 309–314.

[28] RIOT. *RIOT Community Processes · RIOT-OS/RIOT Wiki*. URL: https://github.com/RIOT-OS/RIOT/wiki/RIOT-Community-Processes (visited on 06/08/2019).

[29] Challouf Sabri, Lobna Kriaa, and Saidane Leila Azzouz. “Comparison of IoT constrained devices operating systems: A Survey”. In: *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*. IEEE. 2017, pp. 369–375.

[30] *Scheduler*. [Online; accessed 8. Jun. 2019]. June 2019. URL: https://riot-os.org/api/group__core__sched.html.

[31] The Unix Heritage Society. *Introduction to Operating Systems*. 2012. URL: https://minnie.tuhs.org/CompArch/Lectures/week07.html (visited on 06/08/2019).

[32] Dr. Apurva Shah Vatsalkumar H. Shah. “An Analysis and Review on Memory Management Algorithms for Real Time Operating System”. In: *International Journal of Computer Science and Information Security (IJCSIS)* 14.5 (2016).

[33] Tao Xu et al. “Performance benchmarking of FreeRTOS and its Hardware Abstraction”. In: *Unpublished doctoral thesis, Technical University of Eindhoven* (2008).

[34] Ming-Yuan Zhu. “Understanding FreeRTOS: A requirement analysis”. In: *CoreTek Systems, Inc, Beijing, China, Technical Report* (2016).

[35] *Zolertia/Resources*. [Online; accessed 9. Jun. 2019]. June 2019. URL: https://github.com/Zolertia/Resources/wiki/RE-Mote.

[36] *Zolertia/Resources*. [Online; accessed 9. Jun. 2019]. June 2019. URL: https://github.com/Zolertia/Resources/wiki/The-Z1-mote.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**