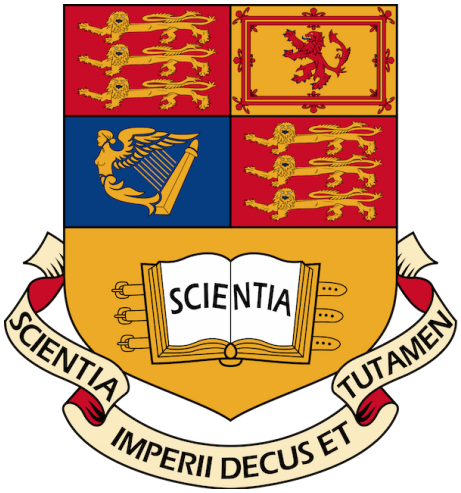
Imperial College London

Department of Computing



Operating system for Internet of Things

by

Hubert de TAFFANEL de LA JONQUIERE

Submitted in partial fulfillment of the requirements for the MSc Degree in Computing (Software Engineering) of Imperial College London.

September 2015

Abstract

Everyday objects become increasingly connected between them and to the Internet. The objective of these connected devices, also called Internet of things, is to improve our daily life. Apple, Google, Samsung and many other companies developed connected devices in order to make our home, our watch, our cars smarter. However, in this environment, devices may have limited computing and energy resources.

In this project, a real-time operating system for Internet of Things, developed in C, is presented. This operating system, called Mongoose, works with limited computing resources and has real time capabilities. More precisely, Mongoose is developed to run on an Arduino Uno board with 16MHz microcontroller, ATMega328p, and 2KB of RAM. This operating system is capable of running concurrently multiple applications (up to 3). More specifically, this project focuses on multitasking and inter process communication abilities of Mongoose. Although this operating system is designed for Internet of Things, it does not integrate yet services like Internet modules.

Mongoose's development environment should be developer friendly. In fact, we try, by this environment, to reduce the learning curve and make Internet of things development more accessible. For this reason, Mongoose's drivers have been developed to provide a clear and consistent API. In addition, applications for Mongoose can be developed with a subset of Python, C or C++ via any text editor. For Python applications, a parser, which has been implemented in this project, translate them in C. By this way, this development environment keeps the syntax simplicity of Python and the speed of C.

Acknowledgment

First, I would like to thank my supervisor Dr Benny Lo for letting me working on this project, for his time, his indispensable guidance and help. Furthermore, I would like to express my gratitude to Dr Andrea Gaglione for helping me in the implementation and for answering my questions.

I would like to thank my parents and my brothers and sister who have supported me during my studies and their invaluable help.

I would like to thank my father and Dr Jean-Philippe Poli for transmitting me the taste of Computing.

Finally, I would like to express my gratitude to Jan Povala and Hélène Bories for reading this thesis and for giving helpful feedback.

Contents

Chapter 1

Introduction

Sensor nodes, vehicles, home devices will be interconnected and connected to the internet in the future. Network of these diverse, connected devices is called Internet of Things (IoT). IoT technologies can potentially change our lives as in the past with the internet and mobile technologies. However, IoT corresponds to a large spectrum of devices from 8-bit microcontrollers to energy-efficient 32-bit processors. These devices are expected to "fulfill the requirements of cyberphysical systems" [1]. To achieve that, these devices will have to be resource constrained in terms of computing power, energy, memory and communication.

An operating system for IoT has to be reliable, has to have a real time behaviour and also provide an adaptive "communication stack to integrate the internet seamless" [1]. Also this operating system has to be portable on a large range of hardware.

Google, Intel and other technology companies are developing operating systems with a small footprint in order to fulfill these requirements. Furthermore, operating systems such as TinyOS and Contiki have been developed for lightweight devices. However, they are too complicated to use.

1.1 Context

Research on Wireless Sensor Network (WSN) started around 1980 [3]. In this purpose (WSN), operating systems, like TinyOS and Contiki, have been developed for sensor nodes. Contrary to sensor nodes which mainly use radio communication, IoT is based on connecting sensor devices to the internet. The concept of IoT becomes a major concern. Indeed, the number of devices has grown over the past years regarding to the population, as shown in figure 1.1. Aim of IoT is to build distributed machine-to-machine (M2M) applications [4]. More precisely, Iot will have many applications such as smart parking, air pollution, potable water monitoring, etc [6].

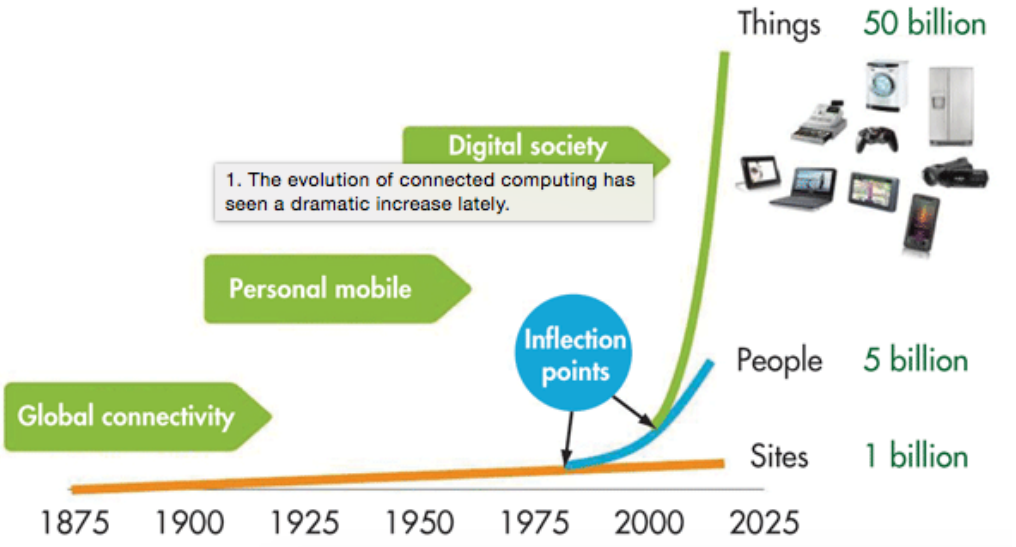


Figure 1.1: Connected devices [2]

1.2 Requirements

The number of smart devices increases every year. Hence, IPv4 addresses are not anymore sufficient. Nonetheless, IPv6 will provide enough addresses for billions of new sensors and devices. The deployment of IPv6 is a key element for the development of IoT. Moreover, these smart devices need to be self-sustaining or at least to have a long useful life. So energy consumption is also a key element for IoT [2].

Operating systems for IoT have to fulfill these requirements:

- energy efficient
- available for a large range of hardware
- connectivity by WiFi, Bluetooth, USB, ...
- small memory footprint
- developer friendly
- modularity

1.3 Objectives

The goal of this project is to build an operating system for IoT working on an Arduino Uno board. With this operating system, developers will be able to create multiple applications, which are running concurrently, in Python, C and C++. This operating system will be

evaluated regarding its memory footprint and its real-time performances.

1.4 Contribution

We have produced a real-time operating system working on Arduino Uno board with an 8-bit microcontroller and 2KB of RAM. This operating system, Mongoose, is designed to run multiple applications with a round robin scheduling policy. Additionally, the kernel period used by the scheduler, also called quantum, has been chosen regarding real-time performances. Mongoose provides an inter process communication mechanism too. This mechanism lets applications communicate through channels.

Moreover, a development environment has been designed for this operating system. In this environment, applications can be developed in a subset of Python, C and C++ via any text editor. In order to have python application with Mongoose, we have developed a parser which translates these applications in C. Therefore this environment keeps the syntax simplicity of Python and the speed of C. Finally, a clear and consistent API for hardware drivers and operating system services is implemented.

1.5 Project outline

Firstly, this report starts with a state of the art of existing operating systems for resource constrained environment. Secondly Mongoose, a real-time operating system developed in this project, is presented. Thirdly, the use of Python with this operating system is explained. Finally, examples with multiple python applications are compared to ArdOS, a real-time operating system available on Arduino Uno board.

Chapter 2

Operating Systems for embedded systems

2.1 Arduino and ArdOs

Arduino is an "open-source prototyping platform based on easy-to-use hardware and software" [7]. Multiple libraries are available to access peripheral devices like sensors, buttons, motors, LEDs, etc. The development of applications on Arduino boards can be done using Arduino IDE. The main goal of this platform is to let people without programming and electronic background working on these boards [7].

Arduino is cross-platform (with the IDE) and offers a simple and accessible development environment for beginners and flexible enough for advanced users. Software and hardware can be extended. This environment is also suitable for IoT [7].

No operating system on Arduino Uno board is provided by Arduino. However, multiple operating systems have been developed over the past years. We will focus on ArdOS which will be used as a comparison for Mongoose in chapter 5.

ArdOS is an operating system working on "the Arduino series of boards based on the ATmega168/328/1280/2560 series of microcontrollers" [8]. It is a real-time operating system which can be used with Arduino libraries and Arduino IDE. This operating system uses a priority based scheduler with inter task communication and coordination mechanism [8].

Two communication mechanisms are available: FIFO message queues and prioritized message queues. The coordination mechanisms used are: binary and counting semaphores, mutex locks and conditional variables [8].

"The design philosophy behind ArdOS is to strike a balance between ease-of-use, compactness, and sound real-time design principles" [8].

2.2 Contiki

Contiki is an open source operating system for sensor devices. This OS has an event-driven kernel and supports preemptive multi-threading. Generally this operating system needs 1kB in RAM and 30kB in ROM [9].

2.2.1 Overview

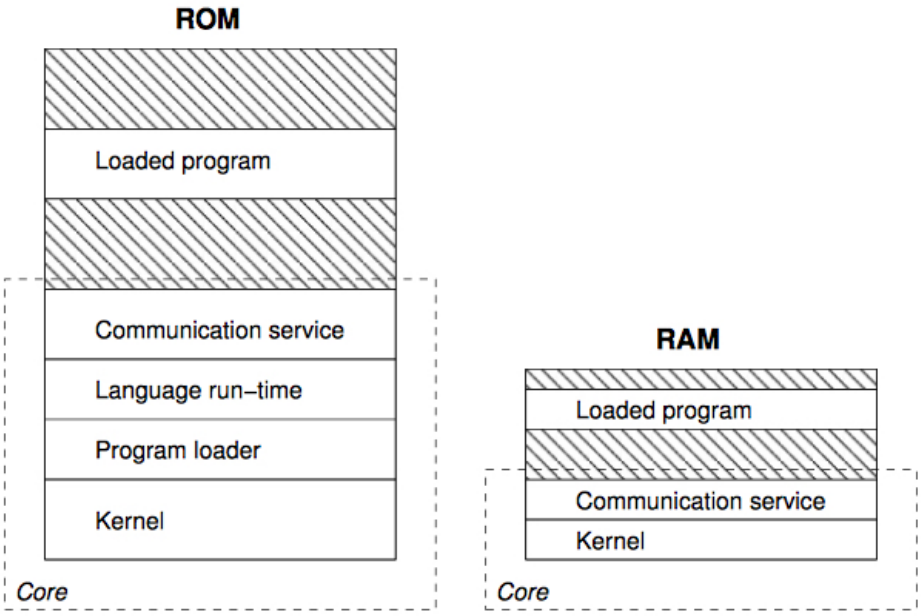


Figure 2.1: Contiki [10]

Contiki is composed of a kernel, libraries, processes and a program loader. Processes correspond to user programs or services which can be called by some user programs. Processes communicate together through the kernel and can access to the hardware layer. It is also the case for device drivers. In fact, the kernel does not provide "a hardware abstraction layer" [10]. Contiki has two parts: the core and loaded programs. The core is composed of the kernel, the program loader, libraries and communication services. It is compiled into a binary image before the deployment of Contiki. It can be changed after deployment thanks to a special bootloader. This situation does not happen in general. However, applications can be downloaded through the network or directly installed. Hence, they become accessible to the program loader [10].

2.2.2 Kernel Architecture

Contiki has an event-driven kernel in order to provide concurrency without multi-threading which is resource consuming. Indeed, multi-threading mechanism needs a stack per thread and locking mechanisms in order to avoid concurrent threads to access shared memory [10].

In this event-driven model, processes can share the same stack because they are implemented as event handlers which run to completion and cannot block. This means a process is scheduled when an event destined to this process occurs. More precisely, its event handler is triggered with this event [9]. Two event handlers cannot run in concurrency. Thus, no locking mechanisms are needed [10].

Moreover, the kernel and the applications share the same address space. Therefore the applications can access to global data. However, this means also that one application can corrupt other applications and also the kernel. The main advantage of this design is the memory management. Indeed, the applications and the kernel can effectively share global information [9].

2.2.3 Events

As said before, Contiki is based on events. Each application corresponds to a process. The process is related to an event handler which is triggered when an event occurs. Processes interact with each other through events or global data.

Firstly, events can have one destination or be sent to all processes. In addition, events can be issued by the kernel and also processes but the kernel cannot receive an event. The kernel can only send an event when it is initiated, stopped or polled. On the contrary, a process can send an event at anytime.

Secondly, events can be asynchronous or synchronous. When an asynchronous event occurs, it is added to the kernel's event queue. These events will be scheduled by the operating system in a FIFO (First In First Out) order. When the event is delivered, the process's event handler is called. This mechanism is a "deferred procedure" [9]. However, when a synchronous event occurs, the original process is resumed after the message is delivered. This mechanism is an immediate procedure.

2.2.4 Preemptive multi-threading

Multi-threading feature is implemented as a library and only linked to applications which require it [10]. It uses protothreads which are stackless and lightweight threads. Indeed, Contiki needs only two bytes per protothread and there is no extra stack for a thread [11]. Protothreads are implemented as an event handler and a local continuation. The first part is associated with the process. With a location continuation, the protothread's context can be stored and restored when it is needed to be blocked. Indeed, an event handler in Contiki runs to completion but an event handler can be blocked for a protothread. A local continuation means that the local variables and call history are preserved [9].

2.2.5 File system

Contiki uses Coffee file system for flash-based sensor node. It uses a small and constant footprint per file. File modifications are not stored as a spanning log structure. It uses micro logs instead. Coffee is based on a page structure. It allocates a predefined amount of pages per file. If it is not enough, a greater space is created for the file and the data are copied into it. Coffee uses a first fit algorithm for page allocation and a garbage collector to manage obsolete pages [11].

2.2.6 Software update

Contiki has the ability to download an application through the network. This feature is important in sensor network because collecting all sensor nodes is not always possible. Contrary to many OS for embedded devices, Contiki does not need to download a complete binary image which contains the OS, libraries and the new application. In fact, Contiki can load and unload applications at run-time. Therefore, the number of transmitted bytes is generally reduced and it is less energy consuming [10].

2.3 TinyOS

TinyOS is an open source operating system for sensor network. It was developed by the University of California in Berkeley. TinyOS uses an "event-driven programming model". This operating system is based on components which are in NesC, a programming language derived from C [12]. Concurrency is implemented with non-preemptive tasks.

2.3.1 Overview

The operating system and the applications are based on components which have three computational abstractions: commands, events and tasks. All of them are C functions [13].

- Commands are requests sent to a component in order to execute a feature.
- Events are generated during computation of a component command. They have a higher priority than tasks and can preempt each other.
- Tasks are deferred procedures. They are not executed immediately. Tasks are added to a FIFO scheduler and they will be executed at a later time. When they are posted, the current running program is not stopped.

2.3.2 Architecture

TinyOS has a monolithic architecture as Linux and is an event-driven operating system. This operating system provides a single shared stack and is based on a component model which will be described later. Kernel and user address spaces are not separated.

2.3.3 Component model

There are two kinds of components: modules and configurations. These components are written in NesC, a dialect of C. Firstly, a module is a set of interfaces with commands and events. They have "private state variable and data buffer which only it can reference" [13].

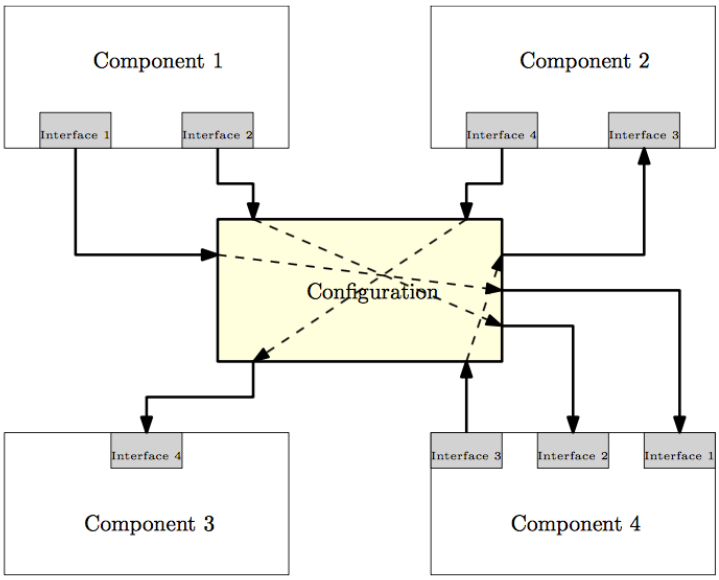


Figure 2.2: Wiring in TinyOS [12]

Secondly, configurations specify interfaces provided by components and also which other components they need. Connections between components are called wiring. The figure 2.2 shows an example of configuration. Components can be built in a hierarchical structure. More precisely, high level components can use low level components [12].

2.3.4 Events and commands

Commands and events correspond to a "bounded amount of work" [16]. An event is signaled when an action occurs. It is executed immediately and does not block. Hardware interrupts send low-level events [16]. A command is a request to a component to start an amount of work. It returns immediately. When the action is finished, an event is sent.

2.3.5 Interfaces

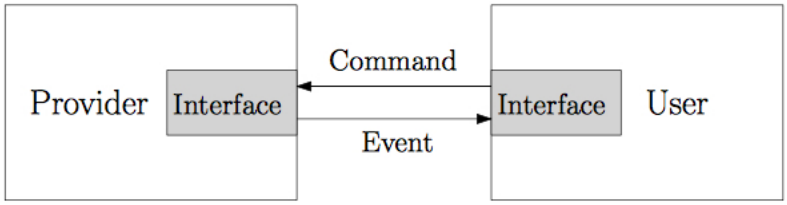


Figure 2.3: Interface in TinyOS [12]

Figure 2.3 presents how events and commands are implemented with interfaces. Indeed, a component specifies which events and commands that they handle through interfaces. As shown in figure 2.3, commands are sent from a user program and the components can send events when it completes a command. Therefore, commands and events are inter component communications and interfaces are bidirectional [13].

2.3.6 Tasks and scheduling

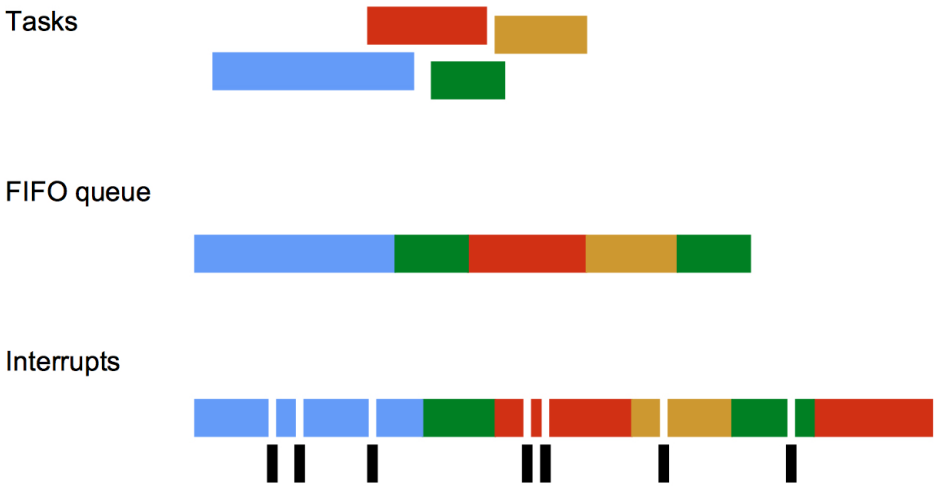


Figure 2.4: Concurrency in TinyOS [16]

TinyOS has only one thread with a shared stack and no heap. Tasks in TinyOS are deferred procedures. They can be used for a large amount of computation. Tasks are added to a FIFO scheduler when they are posted. When tasks are executed, they run to completion. However, they can be preempted by events and interrupts but never by an other task. Finally, tasks can be posted by events and commands and also can use commands and events [15].

2.3.7 Memory Management

TinyOS runs on embedded systems which have a very small amount of memory. There is also no memory management hardware. This kind of environment is not suitable for a dynamic memory allocation. Also NesC does not support dynamic memory allocation. Applications are a set of components which are compiled together. At build-time, the linker determines the amount of memory needed. So, this approach is static [14].

2.3.8 File system

TinyOS provide a single-level file system. Indeed, generally only one application runs. Therefore, a single-level file system is sufficient considering a scarce memory [11].

2.3.9 Mutlithreading

Tinyos concurrency model has two levels: tasks (synchronous) and interrupts (asynchronous). Interrupts can preempt tasks and tasks run to completion. TOSThreads add a third level of concurrency. This approach is composed of a kernel thread, a thread scheduler, the TinyOS task scheduler, application threads and an API for system calls [17].

However, TOSThreads need two changes in TinyOS code: a boot sequence modified and "a post-amble for every interrupt handler" [17]. The boot sequence encapsulates TinyOS in a single high priority "kernel-level thread" [17] before it boots. The second change ensures that TinyOS runs when an interrupt handler posts a task. This change decreases a little bit the TinyOS performances.

The kernel thread runs TinyOS scheduler which manages tasks. In addition, thread scheduler manages concurrency between application threads. Each thread has a fixed size stack which does not grow over time. Application threads use standard primitives: barriers, condition variable, mutexes and semaphores. They are used for synchronizing application threads. When an application thread needs a system call, it sends a message to the kernel thread thanks to a task. The system call is executed and the active application is paused because the kernel thread has a high priority. When the kernel thread goes back to sleep (TinyOS is not blocking), the thread scheduler gets the control and resumes the application thread [17].

2.4 Riot

Riot is an operating system for IoT in order to bridge the gap between wireless sensor network operating system and traditional operating system using internet (Windows, Mac OS X, Linux).

2.4.1 Overview

Riot uses a modular architecture based on a microkernel. This operating system has small footprint: 1.5KB RAM and 5KB ROM. Riot is written in ANSI C. It supports real multithreading and also application written C++ [1]. Riot has a developer friendly API [18].

2.4.2 Architecture

Riot is based on a microkernel inherited from FireKernel. Therefore it supports FireKernel features and real multithreading. Contrary to a monolithic kernel, device drivers and file system can crash without crashing the whole system. It is due to a microkernel architecture. Riot provides also a TCP/IP network stack [18]. This operating system provides a preemptive multithreading thanks to a priority based-scheduler. This scheduler is considered as a tickless scheduler because there is no periodic event [1]. Riot connects constrained devices to internet using 6LoWPAN which makes it IoT-ready [18].

2.4.3 Limits

Riot project started in 2008 and was published public in 2013 [19]. This operating system is therefore very new. That is why, there are not enough technical details on this operating system. Riot is generally presented as the next generation of operating system for IoT. However, we have not found any concrete implementation details.

2.5 Brillo

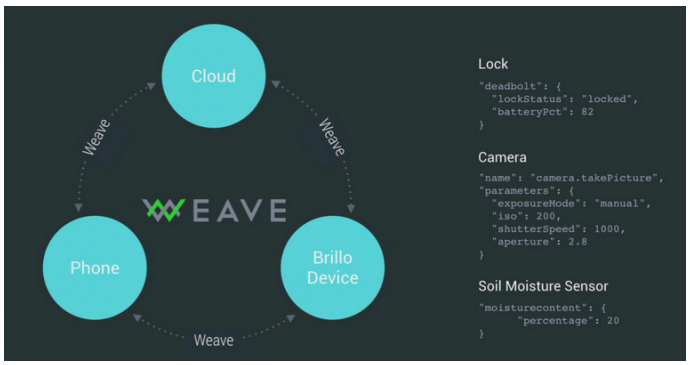


Figure 2.5: Brillo and Weave by Google [21]

Brillo is a platform developed by Google for IoT. It was developed after Google bought Nest, a company working on IoT, in 2014. It will be released in 2015. Brillo will use a protocol, Weave, based on JSON for communication between IoT, mobile devices and the cloud [21]. Google will extend Android platform to IoT.

2.6 Comparison

	Tinyos	Contiki	Riot	ArdOS
Architecture	Monolithic kernel	Modular architecture	Microkernel architecture, modularity (RTOS)	Microkernel architecture (RTOS)
Scheduling	Event driven, FIFO strategy	Event driven, FIFO strategy	Priority-based scheduling	Priority-based scheduling
Mutlitasking	TOSThreads, partial multi-threading	Protothreads, partial multi-threading	Real multithreading inherited from FireKernel	Multitasking between tasks
Programming Language	NesC dialect of C	Subset of C	C and C++	C and C++
Min RAM	<1 kB	<2 kB	1.5 kB	<0.5 kB
Min ROM	<4 kB	<30 kB	5 kB	<3 kB

Table 2.1: Comparison between existing operating systems

Chapter 3

Mongoose Operating system

3.1 Motivation

An operating system provides a model of the board for applications. The two main purposes of an operating system are to manage all resources available with the board and provide a clear abstraction of these resources [23].

This abstraction makes writing applications faster, easier and more maintainable [24]. Multiple applications can run concurrently in most of the operating systems. This ability is called multi-tasking. In fact, each processor core can run a "single thread of execution" [24]. Multi-tasking is therefore an illusion of concurrency. The scheduler, an operating system component, is responsible for choosing which application to execute and at what time. This multi-tasking effect is possible by switching between applications thanks to the scheduler.

The scheduler defines the type of an operating system. For instance, "a multi user operating system (such as Unix) will ensure each user can get a fair amount of the processing time" [24]. On the contrary, desktop operating system will try to stay responsive for the user [24].

Embedded systems have real-time constraints. Therefore, previous examples are not suitable for such environment. As a matter of fact, responsive time to an event has to be limited. Therefore, the scheduler should have a predictive behaviour and thus built with a deterministic approach [24]. Operating systems with this kind of scheduler are called Real Time Operating System (RTOS).

A real time operating system has these requirements [25]:

- Temporal requirements (time constraints)
- Functional requirements (behaviour)

- Small memory footprint
- Deterministic amount of time for RTOS services

We choose to design Mongoose as a real time operating system in order to respect time requirements with a low memory footprint.

3.2 Overview

Mongoose is a real time operating system working on Arduino Uno board with ATMega328p and 2KB of RAM.

This operating system is composed of a scheduler, an inter process communication mechanism, hardware drivers and a hardware event handler. This operating system is capable of running up to 3 applications at the same time.

Mongoose overall architecture is represented by the figure 3.1

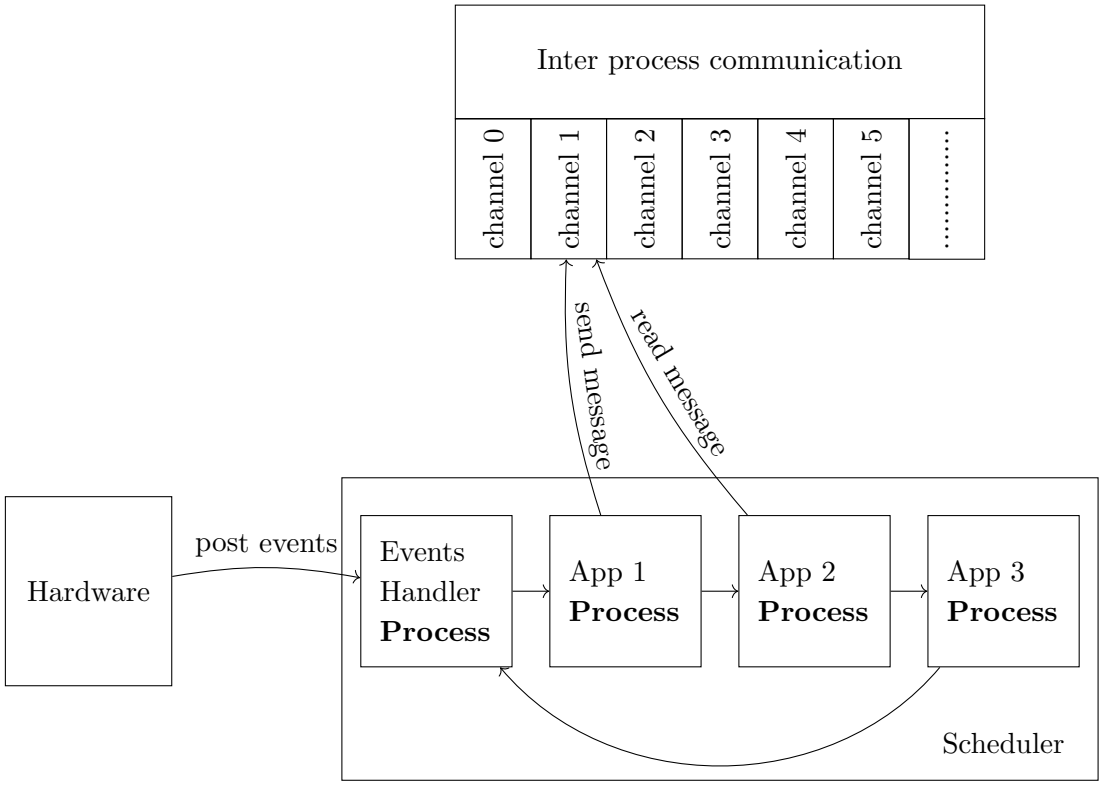


Figure 3.1: Mongoose Architecture

Applications and the event handler are considered as processes with separated stacks. Every application and event handler process have the same priority. When a hardware event occurs, such as a pressed button, event handler process is woken up and has the same priority as applications. If no event is in the event handler process, this process will

not be executed by the scheduler. Time deadlines for an application, expressed as delays, will stop application's execution in order to give to other applications more executing time until the delay is finished.

Each process has its own stack and is not blocking. In fact, Mongoose uses a time sharing round robin scheduler. Each process has a cyclic fixed executing time. However, an action related to an event cannot be preempted by other events. The scheduler will be explained in more details later in this chapter.

3.3 Arduino Uno: Hardware

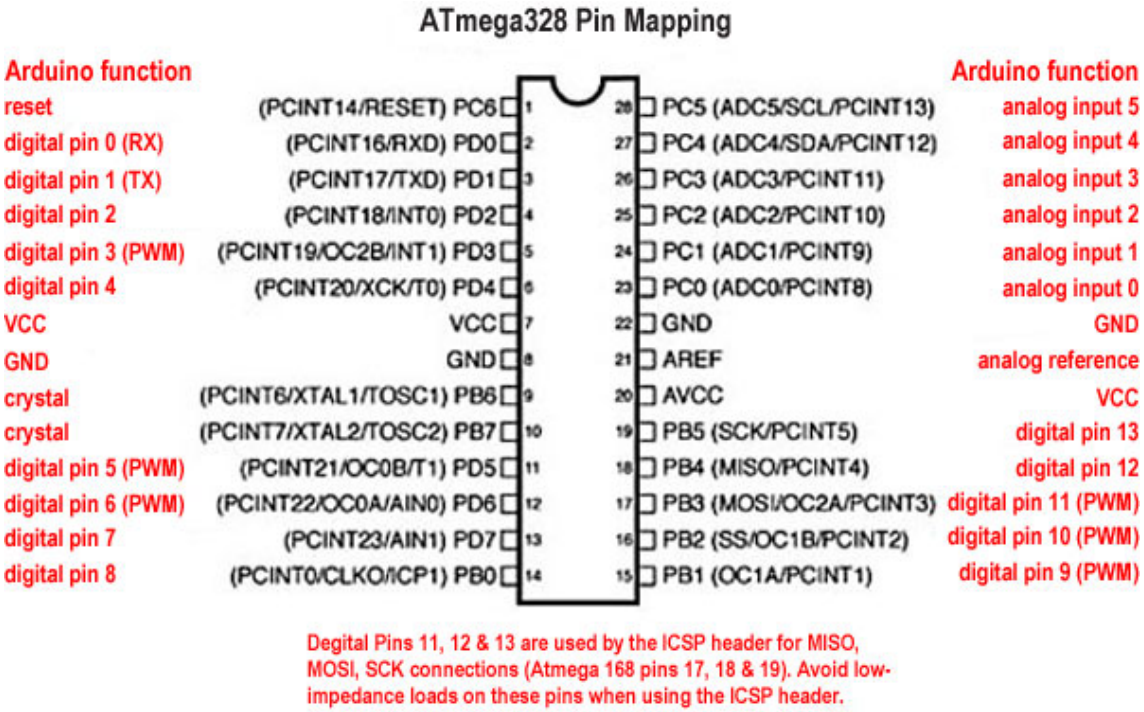


Figure 3.2: Pin mapping ATMega328 [26]

Arduino Uno R3 is a microcontroller board with an ATmega328p. This board has a 16MHz crystal oscillator, 14 digital pins (input/output) and 6 analog pins.

ATMega328p is a microcontroller which has Harvard architecture with 32 KB of flash memory, 2KB of SRAM and 1 KB of EEPROM [27]. It has 32 general purpose working registers (memory space within CPU) [30]. This microcontroller uses the PicoPower technology which needs less power than ATMega328 [31].

Flash memory contains the source code and the bootloader (0.5 kB). SRAM is a static RAM. More precisely, data is hold until the power is off [27]. SRAM is faster than DRAM, dynamic RAM, which needs to refresh periodically the data [44]. Finally, EEPROM is an Electrically Erasable Programmable Read-Only Memory which can hold data even if the

power is off [29].

Figure 3.2 shows the pin mapping of ATMega328. It is the same for ATMega328p. This diagram is used in order to develop hardware drivers.

Mongoose is designed to fit in ATMega328p SRAM and ATMega328p flash memory but EEPROM is not used by this operating system.

3.4 Kernel Architecture

An operating system has two parts: the kernel space and the user space. The two main kernel architectures are: monolithic kernel and microkernel. In this section, these approaches and Mongoose kernel architecture will be presented

3.4.1 Monolithic kernel

In the monolithic kernel, all basic features of an operating system are in the kernel space as shown in the figure 3.3. More precisely, memory management, drivers for hardware, process management, file system, system services are in kernel mode. For IoT, some of these features like file system are not present. Applications are the only part of the system which is running in user mode.

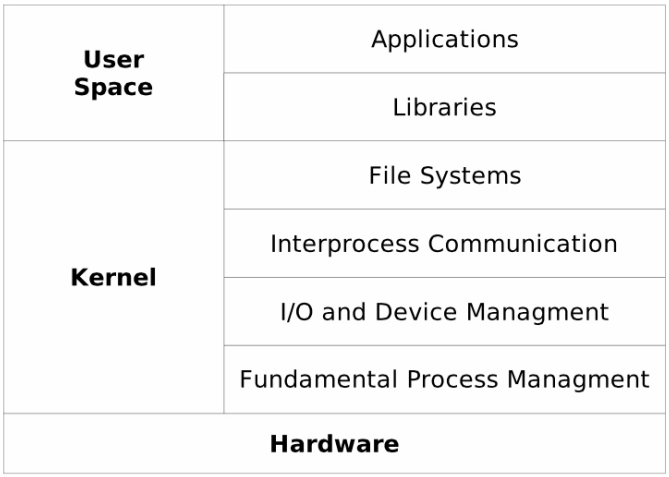


Figure 3.3: Monolithic kernel [22]

This kernel is implemented with one process and a single kernel address space [22]. In this architecture, kernel size can be an issue for IoT which can have limited resources.

3.4.2 Microkernel

A Microkernel architecture only has inter process communication, memory management and scheduling in kernel mode. Therefore, applications are no longer the only part in user mode.

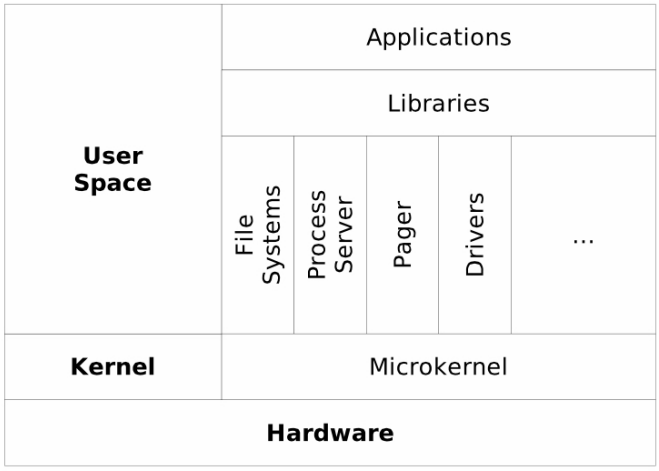


Figure 3.4: MicroKernel [22]

The kernel size can be very small and it can fit in the processor's first level cache [22].

3.4.3 Mongoose Kernel

For Mongoose, we have chosen the microkernel approach in order to minimize the memory footprint.

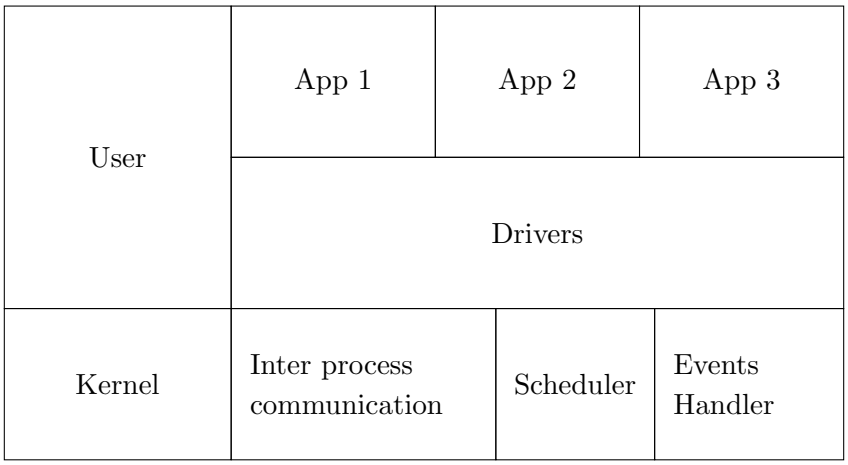


Figure 3.5: Mongoose kernel

As shown in figure 3.5, this architecture is split in two: kernel and user space. On one hand, the kernel contains inter process communication, event handler and scheduling. On the other hand, user space contains drivers and applications.

Applications and event handler are considered as processes and are in the scheduler queue (figure 3.1). Despite of that, the event handler process is not in the user space. In fact, this process is accessible by all applications through posted events. In addition, the event handler process needs to call kernel services which are not accessible by applications. For these reasons, this process was implemented in the kernel space.

Finally, the inter process communication mechanism is also in the kernel space. More precisely, this mechanism needs also kernel services, which are not available to applications.

3.5 Drivers

Drivers use processor registers which are memory spaces within CPU. ATMega328p is an 8-bit microcontroller thus all these registers have an 8-bit size [33]. In this section, we present how drivers can be implemented and what drivers have been developed with Mongoose. Diagrams in this section are built by gathering data from different sources. The main source is the ATMega328p datasheet [35].

3.5.1 Serial communication

ATMega328p is an AVR microcontroller. Serial communication, also called UART (Universal Asynchronous Receiver/Transmitter) is achieved thanks to the AVR Libc, which defines C macros for baud rate calculation. More precisely, Baud rate is the rate for data transmission. Three macros are thus defined: *UBRRL_VALUE*, *UBRRH_VALUE* and *USE_2X*. Firstly, *UBRRL_VALUE* and *UBRRH_VALUE* are used to set up UART speed. Secondly, *USE_2X* is defined if double speed mode is required [32].

This kind of microcontroller has three registers for serial communication:

- **UCSR0A** : status data
- **UCSR0B** and **UCSR0C** : configuration setting

Figure 3.6 shows all possible values for these registers. The meaning of these values can be found in **Appendix A**

bits	7	6	5	4	3	2	1	0
UCSR0A	RXC0	TXC0	UDRE0	FE0	DOR0	UPE0	U2X0	MPCM0
UCSR0B	RXCIE0	TXCIE0	UDRIE0	RXEN0	TXEN0	UCSZ20	RXB80	TXB80
UCSR0C	UMSEL01	UMSEL00	UPM01	UPM00	USBS0	UCSZ01	UCSZ00	UCPOL0

Figure 3.6: Serial Communication Registers [32]

Receiving (RX) and Transmitting (TX) are enabled by *RXEN0* and *TXEN0* flags on **UCSR0B**. Data size on the communication channel is set by *UCSZ20* on **UCSR0B** and *UCSZ01*, *UCSZ00* on **UCSR0C**. Possible sizes are shown on figure 3.7 [32].

UCSZ20	UCSZ01	UCSZ00	Data size
0	0	0	5-bit
0	0	1	6-bit
0	1	0	7-bit
0	1	1	8-bit
1	1	1	9-bit

Figure 3.7: Data size [32]

In order to transmit a character from Arduino Uno to the desktop computer, UART has to be ready. When *UDRE0* flag is set in **UCSR0A**, data register is empty therefore a character can be transmitted by putting it on *UDR0* [32].

Data can be received with UART from *UDR0*. When *RXC0* flag is set, unread data is available. This byte (8-bit) is read on UDR0 [32].

Finally, STDIN and STDOUT are redirected to UART. By this way, writing and reading functions from AVR Libc can be used [32].

We have chosen 8-bit in order to transmit a character at a time. This driver implementation is based on the source code provided in reference [32].

3.5.2 Digital pins

ATMega328p has 3 General-Purpose Input/Output (GPIO) ports: B, C and D. As shown on figure 3.2, digital pins are on port B and D. Port B corresponds to digital pins from 0 to 7 and port D corresponds to digital pins from 8 to 13 [33].

Each port has three 8-bit registers (x for the port name) [33]:

- **DDRx** : Data Direction Register
- **PINx** : Pin Input Register
- **PORTx** : Pin Output Register

Each pin can be an input or an output. For instance, digital pin 2 needs to be an output if we want to turn on a led on digital pin 2. **DDRx** set "data direction of port pins" [34]. Writing 0 makes the pin as an input and writing 1 as an output.

For instance, if only digital pin 3 (port D) is an output, data direction register will be:

DDRD = 0b00001000

PINx (Pin Input Register) is used to read data on the corresponding pin. For reading, the related digital pin has to be defined as an input in **DDRx** register [34].

PORTx (Pin Output Register) is used to write data on the corresponding pin. For writing, the related digital pin has to be defined as an output in **DDRx** register [34].

In this example, a led on digital port 5 is toggled every second:

```
int main(){
    DDRD |= _BV(PORTD3);//Define digital pin 3 as output
    while(1){
        PORTD &= ~_BV(PORTD3);// Write 1 (on)
        _delay_ms(1000);
        PORTD |= _BV(PORTD3);// Write 0 (off)
        _delay_ms(1000);
    }
    return 0;
}
```

Pin mapping on ATMega328p, on figure 3.2, shows the pin mapping to use a specific digital pin.

Pin value change can be detected on ATMega328p. Indeed, this microcontroller manages hardware interrupts. There are two kinds of hardware interrupts: external interrupts (available on digital pins 2 and 3) and pin change interrupts (available on 23 pins) [37].

Pin change interrrupts can occur on RISING or FALLING signal edges. However, contrary to external interrupts, it is up to the interrupt handler to interpret it as a RISING or a FALLING signal edge [37].

We have implemented for this driver only the pin change interrupts.

When an interrupt occurs, the corresponding interrupt routine service (ISR) is called. Pin change interrupts share the same ISR between pins on the same port [36].

Pin change interrupts are set up by three registers:

- **PCICR** : Pin Change Interrupt Control Register
- **PCIFR** : Pin Change Interrupt Flag Register
- **PCMSKn** : Pin Change Mask Register n

PCICR register is responsible to enable pin change interrupts on a port. Global interrupts have to be also activated. If it is set for a port, any change on the corresponding pin will trigger an interrupt. Each pin is associated to a *PCINTx* as shown on the pin mapping figure 3.2. Pin change interrupts are activated individually thanks to **PCMSK0**, **PCMSK1**, **PCMSK2**. The content of these registers can be found in figure B.1 (**Appendix B**).

bits	7	6	5	4	3	2	1	0
PCICR	-	-	-	-	-	PCIE2	PCIE1	PCIE0
PCIFR	-	-	-	-	-	PCIF2	PCIF1	PCIF0

Figure 3.8: PCICR and PCIFR registers [35]

Figure 3.8 shows which bit to set in **PCICR** in order to enable pin change interrupts on port B, C, D. Bit 7 to 3 are not used. They will always be zeros [35].

A flag will be set in **PCIFR** register when an interrupt occurs. If interrupts are enabled on port A (respectively C and D), MCU will jump to the *PCI0* (respectively *PCI1* and *PCI2*) interrupt vector [35].

In the digital driver we have written, when the MCU jumps to an interrupt vector, it will first detect if it is a FALLING or a RISING signal edge. Then, if it is a FALLING signal edge (like pressing a button), the corresponding listener is sent to the event handler process. Finally, the event handler process is scheduled as soon as possible in order to execute the events listener. RISING signal edges are not implemented in this driver. Its implementation is based on the ATMega328p datasheet [35].

3.5.3 Analog pins

ATMega328p has 6 ADC (Analog-to-Digital Converter) channels. There are on port C as shown on the figure 3.2. These channels can be used to read analog sensor values such as temperature sensor, sound sensor or rotary angle sensor. They also have the same features

as digital pins on port B and D [38].

Analog-to-Digital Converter (ADC) converts an analog input voltage to a 10-bit digital value after a sequence of approximations. This voltage is measured between GND (minimum value) and AREF (maximum value) [35].

Analog pins have five 8-bit registers: [35]

- **ADMUX** : ADC Multiplexer Selection Register
- **ADCSRA** : ADC Control and Status Register A
- **ADCL** and **ADCH** : The ADC Data Register
- **ADCSRB** : ADC Control and Status Register B
- **DIDR0** : Digital Input Disable Register 0

We are going to explain only the first three registers which are used in the analog driver written with Mongoose. However, ADCSRB and DIDR0 details can be found in the Atmel datasheet [35].

bits	7	6	5	4	3	2	1	0
ADMUX	REFS1	REFS0	ADLAR	-	MUX3	MUX2	MUX1	MUX0

Figure 3.9: ADMUX – ADC Multiplexer Selection Register [35]

ATMega328p has only one ADC and 6 ADC channels. Between these, there is the analog muliplexer which connects a specific channel to the ADC. The input voltage has to be between GND (0V) and the voltage reference, otherwise it will damage the hardware [35]. Analog multiplexer is set up by **ADMUX** register [35].

Voltage reference can be changed with *REFS1* and *REFS0* values. All possible values and meanings are in the figure B.3 in **Appendix B**. We have chosen AVCC with external capacitor at AREF pin (*REFS1* is set to 0 and *REFS0* is set to 1) for the voltage reference in the driver. Thus, we can measure a signal between 0 and 5V [35].

Channels are connected to the ADC via the ADC multiplexer with *MUX3*, *MUX2*, *MUX1*, *MUX0* values. All possible values are on the figure B.2 in **Appendix B** [35].

ADLAR value changes the presentation of the ADC conversion. Figure B.4 and figure B.5 in **Appendix B** shows the two possible presentations [35].

ADCSRA register sets the ADC parameters. This register is presented on figure 3.10. Firstly, the ADC is enabled if *ADEN* bit is 1, or disabled otherwise. If the ADC is disabled during a conversion, it will be ended. Secondly, ADC conversion will be launched if *ADSC* bit is 1. This bit will return to 0 when the conversion is finished [35].

bits	7	6	5	4	3	2	1	0
ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Figure 3.10: ADCSRA – ADC Control and Status Register A [35]

In order to keep a 10-bit conversion precision, the working frequency of the ADC has to be between 50 kHz and 200 kHz. Conversion precision is lost if it is higher than 200 kHz. The ADC has a prescaler which will set an acceptable ADC clock frequency. This prescaler is set by *ADPS2*, *ADPS1* and *ADPS0*. All possible prescaler values are presented on figure B.6 on **Appendix B**.

$$ADC_{frequency} = \frac{CPU_{frequency}}{ADC_{Prescaler}} \tag{3.1}$$

ATMega328p frequency is 16 MHz. Therefore, ADC clock frequency is 125 kHz with a prescaler of 128. This frequency is in the range explained before. However, it is 250 kHz with a prescaler of 64. Conversion is thus lost. As a consequence, we implemented the analog driver with a 128 prescaler in order to have a 10-bit precision [35].

$$ADC = \frac{V_{IN} \times 1024}{V_{Ref}} \tag{3.2}$$

When ADC conversion is completed, *ADSC* flag is 0 as we have said earlier. In addition, *ADIF* is set when ADC data register (ADCH and ADCL) are updated. The result, between 0 and 1023, is then available [35].

3.5.4 Timer

Arduino Uno microcontroller has three timers: two 8-bit timers (Timer0, Timer2) and one 16-bit timer (Timer1). We have implemented the 16-bit timer in Mongoose scheduler in order to achieve a time-sharing scheduling.

We will explain Timer1 in detail. Timer0 and Timer2 follow the same approach but they are 8-bit timers, i.e. only 8-bit registers.

Timer 1 has the following registers which are 8-bit or 16-bit:

- **TCNT1** : Timer/Counter1 Register (16-bit)
- **OCR1A** : Output Compare Register A (16-bit)
- **OCR1B** : Output Compare Register B (16-bit)
- **ICR1** : Input Capture Register (16-bit)

- **TCCR1A** : Timer/Counter1 Control Register A
- **TCCR1B** : Timer/Counter1 Control Register A
- **TCCR1C** : Timer/Counter1 Control Register C
- **TIFR1** : Timer Interrupt Flag Register
- **TIMSK1** : Timer Interrupt Mask Register

ATMega328p is an 8-bit microcontroller, i.e. 8-bit data bus. Thus, 16-bit registers will need two writes or two reads. This timer has an 8-bit temporary register which is used to store temporary 8 bits from a 16-bit register. This temporary register is shared between all 16-bit registers of timer 1. In order to avoid corrupted data, these reading/writing operations are atomic: no interrupts can stop them [35].

bits	7	6	5	4	3	2	1	0
TCCR1B	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10

Figure 3.11: TCCR1B – Timer/Counter1 Control Register B [35]

The timer increments **TCNT1** (figure 3.12), counter register, at regular intervals thanks to a source clock. In addition, chip's internal clock is used as a source clock. This internal clock has the CPU frequency which is 16MHz for ATMega328p [39]. Therefore incrementing operation occurs every 62,5 ns. This period can be increased by a prescaler.

Timer Prescaler is set by Clock Select bits which are *CS12*,*CS11* and *CS10*. This prescaler decreases the timer frequency. All possible prescaler values are on figure C.1 in **Appendix C**. For instance, clock frequency will be 15,6 kHz with a prescale of 1024. Clock Select is also used for external clock setups [35].

bits	15	14	13	12	11	10	9	8
TCNT1H	TCNT[15:8]							
TCNT1L	TCNT[0:7]							
bits	7	6	5	4	3	2	1	0

Figure 3.12: TCNT1H and TCNT1L – Timer/Counter1 [35]

TCINT1 is a 16-bit register. The counter is incremented between 0 (BOTTOM) to 65535 (TOP). When TOP is reached, the counter goes back to BOTTOM. The TOP value can be changed thanks to OCR1A and OCR1B registers.

bits	7	6	5	4	3	2	1	0
TIMSK1	ICIE1	ICES1	-	-	-	OCIE1B	OCIE1A	TOIE1

Figure 3.13: TIMSK1 – Timer/Counter1 Interrupt Mask Register [35]

A timer interrupt can be set thanks to **TIMSK1** register. There are different kinds of timer interrupts: Input Capture Interrupt (*ICIE1*), Output Compare B Match Interrupt (*OCIE1B*), Output Compare A Match Interrupt (*OCIE1A*), Overflow Interrupt (*TOIE1*). Each interrupt will be triggered by different kinds of event.

bits	7	6	5	4	3	2	1	0
TIFR1	-	-	ICF1	-	-	OCF1B	OCF1A	TOV1

Figure 3.14: TIFR1 – Timer/Counter1 Interrupt Flag Register [35]

The input capture event can be triggered by a signal change at the input capture pin (ICP1) or by analog comparator pins. More precisely, it is triggered by an external event. Then the timer/counter value is saved in the **ICR1** register. When it happens, *ICF1* flag is set in **TIFR1** register and the corresponding ISR (Interrupt service routine) is executed [35].

Both Output Compare Register (**OCR1A** and **OCR1B**) values are compared to Timer/Counter value at every timer cycle. These register values can be between the BOTTOM value and the TOP value. When a match between OCR1x value and Timer/-Counter value happens, an output compare match event occurs. Then the corresponding *OCF1x* (A or B) flag is set in **TIFR1** register and the corresponding ISR is called [40].

The overflow event is triggered when Timer/Counter value reaches the TOP value. When it occurs, *TOV1* flag is set in **TIFR1** register and the corresponding ISR is executed [40].

We have set this timer for the scheduler with overflow interrupt. The value of needed overflow ($O_{counter}$) and the Timer/Counter start value (TCNT1) are computed with equations 3.3, 3.4 and 3.5 in order to reach a specific delay (Δ_t). These equations are based on explications given in reference [41].

$$O_{timer} = \frac{\Delta_t \times f_{CPU}}{C_{timer} \times k_{prescaler}} + 1 \quad \begin{cases} O_{timer} : \text{timer overflow} \\ \Delta_f : \text{delay in seconds} \\ f_{cpu} : \text{CPU freqency in Hz} \\ C_{timer} : \text{Timer/Counter range} \\ k_{prescaler} : \text{timer prescaler value} \end{cases} \tag{3.3}$$

$$O_{counter} = \lfloor O_{timer} \rfloor \tag{3.4}$$

$$TCNT1 = \lfloor (O_{timer} - O_{counter}) \times (C_{timer} - 1) \rfloor \tag{3.5}$$

The prescaler value has been chosen considering the delay needed by the scheduler in order to minimize the number of overflow interrupts and to keep a good precision. For this timer a prescaler of 8 has been chosen. Therefore only one timer overflow occurs if the needed delay is less than 32.768 ms with an error of 0.5 μs.

3.6 RAM in ATMega328p

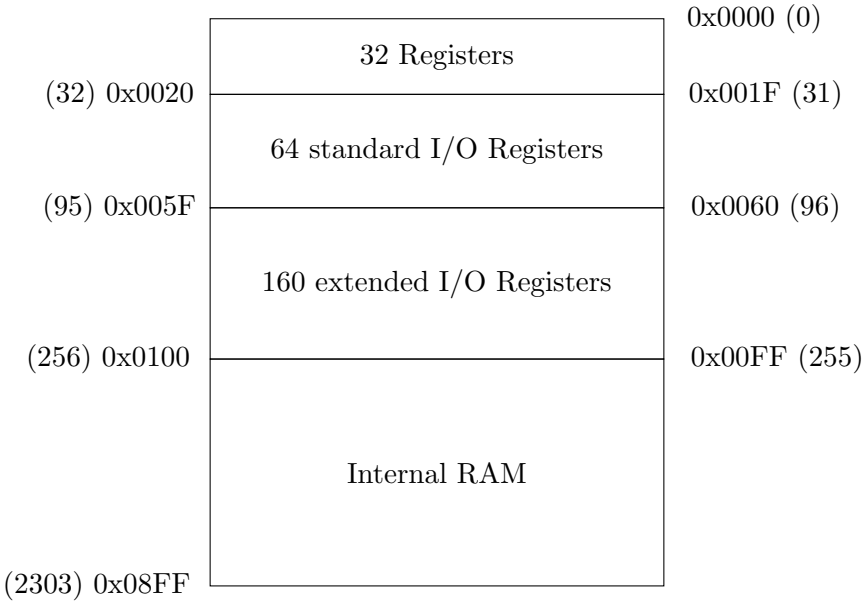


Figure 3.15: Data memory [35]

RAM (figure 3.15) in ATMega328p has four parts : 32 location addresses for the register file, 64 location addresses for standard I/O registers, 160 location addresses for extended I/O registers and 2048 location addresses for the internal RAM [35]. I/O registers have been presented in the driver section. This section explains the register file and the internal memory which are used for running applications and switching between them.

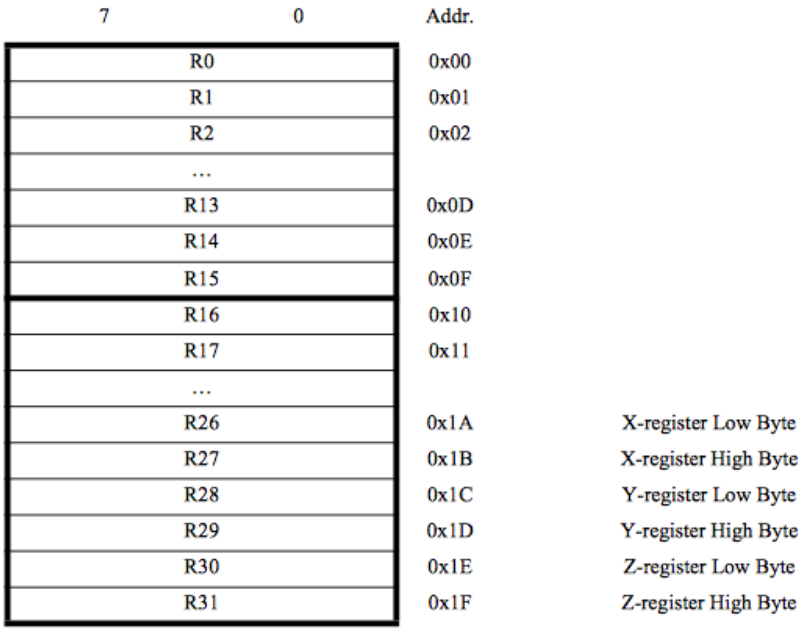


Figure 3.16: General Purpose Working Registers [35]

The register file, also called general purpose working registers, is composed of 32 registers of 8 bits (figure 3.16). They are used to store data temporarily. It can be "a byte of data to be proceeded, or an address pointing to the data to be fetched" [42]. These registers, R0 to R31, are located in the lowest position in the SRAM.

All arithmetic and logic operations work with these registers. Internal RAM is accessed by loading or saving operations. The last 3 registers: X-register, Y-register, Z-Register are 16-bit registers, contrary to the other registers. They are used for indirect addressing of the data source [35]. X-register and Y-register are used for access to 16-bit addresses in the internal RAM. In the same way, Z-register is used for access to 16-bit addresses in the flash memory which contains the program memory [43].

The last part of the RAM is called the internal RAM. It contains four sections as shown in figure 3.17.

The .data section is located at the bottom of the internal RAM. It contains static data and global variables which are initialized. On the contrary, the .bss section contains static data and global variables which are not initialized. Then the internal RAM contains the heap just after the .bss section. However, the stack is located at the top of the internal RAM. "there is no hardware-supported memory management which could help in separating the mentioned RAM regions from being overwritten by each other" [44]. At run-time, only the stack and the heap can grow. Consequently, the heap cannot override the two first sections.

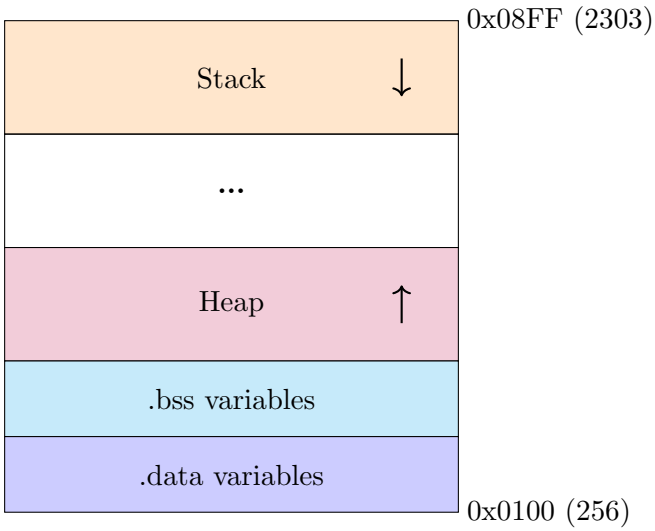


Figure 3.17: Internal RAM [44]

The stack contains local variables and return addresses. It is implemented to grow from the highest memory location to the lowest memory location [44]. Allocation and deallocation of memory blocks are done with a LIFO (Last In First Out) design. These operations are performed automatically during execution [45].

The heap contains variables which are dynamically created. Memory blocks have to be freed by the applications. The heap is also contained in the internal RAM and starts at the first memory address. However, we choose not to use the heap in Mongoose implementation because only 2KB are available in the internal RAM. In fact, this allocation/deallocation operations can fragment the internal RAM and memory space can be lost [44]. Also, applications written in Python cannot use the heap. Nonetheless, applications in C/C++ can work with the heap.

3.7 Multitasking

Multiple applications can run concurrently on the board thanks to Mongoose scheduler. In this section, we will explain how the scheduler is implemented and how switching between applications may occur.

3.7.1 Switch Context

The scheduler has a queue of processes to execute. Each process can be preempted in order to let other processes be executed. In order to run, each process has a context which is composed of the 32 General Purpose Working Registers, the Stack Pointer Register, the Program Counter and the SREG (status register) [46].

To switch between processes, the current process context is saved, then the next process context is loaded, and finally the execution is resumed. Each process has its own stack which is used for saving its context. The 32 General Purpose Working Registers have been explained in section 3.6. We explain now the three other kinds of registers which compose a process context.

Firstly, the program counter (PC) holds the address of the next instruction to be executed and works with the Z-register. When Mongoose switches between two applications, the previous program counter is stored in the current stack and the next program counter is loaded from the next stack. Thanks to the program counter, a process will be resumed from the last instruction executed. Program Counter is a 16-bit register. It is stored as two 8-bit registers called PCH (High) and PCL (Low).

bits	7	6	5	4	3	2	1	0
SREG	I	T	H	S	V	N	Z	C

Figure 3.18: SREG – AVR Status Register [35]

Secondly, the Status Register (SREG) contains the result of the last arithmetic operations. This register will be also saved in the current stack when Mongoose switches between two applications. Flag I on bit 7 is used to enable/disable global interrupts. By disabling this flag, atomic operations can occur without being preempted by timer or hardware interrupts. The other flags (T,H,S,V,N,Z and C) are used for arithmetic, logical and bit-functions [35].

Finally, the Stack Pointer Register points to the top of the current stack after the context space. It is a 16-bit register. It is stored as two 8-bit registers called SPH (High) and SPL (Low).

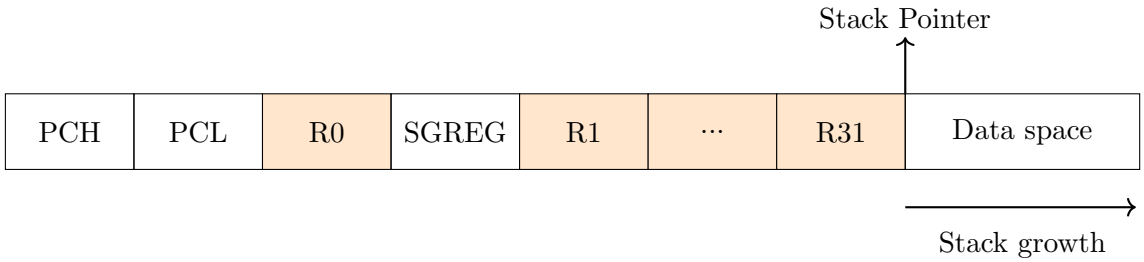


Figure 3.19: Stack shape

Figure 3.19 shows the resulting stack shape. The following steps represent how the context is saved [47]:

1. Save Register R0

2. Move SREG register in R0
3. Save SREG
4. Disable all interrupts
5. Save R1 and clear R1
6. Save R2 to R31
7. Load in R26 and R27 (X-Register) with the address where the Stack Pointer will be saved
8. Stack Pointer is saved (high byte then low byte)

The next following steps represent how the context is restored [48]:

1. Load in R26 and R27 (X-Register) the address where the Stack Pointer can be retrieved
2. Load Stack Pointer (low byte then high byte)
3. Restore R31 to R1
4. Restore SREG thanks to R0
5. Restore R0

Saving and restoring the context is done with assembly language. Stack shape is built in C when a process is initialized. This part of Mongoose source code is based on AvrRTOS kernel written by Tapio Hirvikorpi and Johannes Aalto for ATmega88. It can be found in reference [50] and [51].

3.7.2 Process and stack

Switching between processes has been explained previously. We explain here how processes are initialized and how the internal RAM is divided. The internal RAM corresponds to the RAM without the 32 general purpose working registers and the I/O registers.

The internal RAM has a capacity of 2 kB. The .data section, the .bss section and the heap is common to all processes.

At the beginning, each process is initialized with a stack size and the process' main function. The stack shape, as explained above, is built considering the memory needed. Each process has its own stack. In order to fit in the internal RAM, Mongoose evaluates how much free RAM is available. This evaluation is done by creating a stack variable (local variable for instance) and a system variable. "At any point in time, there is a highest point in RAM occupied by the heap. This value can be found in a system variable

called _brkval" [49]. In addition, the first variable will be addressed at the end of the free RAM. Indeed, the stack starts at the top of the RAM as detailed above in section 3.6.

After this evaluation, Mongoose sums all stack sizes each time a process is initialized. If the sum of these application stack sizes is above the free RAM size, an OS error will be printed on the serial port and Mongoose does not start.

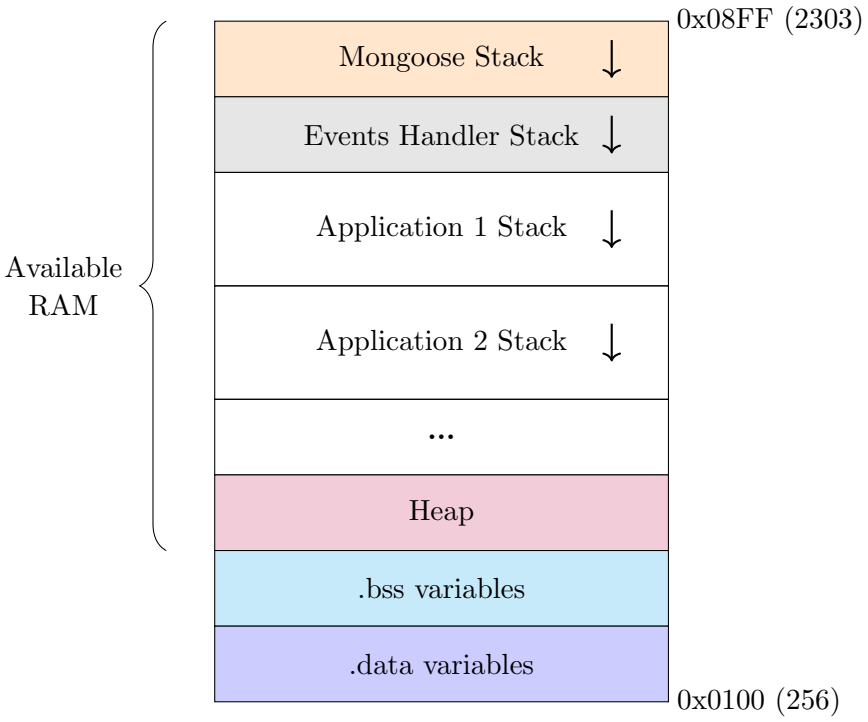


Figure 3.20: Internal RAM and stacks

The internal RAM will contain all process stacks. In addition, the heap, which is common to the whole system, is not used if no application in C/C++ uses it. Therefore the heap can take a negligible memory space.

Figure 3.20 shows how stacks are located in the internal RAM. From the top of the internal RAM, the first stack is allocated for Mongoose kernel, then for the event handler process and finally the applications. The heap is located at the beginning. If the heap is not used by any application written in C/C++, the internal memory will contain only Mongoose stack and process stacks.

There are three kinds of stack sizes: SMALL, NORMAL and LARGE. These sizes are precised when an application is added to the Mongoose scheduler. SMALL should be used by application which manipulated hardware like blinking a LED or take a sample of temperature every second. However, LARGE should be used for heavy background applications which compute, for instance, a Fourier transform.

Mongoose kernel and the event handler have a SMALL and a NORMAL stack sizes, respectively.

Collision, on one hand between processes' stacks and, on the other hand between the last stack and the heap can happen. No mechanism has been implemented with Mongoose to prevent this situation. Thus, stack sizes have to be chosen carefully considering the whole system, especially if the heap is used.

The positions of the application stacks are set in the same order as they are added to Mongoose scheduler. Hence, the last application stack can grow until there is no more free RAM, i.e the top of the heap. If the heap is not used by any application, the last stack size will be greater or equal to the size asked during its initialization. It cannot be less because of the free RAM evaluation. Nonetheless, if the heap is needed by an application, the last stack size could be smaller than the asked size. It can lead to a collision with the heap. This situation cannot be detected during the free RAM evaluation because it depends on applications' usage of the heap (Mongoose does not use the heap).

3.7.3 Scheduler

ATMega328p is a single core microcontroller. Thus only one instruction is executed at a time. A scheduler is needed in order to achieve concurrency between processes.

The scheduler is a policy to set which process will be executed at a certain point of time. This execution order can be defined by priorities or time constraints.

Scheduling algorithms can be static or dynamic. In the static approach, the order is computed in advance and has a small overhead at run-time. In the dynamic approach, the order is computed at run-time and has a bigger overhead. Despite this overhead, the dynamic approach is more flexible for random events [52].

A scheduler can be preemptive or non-preemptive. In the preemptive approach, a process can be suspended and another process is executed instead. Preemption is done with context switching which has been explained earlier. In the non-preemptive approach, a process cannot be suspended during execution. It can be implemented as a mechanism for atomic execution. Finally, in the hybrid approach, which combines this two scheduling approaches, process can be suspended and also have non-preemptable section of code [52].

Different classes of scheduling algorithms for real-time operating systems are possible [53]:

- **Clock-driven**: At a specific time, the scheduler chooses the next job to execute. The schedule is usually done before run-time. It is simple but not flexible.
- **Weighted round robin**: It is a preemptive time-shared scheduling. All processes are in a FIFO (First In First Out) queue. The scheduler executes the top of the queue for at most a period, also called quantum and time slice, then adds it at the end of the queue. Each process will have an executing time at each round.
- **Priority-driven**: Each process has a priority. Scheduling decisions are based on these priorities. When releases or process completions occur, the scheduler will

choose the next process to execute.

The main goal of Mongoose is to concurrently execute applications which are represented by processes. Each application in Mongoose is executing infinitely (they never end). If each application has different priorities, the highest priority application will be executed all the time unless this application explicitly releases the CPU. In this approach, these release steps have to be included in the application development. We want to ensure a minimum executing time to each application even if explicit releases are not used. Therefore pure priority-driven algorithms do not seem to be adequate.

Moreover, we want this scheduler to make decisions at run-time. In fact, random hardware events like pressing a button cannot be anticipated before run-time. Consequently clock-driven algorithms do not seem to be adequate.

Mongoose is a real-time operating system. Therefore this operating system has to respect time constraints, be responsive to external events and have a predictive behaviour. As a consequence, we choose to base Mongoose scheduler on **a round robin algorithm with fixed time slice which can be reduced by time constraints and hardware events**. Each application will have the same priority. Thus this policy is fair between applications. This scheduler is dynamic and hybrid (preemptive approach with non-preemptive mechanisms).

When Mongoose is launched, every application is added to the scheduler FIFO queue after its stack shape is built (explained in section 3.7.2). A clock is then associated to the scheduler with a constant period called kernel period. This clock is implemented with the Timer 1 detailed in section 3.5.3. Every application is ready to run before the scheduler begins.

When Mongoose runs, the scheduler begins. At each kernel period, the clock triggers an interrupt which suspends the current process. The current process is saved. Then the scheduler chooses the next process to execute following the FIFO (First In First Out) policy. The previous process is now at the end of the queue. The next process' context is loaded. This process continues its execution until a new timer interrupt occurs. The kernel period is the time slice allocated to each process. It is also called quantum.

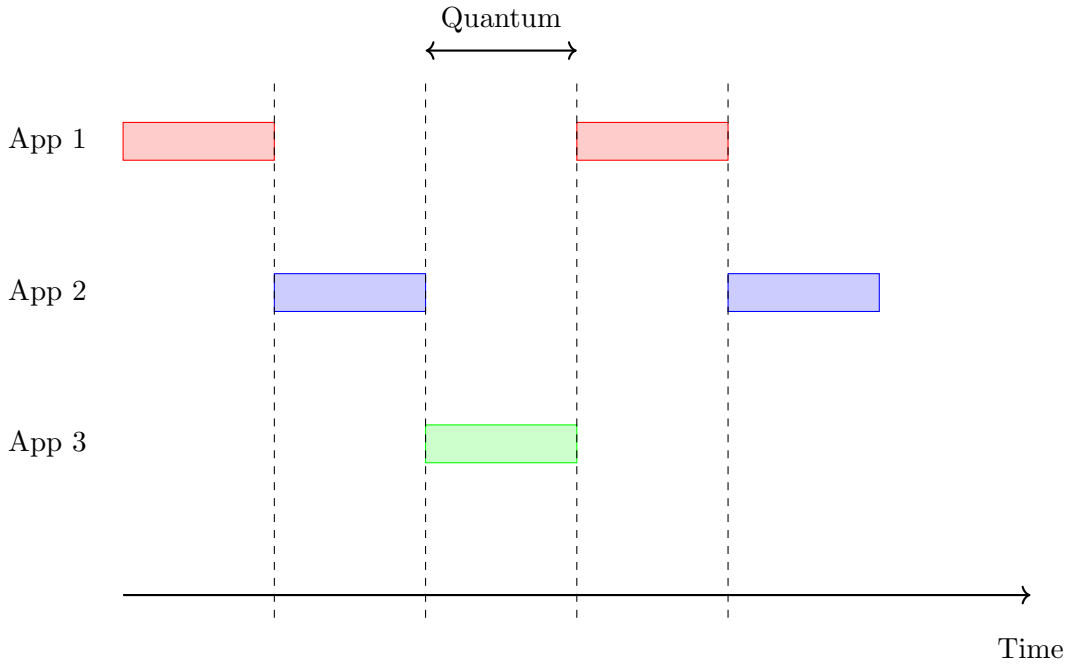


Figure 3.21: 3 simple applications

Figure 3.21 represents this algorithm with 3 applications, no hardware events and no time constraints.

Time constraints are expressed as sleeping time in Mongoose. More precisely, when an application needs to wait a certain amount of time, this application asks Mongoose to block its execution until this amount of time passes. An array with a position for each process is used to store this amount of time. When it occurs, the scheduler is called in order to block this process and switch to the next process.

At each clock interrupt, the sleeping array is updated and each sleeping time is decreased of the kernel period (quantum).

Considering 3 applications with time constraints, figure 3.22 represents how processes are scheduled. Scheduler computational time is considered equal to zero. This example has the following characteristics:

- **App 1** asks periodically 100 ms of sleeping time.
- **App 2** asks periodically 50 ms of sleeping time.
- **App 3** asks periodically 150 ms of sleeping time
- **Kernel period** (quantum) is 50 ms

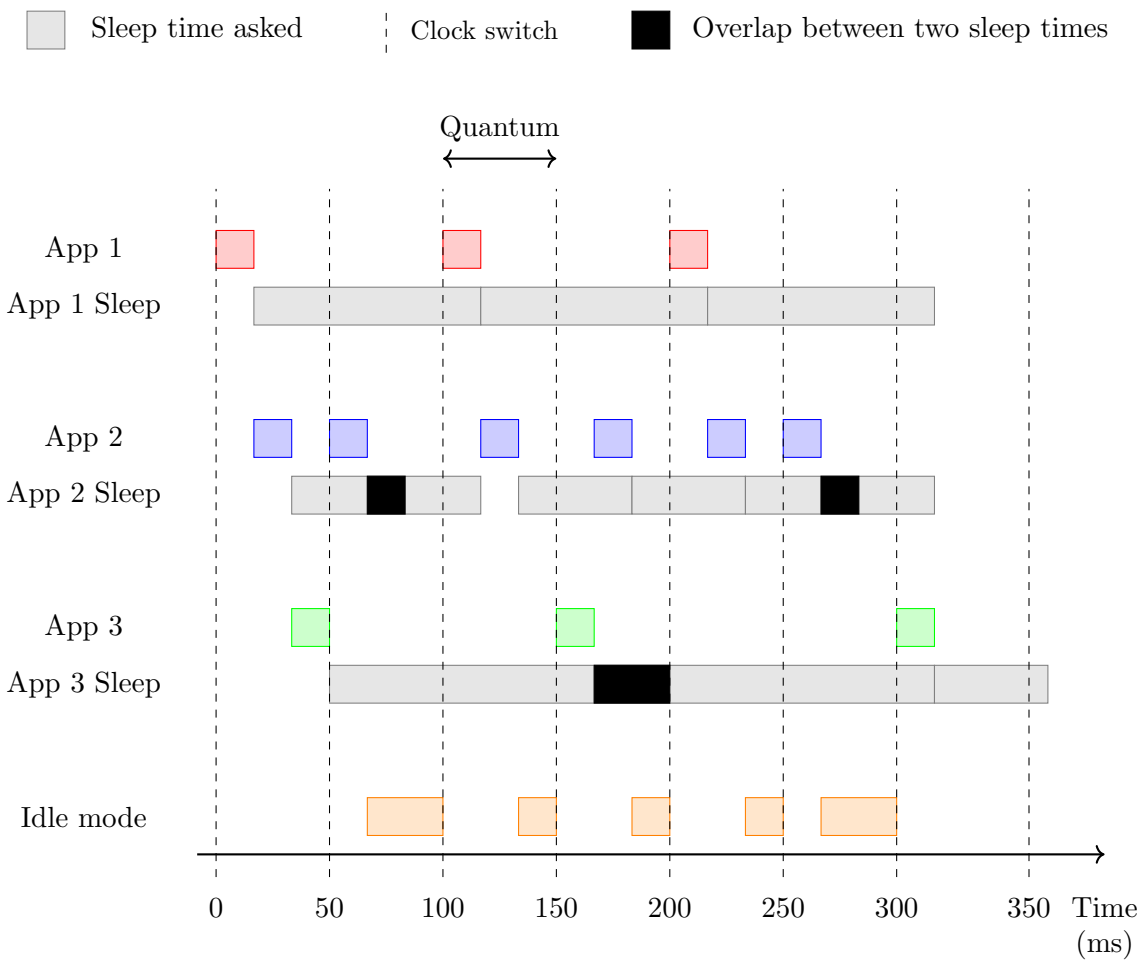


Figure 3.22: 3 applications with time constraints

In this example, we can observe that almost every time constraint is respected with an error between -50 ms and 0 ms. This error depends on the kernel period. In this case, no application needs a full quantum between two time constraints. That is why no application is scheduled after the time constraint passed. Additionally, the scheduler switches between processes. More precisely, every time an application asks a time constraints, the scheduler is called.

For the first approach, the error can be estimated with the kernel period and the number of processes.

There are two worst cases:

- Earliest time: an application begins its sleep mode just before a clock interrupt.
- Latest time: an application finishes its sleep mode just after a clock interrupt. Then when this application wakes up, it is at the end of the Scheduler queue.

This two cases frame approximately the error for a time constraint. The following inequality 3.6 represents this frame:

$$-T_{kernel} \leq \varepsilon_{RR} \leq T_{kernel} + (N_{process} - 1) \times T_{kernel} \quad \begin{cases} T_{kernel} : \text{kernel period} \\ N_{process} : \text{number of processes} \\ \varepsilon_{RR} : \text{error (round robin)} \end{cases} \tag{3.6}$$

The smaller T_{kernel}, the smaller the error ε_{RR} is. However, we considered previously that the scheduler computational time is equal to zero. Thanks to an oscilloscope, we measured this computational time.

Cases	Computational time
Inside clock interrupt	30-40 μs
Outside Clock interrupt	20-25 μs

Figure 3.23: Scheduler computational time

The scheduler switches between application when a clock interrupt occurs, when a sleep mode is asked and when an event is received. Figure 3.23 shows the scheduler computational time inside and outside a clock interrupt. There is no update to the array of sleep times outside a clock interrupt. That is why, these results are different.

This computational time is a loss. As a matter of fact, this time is not used by applications. The number of times that this loss occurs for sleep mode and events reception cannot be reduced because it is independent of Mongoose scheduler. However, the kernel period has a direct impact on this number. Time losses happen more often when the kernel period is small.

For instance, the scheduler is called approximately 1850 times in one second with a kernel period equal to 500 μs. More precisely, this overhead occurs between two kernel periods that is why it is called less than 2000 times. Considering this case, around 75 ms are lost by the scheduler every second. So 7.5% of computational time is not used by the applications because of the clock interrupts. Thus applications which do not need time constraints will particularly be slower. Additionally, the scheduler is called at each clock interrupt. This computational time introduces an error in the clock. More precisely, the kernel was between 530 and 540 μs instead of 500 μs in this example.

This scheduler overhead can be estimated by the equation 3.7:

$$O_{scheduler} = \frac{\delta_{scheduler}}{T_{kernel} + \delta_{scheduler}} \quad \begin{cases} T_{kernel} : \text{kernel period} \\ O_{scheduler} : \text{scheduler overhead} \\ \delta_{scheduler} : \text{scheduler computational time} \end{cases} \tag{3.7}$$

As a consequence, the kernel period has to be determined in order to reduce the time constraint error and to reduce losses due to the scheduler overhead.

The absolute error is approximately the sum of the error due to the scheduler overhead and the error due to the round robin strategy. This absolute error δ_t is estimated by:

$$\delta_T = \frac{T_{kernel} + \delta_{scheduler}}{T_{kernel}} \times T + \varepsilon_{RR} \quad \begin{cases} T : \text{time constraint (delay)} \\ \varepsilon_{RR} : \text{error (robin round)} \\ T_{kernel} : \text{kernel period} \\ \delta_T : \text{absolute error} \end{cases} \tag{3.8}$$

Therefore:

$$\delta_T = \frac{T}{1 - O_{scheduler}} + \varepsilon_{RR} \tag{3.9}$$

Thus the relative error for a delay T is:

$$\delta_{T(r)} = \frac{\delta_T}{T} = \frac{1}{1 - O_{scheduler}} + \frac{\varepsilon_{RR}}{T} \quad \begin{cases} T : \text{time constraint (delay)} \\ \delta_T : \text{absolute error} \\ \delta_{T(r)} : \text{relative error} \end{cases} \tag{3.10}$$

In addition, the responsive time limit is 100 ms. Under this time, a human user will feel that the system is instantly responsive [54]. The scheduler considers at most 4 processes: 3 applications and the event handler process. Thus the kernel period should be less than 20 ms according to equation 3.6.

Time constraints, in Mongoose, are stored with a 16-bit value in milliseconds. Therefore, time constraints are based on delays between 0 and 65 535. So, the relative error should be considered only for this range.

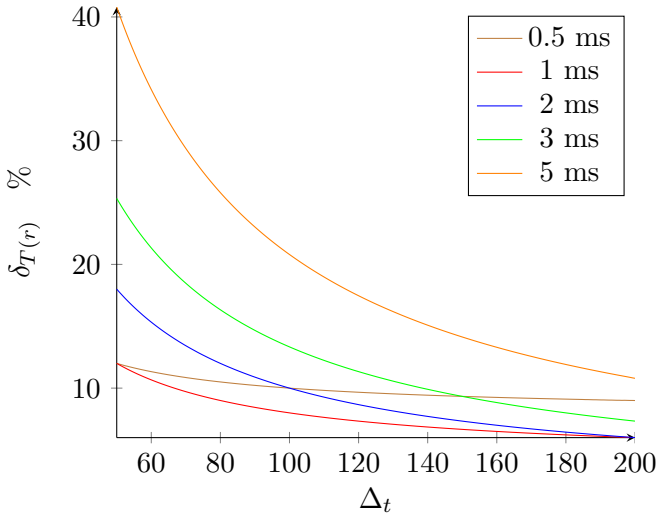


Figure 3.24: Relative error $\delta_{T(r)}$ between 75 and 200 ms

Figure 3.24 represents the relative error $\delta_{T(r)}$ for delays between 50 and 200 ms. It is expressed in the figure as a percentage. The responsive time limit is 100 ms as said before. Therefore the ideal kernel clock should have a small relative overall error around this value. We can observe that kernel clocks (quantum) under 2 ms have a relative error less than 15% for a delay between 75 and 200 ms.

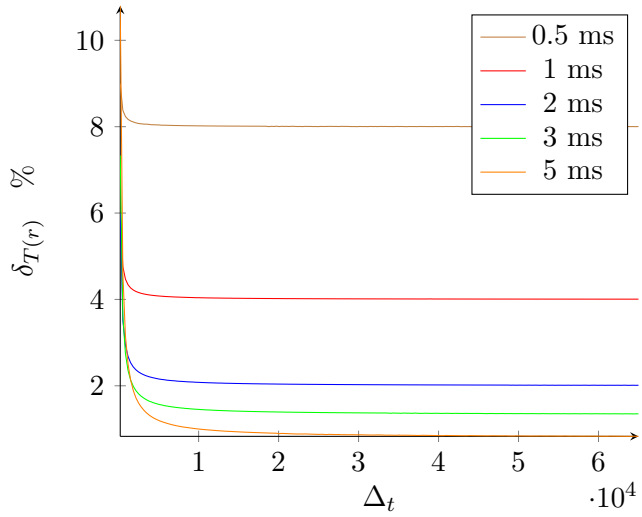


Figure 3.25: Relative error $\delta_{T(r)}$ between 200 and 65535 ms

Figure 3.25 represents the relative overall error for delays between 200 and 65 535 ms (maximum delay). We can see that kernel clocks (quantum) above 2 ms have a relative error less than 2.5% for a delay above 1000 ms.

The kernel clock is constant at run-time. Consequently, it has to be chosen in order to minimize the relative error $\delta_{T(r)}$, for small and long delays. A kernel clock of 2 ms offers good performance for small and long delays.

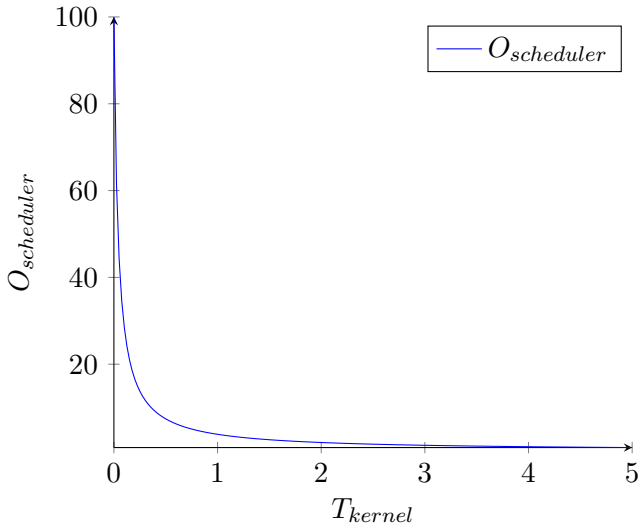


Figure 3.26: Scheduler overhead

We can see in figure 3.26 that a kernel clock above 2 ms introduces an overhead of 2%. If the kernel clock is greater than 2 ms, Mongoose will have bad performance for small delays. That is why we have chosen this value for the Mongoose kernel clock. Some kernel clock performance will be compared to support this choice in chapter 5.

As we said before, the scheduler introduces computational time losses and an additional error. The losses cannot be reduced to zero. However, the time error could be reduced. In fact, when an application needs a delay, a possibility could be to remove from this value the theoretical error especially the error due to the scheduler overhead.

By this approach, if an application needs a delay T, the scheduler could consider this sleeping optimized time T_{opt}:

$$T_{opt} = T - (1 - O_{scheduler}) \times T \tag{3.11}$$

So:

$$T_{opt} = O_{scheduler} \times T \tag{3.12}$$

This approach will be tested in chapter 5. We will observe that the error is small and not related to the kernel period. Nonetheless, for short delays, the error stays related to the kernel period. Only the time constraints example will compare the impact of non-optimized and optimized delays.

3.8 Events

An application can listen on a specific digital pin. For instance, an application can react to a button state. It is called an event. Hardware events are random, therefore these events can block an application which is waiting for a specific event to occur. In order to avoid this case, we choose to use a process which is responsible to handle hardware events for all applications. It is called the event handler process.

As explained in section 3.5, pin value change can be detected. It will trigger a hardware interrupt. The digital driver detects if it is a FALLING or a RISING signal edge. If it is a FALLING signal edge, this driver will send an event to the event handler process.

When an application needs to listen on a specific digital pin, it assigns to this pin a block of code called listener. Only one listener can be assigned to a pin. Thus only one application can listen on a pin. An array of function addresses stores this listener in order to execute it when the corresponding event occurs.

We consider that these listeners need only a short computational time. They may also need to set a time constraint which is possible in the event handler process. In this context, we choose to let listeners run to completion. We adopt a First In First Out algorithm. Indeed, when a hardware event occurs, the corresponding listener is added to a FIFO queue.

Any event cannot preempt another event. Thus a listener can block this process and the next events. A long event can also decrease the responsive time for future hardware events. As we said before, process can be preempted, this non-preemptive execution of events makes the scheduler hybrid (explained in section 3.7.3).

Consider this following example: a human user press a button 20 times in one second. The corresponding listener takes 100ms for instance. Hence, this user will have to wait 2 seconds before any other event is executed. Thus, we choose to set a limited capacity for this FIFO queue. This approach reduces the impact of the number of events on the responsive time. However, events will be lost. When the FIFO queue is full, the following event will be rejected. In addition, we have chosen that if a listener is still executed and its corresponding event occurs again, it will be also ignored.

As shown in Figure 3.21 and Figure 3.22, the event handler process is not executed if no event occurs. More precisely, when the event handler process is called by the scheduler and the FIFO queue is empty, this process will hand over to the scheduler letting execute another process. This process hand over as well when all listeners have been executed and the FIFO queue becomes empty. In order to increase the responsive time, the scheduler stops the current application and wakes the event handler process up when an event occurs.

3.9 Inter process communication

3.9.1 Overview

Applications are processes with separated stacks. Two applications may need to share results. It is possible thanks to inter process communication.

Processes can communicate through channels. This mechanism goes through the kernel. Each channel can store a 16-bit integer. In addition, each process can read or write on any channel. These channels are thus a memory section which is shared among all processes. In order to prevent corrupted data, a mutual exclusion has been implemented with this mechanism.

Mutual exclusion access to shared memory, also called critical section, prevents issues like race condition [55]. When multiple processes read and write on a shared memory and "the final result depends on the order of execution" [56], a race condition happens.

This mutual exclusion access ensures [55]:

- Only one process can access the shared memory
- Any process will not wait infinitely to access the critical section (no starvation)
- Two (or more) processes cannot be blocked because they try to access the critical section (no deadlock)

When a process tries to access the channel, it locks this channel if it is unlock. Otherwise the process waits. Lock and unlock operations are atomic. These atomic operations ensure that only one process can access a channel at a time.

Each channel has a binary flag (locked or free). To lock/unlock a channel, global interrupts are deactivated making this operation atomic. Then, if the channel is locked, the process is stopped and the scheduler switches to another process. Otherwise the channel is locked and the process can go in the critical section reading or writing on this channel. Finally, the channel is unlocked. The scheduler switches then to another process.

These final steps ensure that a process unlocking the channel cannot access it again if there is another process waiting. More precisely, each application has the same priority. This property ensures fairness for accessing a shared memory.

In order to force the order between reading and writing, another binary flag (unread or read) is used. Every time a process tries to read a channel, it has to wait that an unread data is available. In the same way, every time a process tries to write on a channel, it has to wait until the previous data is read. This flag is controlled by atomic operations.

It is a one-to-one communication but a message can be broadcast using multiple channels. There are at most 3 applications. Therefore **6 channels are needed** to have one

channel between every process.

This approach introduces a dependency between two applications. Each channel used needs one reader and one writer. This mechanism lets applications communicate in the right order: read a channel after a new value is written.

3.9.2 Inter process communication through examples

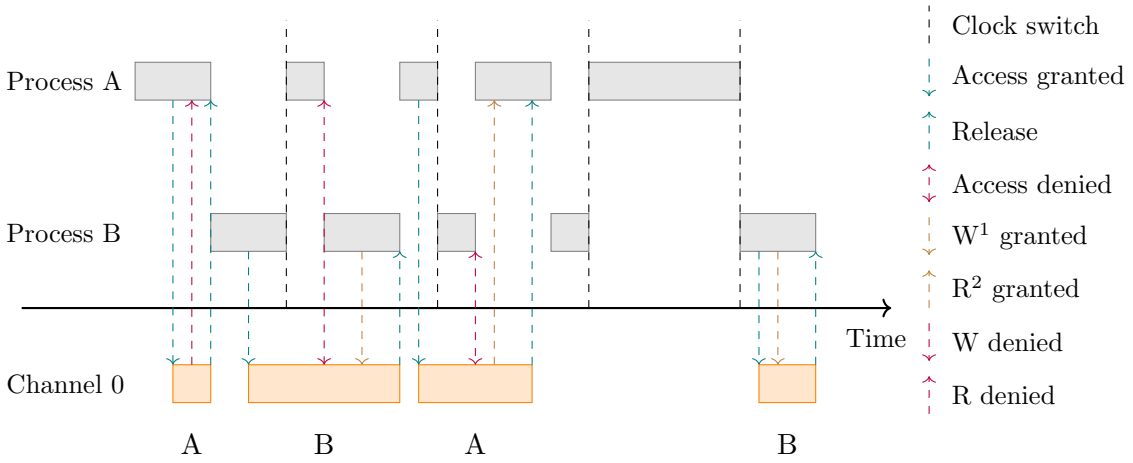


Figure 3.27: Read denied

Figure 3.27 represents the synchronisation between a reader **A** and a writer **B**. In this situation, **A** tries to read before a new value is available. Therefore, the reader is rejected and tries again until the new value is ready. If this process is blocked until the scheduler is called by a clock interrupt, a quantum (time slice) is lost. This is why, every time a process is rejected, the scheduler is called. It will allow B the possibility to write a new value on this channel.

This example also shows the lock/unlock mechanism avoid two processes to access the critical section at the same time. In figure 3.27, **A** tries to access the critical section when **B** is already inside. This mechanism is atomic avoiding two locks simultaneously. It prevents race conditions.

[1] W : write
[2] R : read

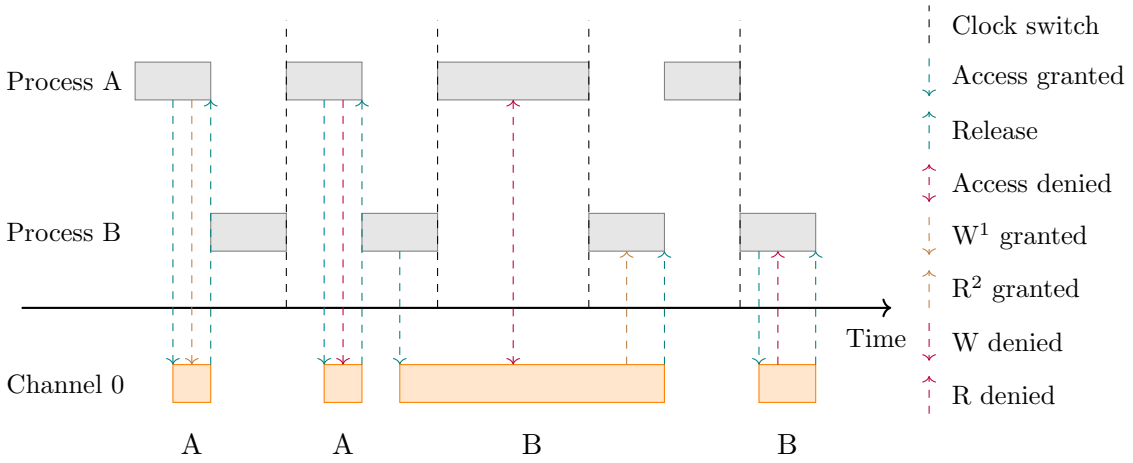


Figure 3.28: Write and read denied

Figure 3.28 represents also the synchronisation between a reader **B** and a writer **A**. In this situation, the writer tries to write two successive times. Therefore, the writer is accepted the first time and rejected the second time. It tries again until the last value is read by the reader. After a successful write, the scheduler is called letting a possibility for the reader. By this way, the number of unsuccessful writes is reduced if a process tries to write multiple times in only one quantum. In addition, the reader **B** tries to read two successive times before **A** writes a new value. The second reading is therefore rejected and **A** tries again until a new value is available.

3.9.3 Limitations

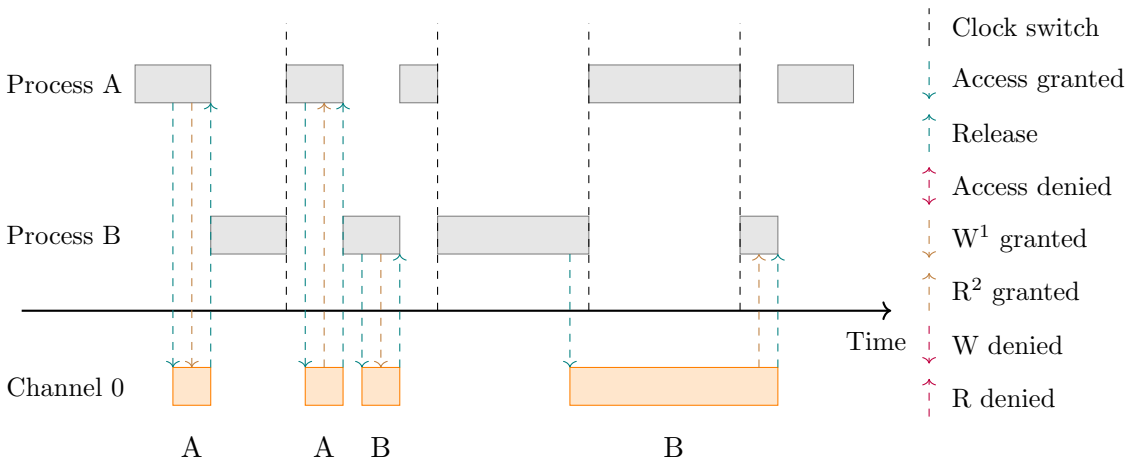


Figure 3.29: Communication in two directions on the same channel **(incorrect)**

[1] W : write
[2] R : read

Figure 3.29 represents a communication in two directions between **A** and **B**. They are reader and writer on the same channel. Firstly, **A** writes and reads successively. After **B** writes and reads successively also. At the end, **A** did not get **B**'s message and **B** did not get **A**'s message. As a consequence, no communication was established between this two processes and computational time is lost. Communication in two directions on the same channel is not reliable. In fact no mechanism ensures that a process does not read a value sent by itself.

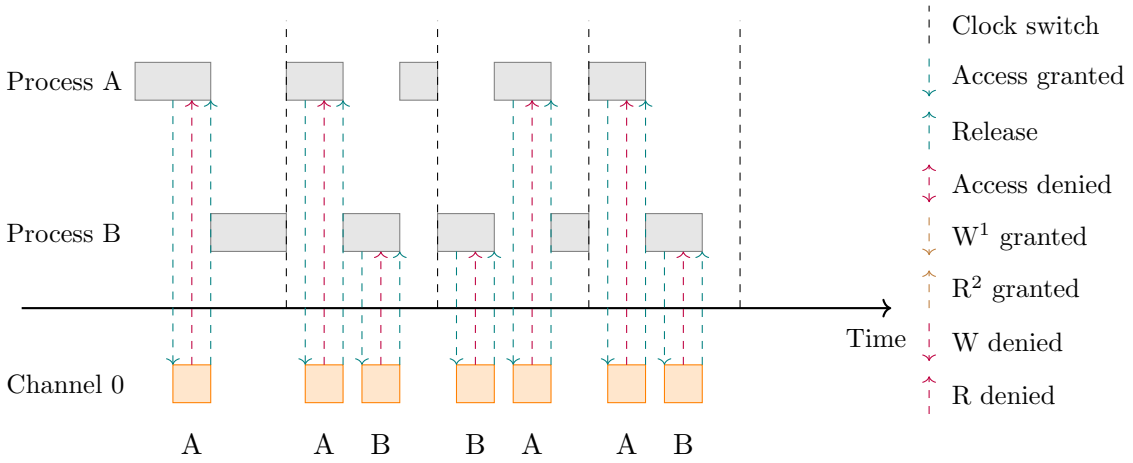


Figure 3.30: Communication in two directions on the same channel (**deadlock**)

Figure 3.30 represents also a communication in two directions between **A** and **B**. They are reader and writer on the same channel. Firstly, **A** tries to read two times but these reading operations are rejected because no new data is available. **A** is waiting that **B** writes a new value. Then **B** tries to read the value on this channel instead of writing a new value. Thus, **B** is waiting for **A** too. Consequently, **A** and **B** are blocked infinitely. This situation is called a **deadlock**.

This issue does not occur because of the access mechanism (mutual exclusion access). It happens because of the synchronisation mechanism.

The two previous examples show that this mechanism is not suitable for communication in two directions on the same channel. That is why, there are 6 channels in order to have two channels between each application.

In addition, this mechanism has issues when there are not exactly one reader and one writer:

1. Two processes cannot successively read the same value on a channel

2. Two processes cannot successively write on a channel

[1] W : write
[2] R : read

3. A writer is blocked if no process is a reader on this channel

4. A reader is blocked if no process is a writer on this channel

If three are two readers and one writer on a channel (*case 1*), some issues are possible. Firstly, only one reader can have the new written value. In the best case, the two readers succeed to read some values written but not all of them. In the worst case, one reader read first every new value from the writer. Therefore, the second reader will be blocked infinitely. In fact, it will access to the critical section. However it will be blocked because every new written value is read by the first process. The second reader will access the critical section infinitely and its reading operating will be declined. This process will be therefore blocked by **starvation**.

If there are two writers and one reader on a channel (*case 2*), a writer can be blocked. Indeed, a writer, in this approach, cannot override a value if it has not been yet read by a reader. In the best case, the reader reads the channel between every writing operation. Thus, no data is lost. In the worst case, one writer is trying infinitely to write a new value just after the other writer and before it is read by the reader. So, this writer will infinitely access to the channel. However, its writing operations are rejected. Therefore, this writer is blocked by **starvation**.

If there is only one writer and no reader on a channel (*case 3*), the writer will be blocked permanently. As a matter of fact, its first writing operation is executed. Nonetheless, its following writing operation is rejected because overriding an unread value is not allowed. Consequently, it will be blocked permanently by **starvation**.

If there is only one reader and no writer on a channel (*case 4*), the reader is blocked permanently. More precisely, its first reading operation is declined. This process will access to the critical section infinitely. However, all its reading operations are rejected. Thus, this reader will be blocked by **starvation**.

In few words, having exactly one reader and one writer on a channel prevents **starvation** and loss of data as well.

Moreover, communication between two applications (**A** and **B**) thanks to two channels can lead to a blocking situation. We suppose in the following situations that *channel 0* is from **A** (writer) to **B** (reader) and *channel 1* is from **B** (writer) to **A** (reader):

1. **A** reads *channel 1* before writing on *channel 0* and **B** reads *channel 0* before writing on *channel 1*

2. **A** or **B** reads two successive times *channel 0* or *channel 1*

3. **A** and **B** write two successive times *channel 0* or *channel 1*

If **A** and **B** try to read their respective channel before writing anything, each process will have to wait until one of the two writes a new data. No writing operation is possible because both processes are waiting each other. As a consequence, they infinitely access

the critical section but all their reading operations are rejected. They are blocked in a **deadlock** situation.

If **A** or **B** reads two successive times a channel without letting the other process writing on the process, the reader on this channel will be blocked. It lets the other channel without a writer. Therefore, the second process will wait permanently in the future. They are blocked in a **deadlock** situation.

If **A** and **B** write two successive times on a channel without letting a process reading this channel, they will both wait infinitely. Indeed, they cannot override an unread value. They are blocked in a **deadlock** situation.

3.9.4 Summary

Mongoose inter process communication mechanism lets applications accessing the critical section without **starvation** and **deadlock**. It is due to equal priorities in the scheduler policy and scheduler call after the shared memory is unlocked. The synchronisation mechanism (read/unread) forces applications to read only unread data and not to override unread values.

Nonetheless, this synchronisation mechanism can create starvation and deadlock situation. **Starvation** can happen when a channel does not have exactly one reader and one writer during use. In addition, **deadlock** situation can occur when two processes communicate on the same channel in two directions. This use is not reliable because a process can read a value it wrote. Finally, **deadlock** situation can also happen for communication in two directions with two different channels. More precisely, if the balance between read-/write operations and their order prevent a proper communication, both processes can be blocked permanently.

An example with three applications communicating together is presented in chapter 5.

3.10 Source code

Mongoose source code is divided in 5: kernel, process, message, event and timer.

Kernel files contain Mongoose kernel source code. They are composed of Mongoose scheduler, context switch functions, process initialisation, functions for executing an atomic block of code and Mongoose's error handler.

Process files manage process capabilities and environment. Mongoose kernel mainly calls these files when it initializes processes and Mongoose scheduler is called.

Message files are in charge of the inter process communication. More precisely, it is responsible of channels' features.

Event files deal with the event handler process. They manage event reception and add them to the event handler process if they are accepted for execution.

Timer files set up Mongoose kernel clock. It is used by Mongoose scheduler in order to achieve a time-share robin round algorithm.

On one hand, functions which start with "MOS" can be called by applications. On the other hand, functions which start with "KOS" are only used by the operating system.

Hardware drivers are composed of: digital driver, analog driver, serial port, led driver. All drivers, except the LED driver, have been explained previously in this chapter. The LED driver is a specific driver to manage the pin 13 on the board which is already connected to a LED. All drivers' functions start with "HOS".

The source code can be found at https://github.com/hdlj/Mongoose .

Chapter 4

Development environment

4.1 Motivation

One of the objectives for this project was to let Mongoose being developer friendly. In order to achieve that, we have chosen to develop a programming environment. Different programming languages are possible:

- C/C++
- Python
- Java
- JavaScript
- Create a new language

Intel allows developers to use C and C++ with Arduino IDE. Other languages are also available like Python, Java or JavaScript. However, additional libraries, compilers or interpreters are needed on the board.

Arduino Uno has only 2kB RAM. On one hand, C seems to be the most appropriate considering memory and speed. On the other hand, Python is known for its learning curve and its simplicity. We have chosen to keep the speed of C and the simplicity of Python. In order to compile Python to c code, some solutions have been already developed. One of the most known is Cython. In fact, Cython is "an optimising static compiler for both the Python programming language and the extended Cython programming language" [57]. More precisely, the Cython language is an extension of Python which supports "C functions and declaring C types on variables and class attributes" [57].

Cython supports all major C/C++ compilers [57]. However, it does not support most of C/C++ compiler used in the IoT realm, such as the ATMega328p used in our experiments. Thus, we propose a new parser from Python to C, specifically designed for

resource constrained devices and fully compatible with the Mongoose stack.

Developers will be able to choose any text editors or IDE they want. In addition, they can develop applications in C, C++ or Python (with the parser).

4.2 Overview

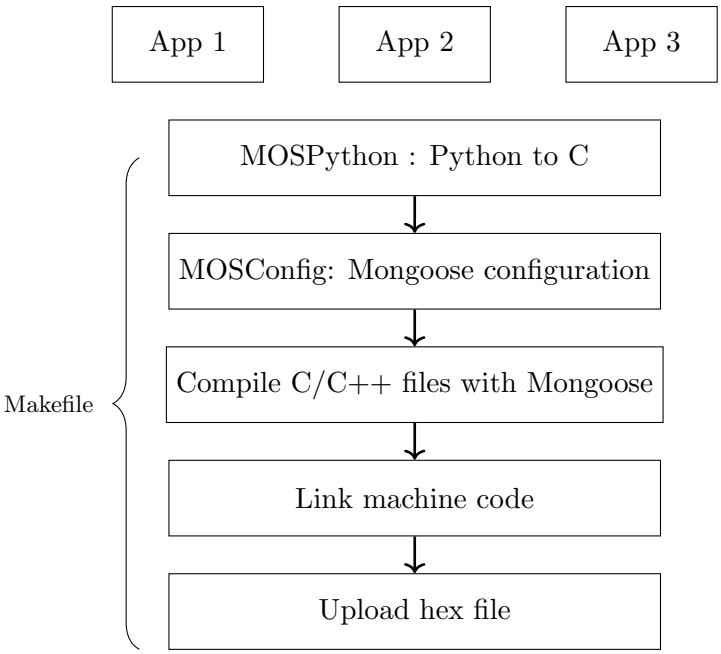


Figure 4.1: Upload a project

After writing all applications in Python, C or C++ (up to 3), developers can upload the result on the Arduino Uno thanks to this command line:

$make upload

Firstly, MOSPython will create a source directory in the working directory if it is not already done. Then, MOSPython will search all python files present in the working directory then it will translate them in C. Secondly, MOSConfig file will be used for configuring Mongoose. This file contains each application start function and what stack size each application needs. Then, all C and C++ files and Mongoose will be compiled in machine code. Finally, all machine code will be linked together in an Intel hex file and uploaded to the board. Each step is executed in the Makefile. With this approach, developers can use every text editors or IDE which supports Python 3.4.

If developers want to store serial port output in the working director, they can use this command line:

$make upload-store-log

4.3 Python Abstract Syntax Tree (AST)

MOSPython is responsible to translate Python files to C files. In order to achieve that, the abstract syntax tree is built for every file thanks to AST module provided with Python.

An abstract syntax tree (AST) is a tree structure which models a program. However, some characters present in the program won't be in the tree so it is an abstraction [58]. Each node in the AST is an object with some properties. Different classes are used to represent the entire program. Classes and properties of AST nodes for Python 3.4 can be found on **Appendix D**.

4.3.1 Example 1: function definition

A function definition can be written as:

```
def addition ( a : int, b : int) -> int:
        return a + b
```

AST can be represented as:

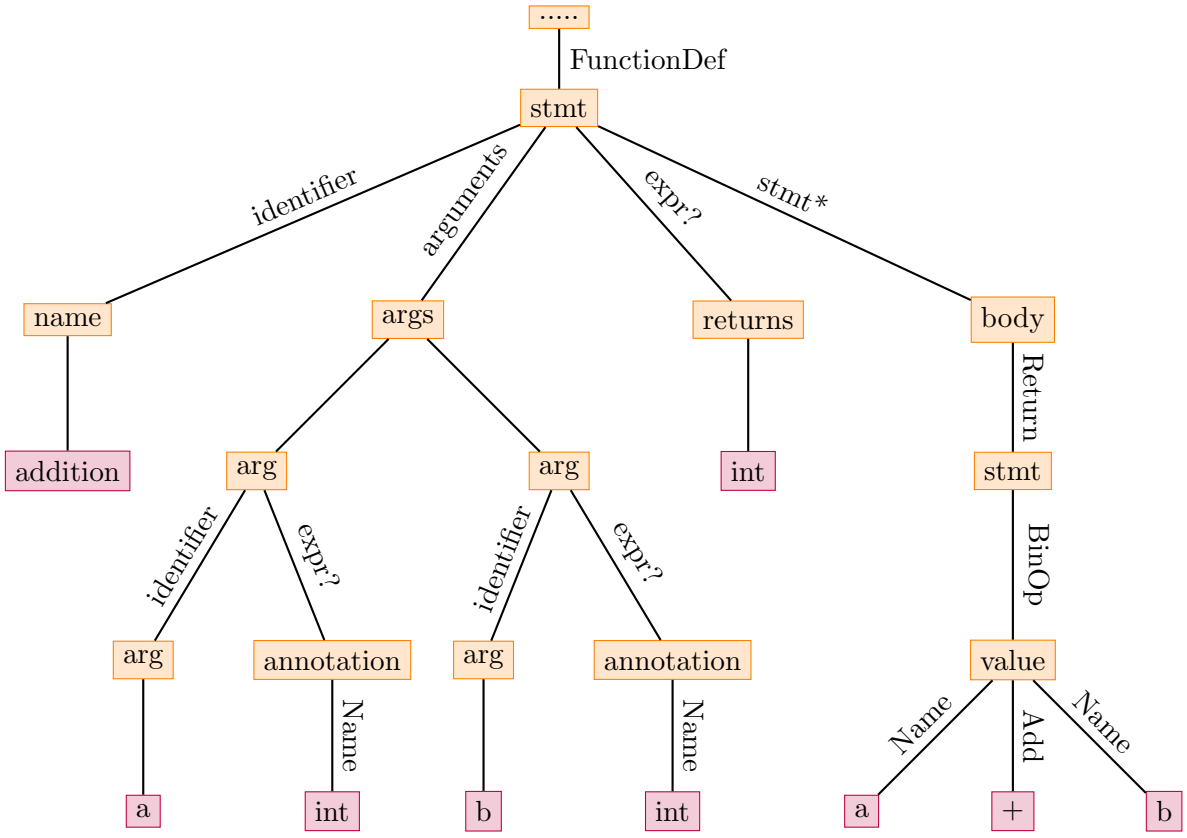


Figure 4.2: AST function definition

4.3.2 Example 2: while loop

A while loop can be written as:

```
counter = 0
while(counter < 8):
        counter = counter +1
```

AST can be represented as:

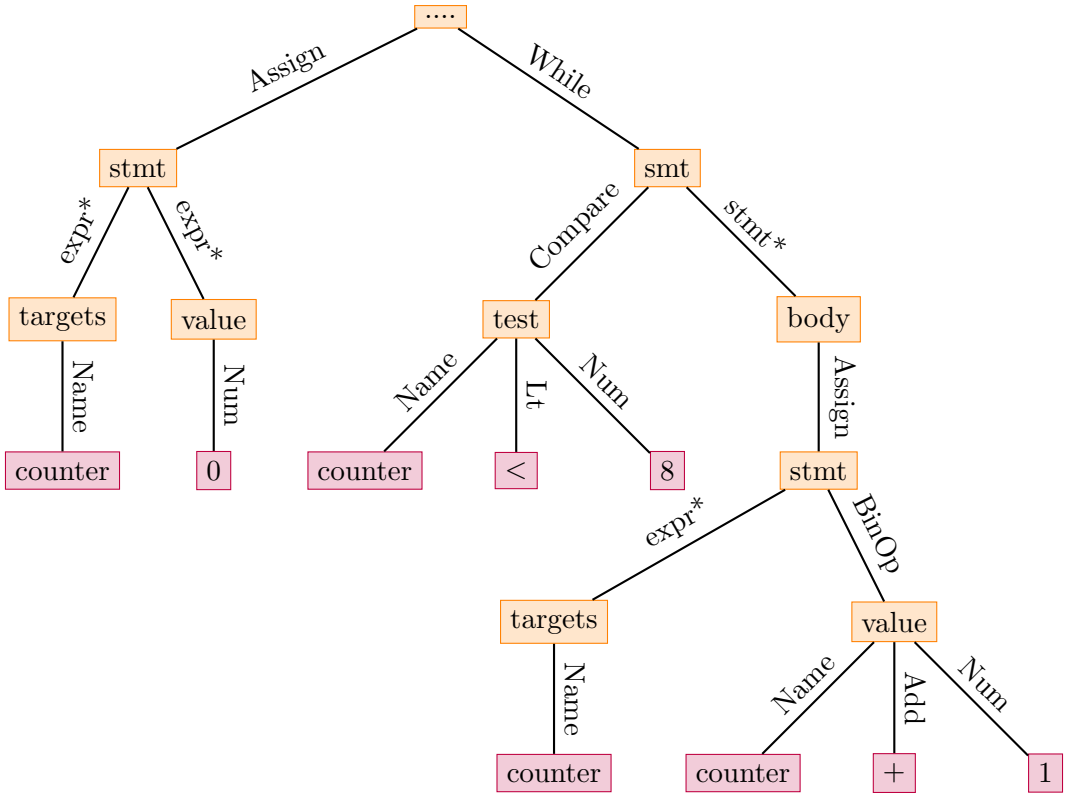


Figure 4.3: AST while loop

4.3.3 Details

Each edge label represents an AST class. Each orange rectangle represents an attribute corresponding to the relevant class. Each purple rectangle represents the actual value of each attribute. When a star is next to a class name, it represents an array. For more visibility, each array is also represented as a set of nodes. Finally, a question mark means that the corresponding attribute is optional.

If a line of code leads to a node not supported by the parser, an error is raised containing the corresponding file name. The translation is then stopped.

4.4 From Python to C: the parser

4.4.1 Header files

After the AST is built for a python file, MOSPython calls the parser on its AST. The parser goes through the tree a first time. Only function definition, variable declaration, assignment are considered. Function declarations and global variables are stored in two dictionaries which are common to every file. These two dictionaries insure that global variables and functions are declared only one time in the whole project.

Function declaration dictionary (FDD) maps function names to function expected return types. In addition, global variable dictionary (GVD) maps variable names and variable types.

With this first traversal, header files (.h) are created. These headers files are accessible by C and C++ files. Consequently, C/C++ application can use global variables and call functions written in Python and supported by the parser.

4.4.2 C files

MOSPython call the parser a second time on each AST. The parser goes through the AST. Each node reached is translated to C code. For instance, a for loop is translated from:

```
counter = 0
for x in range(100):
        counter = counter + 1
```

to:

```
uint16_t counter = 0;
uint8_t x;
for(x = 0; x < 100; x++){
        counter = counter + 1;
}
```

We use the standard AINSI C (C89) for the Python translation.

4.4.3 Inferred type

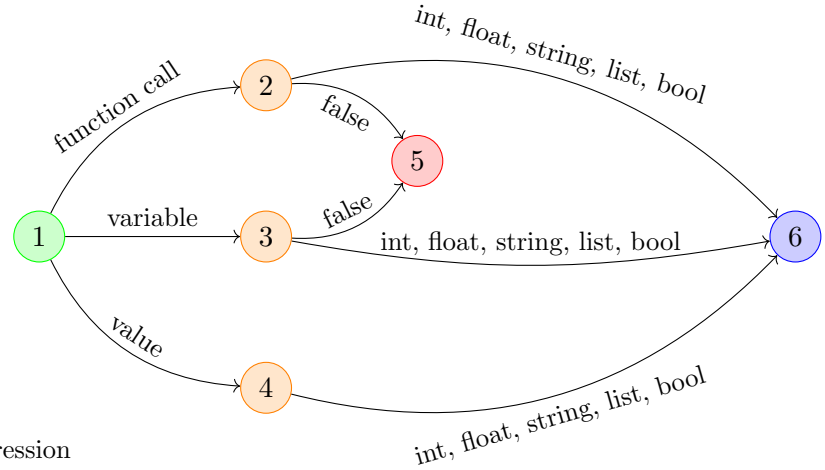
Python uses dynamic typing. However, C needs static type declaration. In order to have a working C program, we need to differentiate local variable from global variable and also detect the type of each variable.

The parser supports the following python type: int, list, float and string. Except the list, these types have an equivalent type in C. For list, it is implemented as a C struct called CList. It has a maximum capacity of 5 elements. Each element is an **integer of 16 bits**. Basic accessors for a list and list initialisation are supported by the parser. However, a global variable, which is a list, cannot be initialised with values. During run-time, if an application tries to access to a position above the size or the 5 elements limit, an error is raised by Mongoose and it stays in idle mode.

This type detection mechanism, during translation, works thanks to dictionaries. As said before, MOSPython work with two dictionaries for global variable types and also function return types. To manage local variables, a third dictionary is used: the local variables dictionary. Like global variables dictionary, it maps local variable names to their types.

MOSPython distinguishes local variables from global variables during the first traversal of the AST.

When a function definition is parsed, the location variable dictionary starts empty. The type of a variable is determined thanks to an assignment statement. This kind of statement is parsed for the two traversal of the corresponding AST. Global variables need to be assigned only to a value. These global variables are first declared outside the scope of functions. They can be declared only one time in the whole project. If no value is assigned to a global variable, the parser will consider this variable as an integer by default. In order to use a global variable in another file, the corresponding file needs to be imported.



1 : Initial Expression
2 : Check if the function is present in the FDD[1]
3 : Check if the variable is present in the LVD[2] or GVD[3]
4 : Parse the value in order to get the type (int, float, string)
5 : Error raised (function called or variable is not declared)
6 : Final type

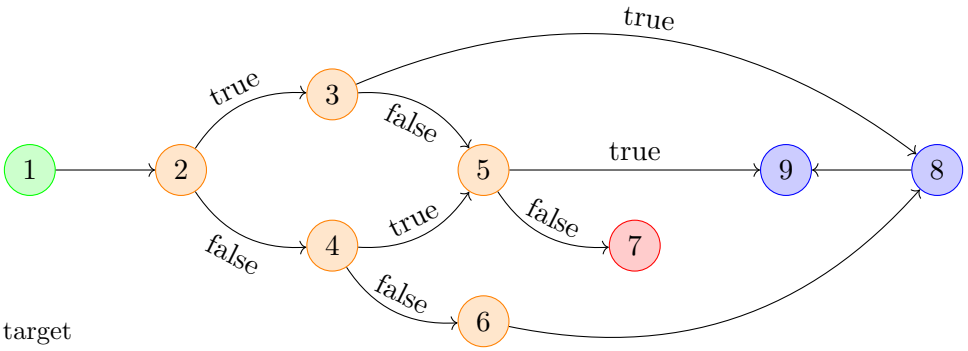
Figure 4.4: Type detection

[1]FDD: Function Declaration Dictionary
[2]LVD: Local Variable Dictionary
[3]GVD: Global Variable Dictionary

Figure 4.3.2 shows how variable types are defined by the parser. It occurs when function return value, another variable or a value is assigned for the first time to this variable. FDD, LVD and GVD are used to determine the type used for an assignment.

If an assignment statement contains a function call that was not declared in the project, the parser will stop the translation and an error is raised. In the same way, if an assignment statement works with another variable that was not declared before, the parser will stop the translation and raise an error. Each type error raised precises which variables are concerned, which file contains the error at which line. It helps the developer to fix the code.

Values types are already contained in the AST except for float and integer. More precisely, float and integer are stored in this tree as a number. Nonetheless, float is distinguished from integer by an analysis directly on the value. In addition, floats use 32 bits and integer can use 8 bits, 16 bits, 32 bits or 64 bits. In a first approach, all numbers can be considered as float. Although it is an easy approach, memory space can be lost which is problematic when there is only 2KB of RAM. In order to avoid explicit type declaration, we make the strong assumption that integer variables need at most 16 bits (unsigned integer int16_ t: 0 to 65 535) and loop counters need only 8 bits (unsigned integer uint8_ t: 0 to 255). Integer will be declared with these types during the translation.



1: Initial target
2: Check if it is a global variable (GVD[4])
3: check if this global variable was already initialized (GVD[4])
4: check if it is a local variable (LVD[5])
5: check if the target and the expression have the same type
6: Add Target to LVD[5] with the expression type
7: Raise an error
8: declare in the code the type
9: assign expression to the target

Figure 4.5: Assignment

Figure 4.3.2 represents how the parser deals with the assignment. This kind of operation lets the parser detecting the type. Each assignment statement is composed of a target variable (left side of the assignment) and an expression (right side of the assignment).

[4]GVD: Global Variable Dictionary
[5]LVD: Local Variable Dictionary

As explained before, this mechanism works with three dictionaries: FFD (function declaration), GVD (global variable), LVD (local declaration).

The first purpose is to detect if the assignment statement will lead to a type declaration. The parser checks if the target is a global variable already declared, a local variable already declared, a new global variable or a new local variable. Only new local variables and new global variables need a type declaration. The new variable is added to the right dictionary (GVD or LVD).

Then, the parser checks the type of all function calls and variables in the expression (right side of the assignment). If different types are used, an error is raised. Moreover, if the variable is already declared, target type and expression type need to be the same, otherwise an error will be raised too. These errors are declared as type errors and interrupt the translation.

4.4.4 MOSInterfaceC: Interface between python code and C/C++ code

As said above, C/C++ applications can have access to global variables and functions written in Python and supported by the parser. As a matter of fact, it is possible by importing the corresponding file (headers).

In order to let Python code have access to C/C++ functions and global variables, the parser needs to be aware of their corresponding types. It is the goal of **MOSInterfaceC** file. Nonetheless, only global variables and functions, which need to be accessible, are declared in this file. Also only bool (boolean), float, uint8_t, uint16_t, string and CList can be used for these accessible global variables and functions. Additionally, functions' return types can be also 'void'. Finally, the corresponding file needs to be imported in the python application.

MOSInterfaceC file is composed of two dictionaries: MOSInterfaceFDD and MOSInterfaceGVD. They map function names to their return type and global variables to their return type.

This dictionary looks as shown below:

```
#
# MOSInterfaceC.py
# Declare here global variables and functions which need to
# be accessible by Python applications
# MOSInterfaceFDD maps function names to their return type
# MOSInterfaceGVD maps global variable to their return type
# Only bool, float, uint8_t, uint16_t, string and CList
# can be used
```

```
MOSInterfaceFDD = {}
# C/C++ functions' declaration
MOSInterfaceFDD['anAcessibleFunction']  = 'void'
#...

MOSInterfaceGVD = {}
# C/C++ global variables
MOSInterfaceGVD['aGlobalVariable']   = 'uint16_t'

#....
```

4.5 MOSConfig: Mongoose configuration

At this stage, all Python applications have been successfully translated by MOSPython.

In order to know which application needs to be launched by Mongoose, an additional file needs to be written. It is a python file which contains a dictionary declaration. This file is common for Python, C and C++ applications.

For each application, this dictionary stores the name of the start function of the application and what stack size (SMALL, NORMAL, LARGE) is needed. The stack size meaning has been explained in the previous chapter.

This dictionary looks as shown below:

```
#
# MOSConfig.py
# Declare here what application to launch
# with Mongoose and the corresponding stack
# size (SMALL, NORMAL or LARGE)
#

APPLICATIONS = {

  'app1':{
      'startFunction' : 'app1Run',
      'stackSize': 'SMALL'
  },
  'app2':{
      'startFunction' : 'app2Run',
      'stackSize': 'NORMAL'
  },
```

```
    'app3':{
        'startFunction' : '',
        'stackSize': ''
    }

}
```

As detailed in the previous chapter, Mongoose can manage at most 3 applications. This dictionary contains a key for each possible application: 'app1', 'app2', 'app3'. If one of these keys is missing, MOSPython will raise an error and will not upload the project on the board.

Each of these keys maps to a dictionary with two keys: 'startFunction' and 'stackSize'. The first key indicates the starting function to call in order to launch the corresponding application. The second key indicates which stack size is needed. If one of these parameters is empty, the application will be ignored. In addition, if no application is launched with Mongoose, MOSPython will raise an error. The reason will be displayed with the error. As before, the project won't be uploaded to the board if an error is raised.

Mongoose will check if the total size of stacks asked can fit in the available RAM. Every global variable will reduce the available RAM. It is determined at run-time. If the total is above the available RAM, Mongoose will not launch any application and will go to idle mode. A message will be printed on the serial port.

4.6 Add drivers and features to Mongoose

New features and new drivers can be added to Mongoose. They have to be written in C or C++ directly in Mongoose source code folder. Nonetheless two files need to be completed: *HOSApi.py* and *MOSApi.py*. These functions will be accessible by Python applications. They are located in the same folder as MOSPython file.

HOSApi stores all driver functions which can be called by applications. Like MOSConfig, this file contains one dictionary which maps function names to their return type. This dictionary looks as shown below:

```
#
# HOSApi.py
# Declare here driver functions which can be
# called by applications
# It maps function name to their return type
# Only bool, float, uint8_t, uint16_t, string and CList
# can be used
#
```

```
HARDWARE = {}
# Digital functions
HARDWARE['HOSDigitalSetup']  = 'void'
HARDWARE['HOSDigitalOn']     = 'void'
HARDWARE['HOSDigitalOff']    = 'void'
HARDWARE['HOSDigitalListen'] = 'void'

# Analog functions
HARDWARE['HOSAnalogSetup']   = 'void'
HARDWARE['HOSAnalogRead']    = 'uint16_t'

#....
```

The return type has to be a C type. Only bool, float, uint8_t, uint16_t, string and CList can be used. In fact, other types are not supported by the parser. Thanks to this file, MOSPython type mechanism can work with functions from drivers.

MOSApi stores all operating system functions which can be called by an application. Like HOSApi, this file contains a dictionary which maps these functions to their return type. This dictionary looks as shown below:

```
#
# MOSApi.py
# Declare here operating system functions
# which can be called by applications
# It maps function name to their return type
# Only bool, float, uint8_t, uint16_t, string and CList
# can be used
#

MONGOOSE = {}
# Mongoose functions
MONGOOSE['MOSSleep']           = 'void'
MONGOOSE['MOSAtomicEnter']     = 'void'
MONGOOSE['MOSAtomicExit']      = 'void'
#....
```

As HOSApi, only bool, float, uint8_t, uint16_t, string and CList can be used.

When MOSPython starts the translation, these two dictionaries are concatenated with FDD (function declaration). However, each applications has to import the corresponding file. **MOSFoundation** has to be imported in every application which needs to access

to Mongoose public functions and driver public functions developed in this project. New driver header files can be included in this file. Applications will need to import only **MOSFoundation**. This header file is contained in Mongoose source code.

4.7 Limitations

MOSPython lets Mongoose work with python applications. This parser only supports a subset of Python. Therefore if developers want python functions which are not available with MOSPython, the applications will not be translated.

In addition, this parser tries to anticipate errors statically. Nonetheless, dynamic errors cannot be found with a parser. Indeed, the purpose of python applications is to let developers work in python even if they do not necessarily know C. Also some static errors may not be found by the parser. The compiler will therefore return also an error message which needs C knowledge. That is why, the translation result is readable in order to modify it directly.

Finally, all types are not supported which can limit applications' capabilities. Advanced applications should be developed in C/C++. All applications do not need to be implemented in the same language. Nonetheless, a python application cannot call functions from a C/C++ application which are not declare in **MOSInterfaceC** file. In fact, the parser will not be aware of them.

Despite these limitations, python applications have a easier syntax. Moreover, no explicit typing declaration is needed in a python application.

Chapter 5

Analysis

5.1 Methodology

Mongoose was introduced in chapter 3. In this chapter, its performance is analysed. All applications, in the following examples, have been developed in python thanks to the development environment presented in the previous chapter. The source code of each example can be found in appendix from **E** to **H**.

Mongoose is designed as a real-time operating system. We will therefore test its time constraint performance and the responsive time to hardware events. More precisely, kernel period choice chosen in chapter 3 will be tested too. Mongoose will be compared to ArdOS, a real time operating system, supported by Arduino Uno board. Contrary to Mongoose, ArdOS scheduler is based on priorities. Each application has to ask explicitly to the scheduler to execute lower priority applications. It is done through time constraints which can be expressed as delays. ArdOS has been presented in chapter 2 with Arduino environment.

Only time constraints in the first example will be tested with optimized sleeping times explained in section 3.7.3.

Inter process communication performance will be measured only on Mongoose. In fact, ArdOS uses messages queues which do not have equivalent properties.

Finally, a complex project example will be analysed only on Mongoose too. Because of application stack size, a Fourier transform cannot be performed without modifying ArdOS source code.

Performance is measured with an oscilloscope connected to a digital port on the board. Different digital ports will be used. These measures are average values based on few samples. The results will be explained with Mongoose properties. As a matter of fact, Mongoose has a predictive behaviour which is a requirement for a real-time operating system.

5.2 Mongoose and ArdOS: time constraint

5.2.1 Overview

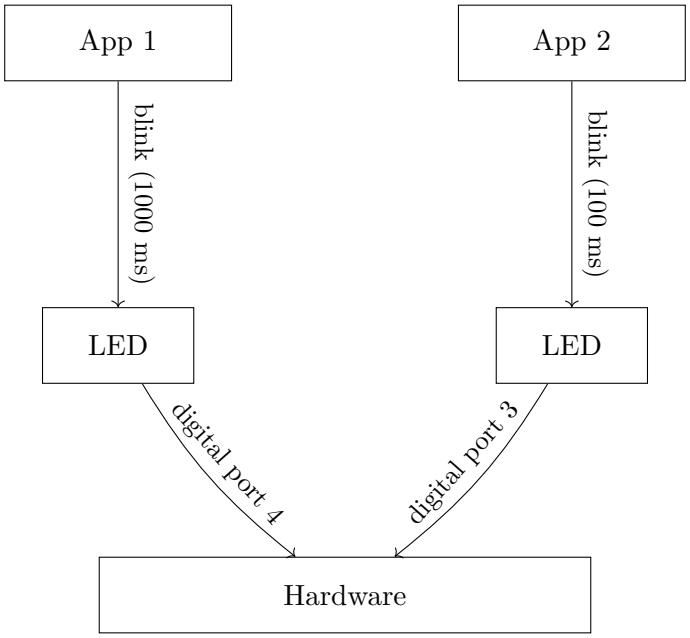


Figure 5.1: Time constraint example

The goal of this example is to measure the impact of Mongoose scheduler on time constraints. Time constraints are expressed as delays. The first application (App 1) blinks a LED every second on digital port 4 and the second application blinks a LED every 100ms on digital port 3. On one hand, applications on ArdOS use Arduino libraries in order to have access to the hardware. On the other hand, applications on Mongoose use drivers developed in this project.

5.2.2 Performance Analysis

5.2.2.1 With non-optimized sleeping time

OS	Kernel clock	LED 3	LED 4	$O_{scheduler}$	ε_{RR} (ms)
Mongoose	1 ms	103 ms	1032 ms	<4%	[-1, +3]
	2 ms	102 ms	1016 ms	<2%	[-2, +6]
	3 ms	103.6 ms	1014 ms	<1.5%	[-3, +9]
ArdOS	1 ms	102.4 ms	1022 ms		

Time constraints are measured with three different kernel periods in order to see the impact of Mongoose overhead.

As explained in chapter 3, the scheduler, based on a round robin algorithm, has also an overhead (ϵ). In fact, an application which has to be woken up could be at the end of the processes queue.

In addition, concurrency between applications introduced a computational lost ($O_{scheduler}$). At each clock interrupt, the scheduler is called which takes between 30 and 40 μs. For a time constraints Δ_t, an error ($O_{scheduler} \times \Delta_t$) is added.

We can observe that the error measured is less than the maximum theoretical error. Moreover, we can see that $O_{scheduler}$ has a direct impact on large time constraints. In the same way, ϵ has a direct impact on small time constraints. With a kernel clock of 2 ms, the error is reduced but still present.

In ArdOS case, the error is due to ArdOS scheduler overhead. More precisely, a kernel clock of 1 ms has a larger impact on time constraints. That is why, Mongoose, working at 2 ms, has smaller errors than ArdOS.

5.2.2.2 With optimized sleeping time

OS	Kernel clock	LED 3	LED 4	$O_{scheduler}$	ε_{RR} (ms)
Mongoose	1 ms	101.3 ms	1006.6 ms	<4%	[-1, +3]
	2 ms	102 ms	1006.6 ms	<2%	[-2, +6]
	3 ms	103.3 ms	1006.6 ms	<1.5%	[-3, +9]

As explained above, time constraint error is due to the round robin scheduling policy and to the scheduler overhead.

An optimisation is possible by anticipating this overhead. For every required delays T, the scheduler will consider the optimized delay T_{opt}:

$$T_{opt} = O_{scheduler} \times T \tag{5.1}$$

In the table above, we can see that long time constraints become unrelated to the kernel period. As a matter of fact, the error for long delays is essentially due to the scheduler overhead.

Nonetheless, short delays are still related to the kernel period. Indeed, the error ε_{RR}, due to round robin algorithm, is not anticipated by this optimization. More precisely, ε_{RR} is not negligible for short delays as explained in section 3.7.3.

However, this optimisation does not reduce the computational time cost related to the scheduler. Therefore, even if a kernel clock of 1 ms seems to be better than 2 ms, $O_{scheduler}$ with 2 ms is twice less than $O_{scheduler}$ with 1 ms. Despite these results, our choice of 2 ms offers a balance between time errors and computational time losses.

5.3 Mongoose and ArdOS: responsive time

5.3.1 Overview

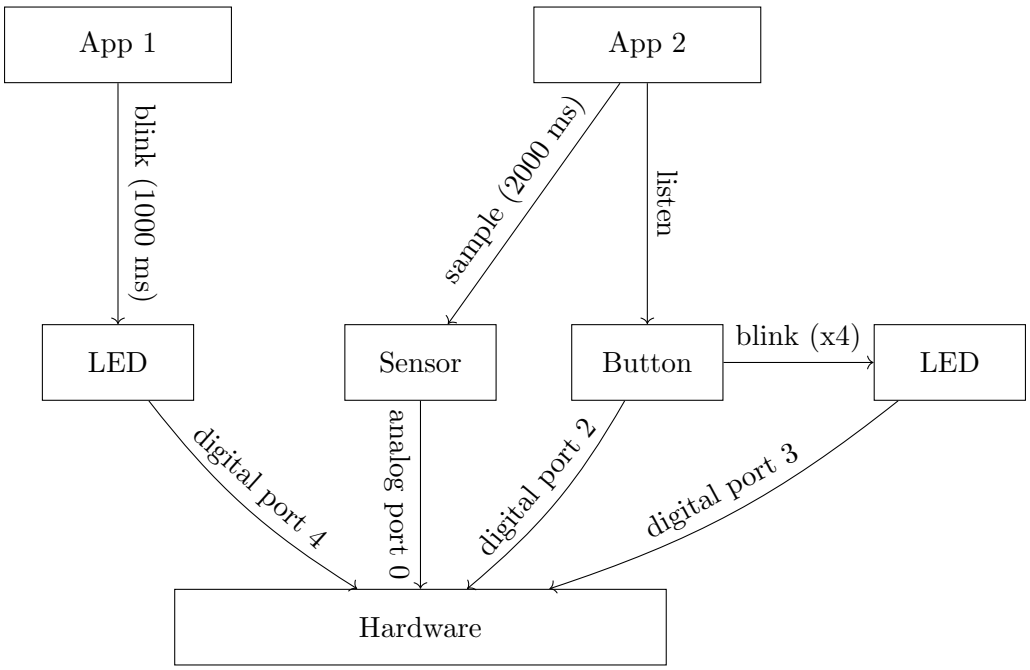


Figure 5.2: Responsive time example

The goal of this example is to measure the impact of Mongoose scheduler with hardware events. No optimisation is used for time constraints. Responsive time is the duration between when an event triggers a hardware interrupt and when its listener starts executing. The first application (App 1) blinks a LED every 2000 ms on digital port 4. The second application takes samples every 1000 ms on analog port 0. It also listens on the digital port 2 for a pressed button. When the button is pressed, this application blinks 4 times a LED on digital port 3 with a period of 500 ms. We have chosen a long listener in order to observe the loss of events (see section 3.8).

As before, on one hand, applications on ArdOS use Arduino libraries in order to have access to the hardware. On the other hand, applications on Mongoose use drivers developed in this project.

5.3.2 Performance Analysis

OS	Kernel clock	LED 3	LED 4	R_t (ms)	$O_{scheduler}$	ε_{RR} (ms)
Mongoose	1 ms	515 ms	1032 ms	7.4 ms	<4%	[-1, +3]
	2 ms	508 ms	1016 ms	7.4 ms	<2%	[-2, +6]
ArdOS	1 ms	512 ms	1022 ms	380 μs		

Two kernel clocks are used in this example in order to observe its impact on the responsive time.

ArdOs does not have an abstraction for hardware event. In order to have the same results, another application has to be developed which is only responsible for dealing with a specific hardware event. A semaphore (controlling access flag) is used for synchronizing this application and the hardware interrupt triggered.

On the contrary, Mongoose has an abstraction for hardware events through listeners and the event handler process. We can observe that this abstraction increases the responsive time (R_t). This responsive time is the same with the two kernel clocks used. Indeed, when the event occurs, the current application is stopped by the scheduler in order to resume the event handler process.

If no event occurs, the event handler process does not get a quantum of execution from the scheduler. However, it is not the case when an event has to be processed. Therefore ϵ is bigger than in the previous example.

The above results show that time constraint error for Mongoose is still bounded by the theoretical error put forward in chapter 3.

5.4 Inter process communication

5.4.1 Overview

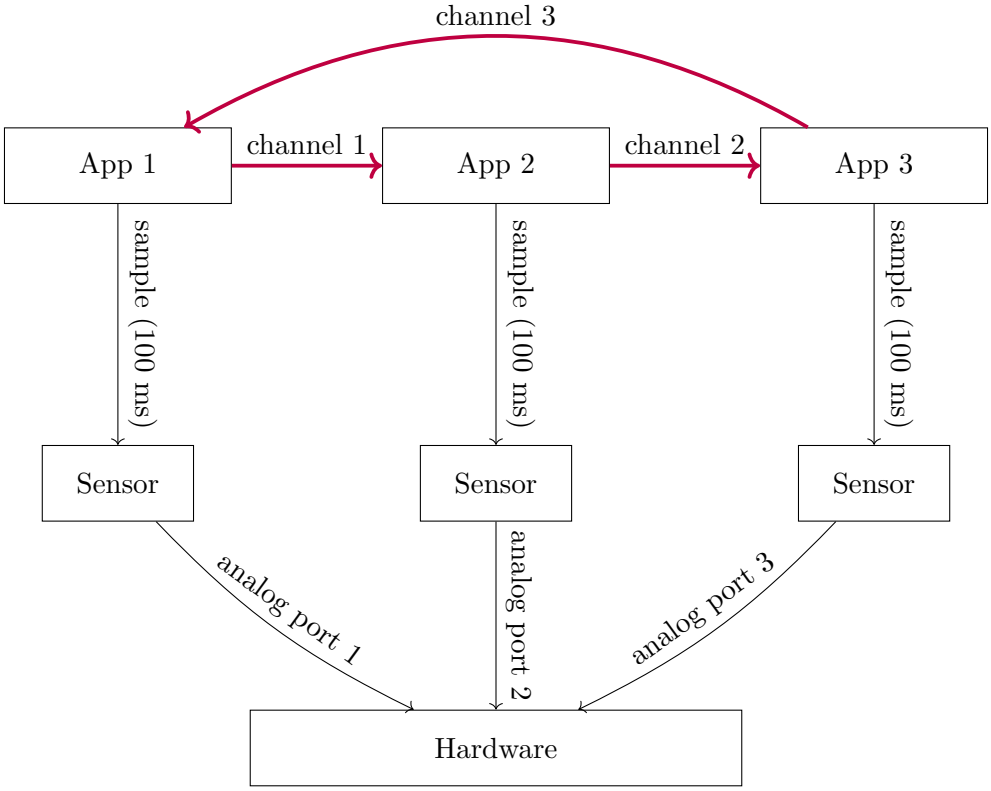


Figure 5.3: Inter process communication example

The goal of this example is to measure the impact of Mongoose inter process communication. No optimisation is used for time constraints. Communication time (C_{ti}) is the duration between the time when a message is sent on a channel i and the time when this message is received. The three applications have the same design. Every 100 ms, application i takes a sample on analog port i (S_i) then sends it on channel i (C_{ti}) and receives a message on another. This inter process communication mechanism is one-to-one. As detailed in chapter 3, these channels should be used only in one direction. That is why, three channels are used.

5.4.2 Performance Analysis

OS	Kernel clock	C_{t1}	C_{t2}	C_{t3}	$O_{scheduler}$
Mongoose	1 ms	415 μs	285 μs	120 μs	<4%
	2 ms	420 μs	290 μs	115 μs	<2%

Each channel i has a different communication time C_{ti} between sending and receiving. This difference is due to synchronisation between applications. As explained in chapter 3, a writer cannot overwrite an unread value on a channel and a reader cannot read a value which has been already read. Therefore they have to wait for each other. This waiting mechanism introduces differences between these communication times. We can observe that communication time is not directly related to the kernel clock. In fact, the inter process communication does not occur with a clock interrupt.

OS	Kernel clock	S_1	S_2	S_3	$O_{scheduler}$	ε_{RR} (ms)
Mongoose	1 ms	104.6 ms	104.6 ms	104.8 ms	<4%	[-1, +4]
	2 ms	105 ms	105 ms	105 ms	<2%	[-2, +8]

Each application i takes samples every 100 ms on its corresponding sensor (S_i). For instantce, it can be a temperature sensor or a sound sensor. We can notice with the results that sampling period is the same for each application and still bounded by the theoretical error.

5.5 Complex project with a Fourier transform

5.5.1 Overview

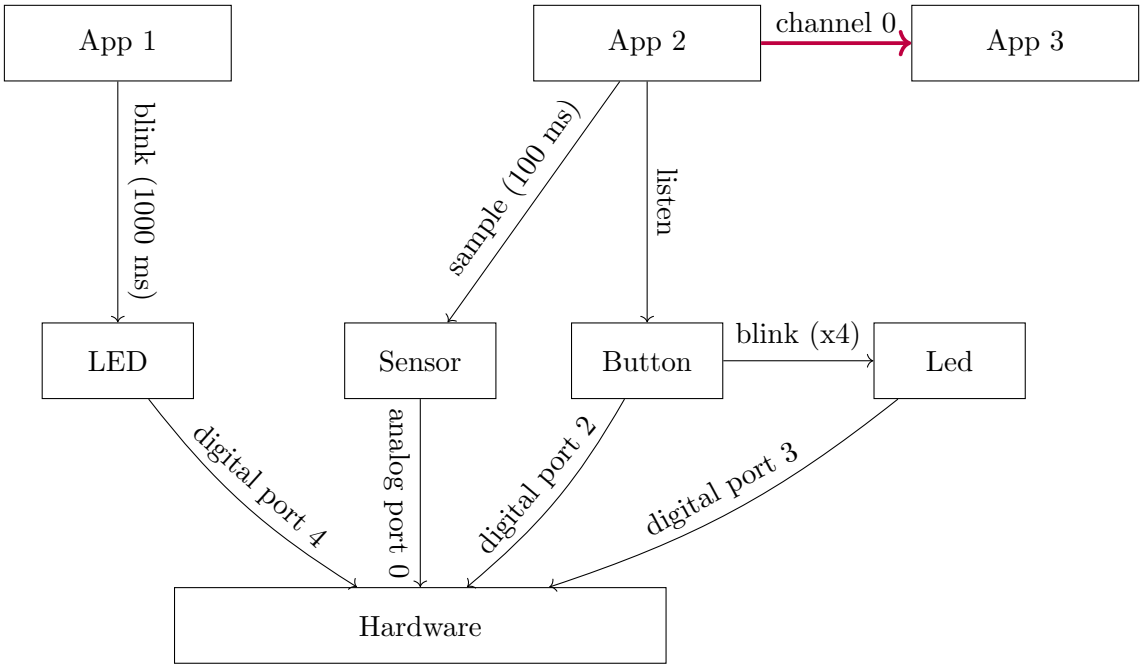


Figure 5.4: Complex project

The goal of the last example is to measure the impact of a long computational operation.

No optimisation is used for time constraints. The first application blinks a LED every 1000 ms. The second application takes samples from a sensor every 100 ms. This application sends the result to the third application on channel 0. It also listens on the digital port 2 for a pressed button. When the button is pressed, this application blinks 4 times a LED on digital port 3 with a period of 500 ms. We have also chosen a long listener in order to observe the loss of events (see section 3.8).

When application 3 receives 5 samples from application 2, the third application performs a Fourier transform (FT) with these results. The Fourier Transform (FT) is implemented as an atomic operation in order to measure the impact of a long atomic operation on time constraints and responsive time.

5.5.2 Performance Analysis

OS	Kernel clock	FT	C_{t0}	R_t	LED 3	$O_{scheduler}$	ε_{RR} (ms)
Mongoose	1 ms	7.4 ms	185 μs	139 ms	525 ms	<4%	[-1, +4]
	2 ms	7.3 ms	190 μs	126 ms	513 ms	<2%	[-2, +8]

The Fourier Transform takes almost 7.4 ms for both kernel clocks. It is the same because the operation is atomic. More precisely, no clock interrupt or hardware interrupt can occur during this operation.

The first consequence is that the clock is stopped during 7.4 ms when a Fourier Transform occurs. There are 4 processes: 3 applications and the event handler process. However, the event handler process is not always executed. So each round of the scheduler takes 3-4 ms for a kernel clock of 1 ms. The Fourier transform blocks the scheduler for 2 rounds. In the same way, each round takes 6-8 ms for a kernel clock of 2 ms. The Fourier transform blocks the scheduler for one round.

These round losses have a direct impact on time constraints. More precisely, it will introduce an additional error in the kernel clock. That is why, we can observe that the constraint time error is not bounded by the theoretical error in the first case. However, it is still the case for the kernel clock of 2 ms. We can notice also that these errors are bigger than in example 2 in both cases.

By comparison to example 2, we can observe that the responsive time is much higher than before. As a matter of fact, if an event occurs during a Fourier Transform, the corresponding interrupt flag is set but the interrupt vector is not called. Clock interrupts are also stopped. The number of lost rounds is different in the two cases. That is why, responsive time is not anymore independent from the kernel clock.

By comparison to example 3, we can see that communication time is still independent

from kernel clock. In fact, this communication time is not related to any interrupt (hardware and clock) and it occurs just before the Fourier transform and after a sample is taken.

5.6 Comparison with existing operating systems

	Tinyos	Contiki	Riot	ArdOS	Mongoose
Architecture	Monolithic kernel	Modular architecture	Microkernel architecture, modularity (RTOS)	Microkernel architecture (RTOS)	Microkernel architecture (RTOS)
Scheduling	Event driven, FIFO strategy	Event driven, FIFO strategy	Priority-based scheduling	Priority-based scheduling	Round robin scheduling
Mutlitasking	TOSThreads, partial multithreading	Protothreads, partial multithreading	Real multithreading inherited from FireKernel	Multitasking between tasks	Multitasking between processes
Programming Language	NesC dialect of C	Subset of C	C and C++	C and C++	C, C++, a subset of Python
Min RAM	<1 kB	<2 kB	1.5 kB	<0.5 kB	<0.5 kB
Min ROM	<4 kB	<30 kB	5 kB	<3 kB	<6 kB

Table 5.1: Comparison between existing operating systems and Mongoose

Chapter 6

Conclusion

6.1 Summary

Operating systems for Internet of Things are designed to work with constrained resources. Their accessibility for developers would increase the number of available connected objects and also applications.

We designed, in this project, a real-time operating system based on a preemptive round robin scheduling with a microkernel architecture. It works on Arduino Uno board with ATMega328p and 2KB of RAM. Multiple applications (up to 3) can run concurrently on the board. Each application is considered as a process with its own stack. They have equal priorities. The quantum (time slice), for the round robin scheduling policy, is 2 ms offering a balance between the scheduler overhead and the impact of the number of processes.

Time constraints are expressed as delays in applications. These delays have a bounded time error, which is related to the kernel period (quantum). However, a long atomic operation can increase this error. In addition, for long delay, this error becomes unrelated to the kernel period when the scheduler anticipates its overhead (optimized time constraints).

Mongoose provides an abstraction for hardware events through the event handler process. An application can register a listener to each type of events. The responsive time depends on event's type, the corresponding listener and applications' behaviour . Events cannot preempted each other and can be lost because of long computational listeners.

Additionally, Mongoose lets applications communicate with each other through channels. Each channel is one-to-one and cannot be used in two directions. Synchronisation between applications is done with a reader/writer concept. A reader cannot read a value, which has been already read. In the same way, a writer cannot override an unread value. Channel access does not lead to starvation or deadlock thanks to the scheduler fairness. Nonetheless, starvation and deadlock can happen because of the synchronisation mechanism.

Drivers for serial communication, digital devices, analog devices have been developed in this project. They offer basic features. Nonetheless, we try to have a clear and consistent hardware API.

Applications can be written in C, in C++ or with a subset of Python. A development environment has been developed in this project. Python applications are translated to C thanks to the parser developed in this project. Developers can develop applications with any text editor or IDE. Uploading operating is done with a makefile. Clear APIs and the possibility of Python make this operating system more developer friendly.

Mongoose has a small memory footprint: 0.5 of RAM and 6KB of ROM. In addition, time constraints are respected with an error, which can be predicted. Finally, the scheduler provides a predictive behaviour. Therefore, Mongoose respects requirements, quoted in chapter 3, of a real-time operating system and the objectives presented in chapter 1. Nonetheless, Mongoose does not manage Internet protocols.

6.2 Future work

Mongoose still needs to be more developed. In this section, we will suggest a list of future work.

6.2.1 Mongoose and internet

One of the objectives of internet of things is to let devices connect to the internet. In fact, Mongoose does not integrate any internet protocols. It would be a main improvement for this operating system. This capability should not decrease Mongoose performances and be able to work with constrained resources. With internet, Mongoose could be able to download new applications or software updates which could make it more flexible.

6.2.2 Mongoose and energy consumption

Energy consumption is also a key feature for internet of things. There is no energy saving mechanism. Thanks to such mechanism, Mongoose would be able to monitor energy consumption of the whole system. Based on this analysis, Mongoose would be able to ask a peripheral device or a process to reduce its power consumption.

6.2.3 Multitasking

Mongoose is already able to run applications concurrently. These applications, as processes, have separated stacks. A priority policy added to the scheduler may be an interesting improvement in order to reduce the time constraint error.

Moreover, each application has one thread of execution. Introducing multithreading possibilities within an application would allow Mongoose to support more complex applications. Also, shared stack between threads could be interesting in order to avoid memory space losses.

Application may crash during execution. Mongoose is not able to detect and stop a crashed application. It could be an interesting start point to make Mongoose more robust. In addition, Mongoose could load and unload application at run-time. It may offer the possibility to have more applications available on a board without launching all of them when Mongoose starts. Downloaded applications from the internet could be launched when they need to start.

6.2.4 Development environment

A development environment has been created in this project. It could be interesting to integrate this environment in the most used IDE on all main operating system (Linux, Mac OS X and Windows).

Only a subset of Python is available for developing applications. Improving the parser capabilities may let python application having more features. For instance, only basic types are supported. Developer may be able to use their own types and more advanced python features.

6.2.5 Memory Management

Mongoose uses for now only static allocation for the memory. Introducing a dynamic memory management may be interesting. Indeed, it could prevent memory space losses due to memory fragmentation.

Moreover, it could be interesting to let Mongoose and applications work with different memory (EEPROM, flash, RAM, SD card,...).

6.2.6 Cross platform

We have worked with ATMega328p for Mongoose development. So, for now, it is only compatible with boards based on this microcontroller. Nonetheless, Mongoose should be available on a large range of hardware. It will be interesting to port this operating system to all IoT platform. It will increase applications' capabilities. In fact, each platform has different hardware and different features.

Appendices

Appendix A

Serial communication registers

UCSR0A Bit #	Name	Description
bit 7	RXC0	USART Receive Complete. Set when data is available and the data register has not be read yet.
bit 6	TXC0	USART Transmit Complete. Set when all data has transmitted.
bit 5	UDRE0	USART Data Register Empty. Set when the UDR0 register is empty and new data can be transmitted.
bit 4	FE0	Frame Error. Set when next byte in the UDR0 register has a framing error.
bit 3	DOR0	Data OverRun. Set when the UDR0 was not read before the next frame arrived.
bit 2	UPE0	USART Parity Error. Set when next frame in the UDR0 has a parity error.
bit 1	U2X0	USART Double Transmission Speed. When set decreases the bit time by half doubling the speed.
bit 0	MPCM0	Multi-processor Communication Mode. When set incoming data is ignored if no addressing information is provided.

Figure A.1: UCSR0A [32]

UCSR0B Bit #	Name	Description
bit 7	RXCIE0	RX Complete Interrupt Enable. Set to allow receive complete interrupts.
bit 6	TXCIE0	TX Complete Interrupt Enable. Set to allow transmission complete interrupts.
bit 5	UDRIE0	USART Data Register Empty Interrupt Enable. Set to allow data register empty interrupts.
bit 4	RXEN0	Receiver Enable. Set to enable receiver.
bit 3	TXEN0	Transmitter enable. Set to enable transmitter.
bit 2	UCSZ20	USART Character Size 0. Used together with UCSZ01 and UCSZ00 to set data frame size. Available sizes are 5-bit (000), 6-bit (001), 7-bit (010), 8-bit (011) and 9-bit (111).
bit 1	RXB80	Receive Data Bit 8. When using 8 bit transmission the 8th bit received.
bit 0	TXB80	Transmit Data Bit 8. When using 8 bit transmission the 8th bit to be submitted.

Figure A.2: UCSR0B [32]

UCSR0C Bit #	Name	Description
bit 7 bit 6	UMSEL01 UMSEL00	USART Mode Select 1 and 0. UMSEL01 and UMSEL00 combined select the operating mode. Available modes are asynchronous (00), synchronous (01) and master SPI (11).
bit 5 bit 4	UPM01 UPM00	USART Parity Mode 1 and 0. UPM01 and UPM00 select the parity. Available modes are none (00), even (10) and odd (11).
bit 3	USBS0	USART Stop Bit Select. Set to select 1 stop bit. Unset to select 2 stop bits.
bit 2 bit 1	UCSZ01 UCSZ00	USART Character Size 1 and 0. Used together with with UCSZ20 to set data frame size. Available sizes are 5-bit (000), 6-bit (001), 7-bit (010), 8-bit (011) and 9-bit (111).
bit 0	UCPOL0	USART Clock Polarity. Set to transmit on falling edge and sample on rising edge. Unset to transmit on rising edge and sample on falling edge.

Figure A.3: UCSR0C [32]

Appendix B

Digital and analog registers

B.1 Digital registers

bits	7	6	5	4	3	2	1	0
PCMSK1	PCINT7	PCINT6	PCINT5	PCINT4	PCINT3	PCINT2	PCINT1	PCINT0
PCMSK2	-	PCINT14	PCINT13	PCINT12	PCINT11	PCINT10	PCINT9	PCINT8
PCMSK3	PCINT23	PCINT22	PCINT21	PCINT20	PCINT19	PCINT18	PCINT17	PCINT16

Figure B.1: PCMSK - Pin Change Mask Register (1,2,3) [35]

B.2 Analog registers

MUX3	MUX2	MUX1	MUX0	Analog channel
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5

Figure B.2: **ADMUX** - ADC Channels [35]

REFS1	REFS0	Voltage Reference
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 1.1V Voltage Reference with external capacitor at AREF pin

Figure B.3: **ADMUX** - Voltage Reference [35]

bits	15	14	13	12	11	10	9	8
ADCH	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2
ADCL	ADC1	ADC0	-	-	-	-	-	-
bits	7	6	5	4	3	2	1	0

Figure B.4: ADCL and ADCH – The ADC Data Register (ADLAR is set to 0) [35]

bits	15	14	13	12	11	10	9	8
ADCH	-	-	-	-	-	-	ADC9	ADC8
ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
bits	7	6	5	4	3	2	1	0

Figure B.5: ADCL and ADCH – The ADC Data Register (ADLAR is set to 1) [35]

ADPS2	ADPS1	ADPS2	ADC prescaler
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure B.6: ADC prescaler [35]

Appendix C

Timer 1 registers

CS12	CS11	CS10	Timer prescaler
0	0	0	Timer stopped
0	0	1	1
0	1	0	8
0	1	1	64
1	0	0	256
1	0	1	1024
1	0	1	External clock source : FALLING edge
1	0	1	External clock source : RAISING edge

Figure C.1: Timer prescaler [35]

Appendix D

Python Abstract Syntax Tree (AST)

Abstract Syntax Tree for Python 3.4 is shown below: [59]

```
-- ASDL's six builtin types are identifier, int, string, bytes, object, singleton

module Python
{
    mod = Module(stmt* body)
        | Interactive(stmt* body)
        | Expression(expr body)

        -- not really an actual node but useful in Jython's typesystem.
        | Suite(stmt* body)

    stmt = FunctionDef(identifier name, arguments args,
                           stmt* body, expr* decorator_list, expr? returns)
          | ClassDef(identifier name,
             expr* bases,
             keyword* keywords,
             expr? starargs,
             expr? kwargs,
             stmt* body,
             expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)

          -- use 'orelse' because else is a keyword in target languages
          | For(expr target, expr iter, stmt* body, stmt* orelse)
          | While(expr test, stmt* body, stmt* orelse)
          | If(expr test, stmt* body, stmt* orelse)
          | With(withitem* items, stmt* body)
```

```
      | Raise(expr? exc, expr? cause)
      | Try(stmt* body, excepthandler* handlers, stmt* orelse, stmt* finalbody)
      | Assert(expr test, expr? msg)

      | Import(alias* names)
      | ImportFrom(identifier? module, alias* names, int? level)

      | Global(identifier* names)
      | Nonlocal(identifier* names)
      | Expr(expr value)
      | Pass | Break | Continue

      -- XXX Jython will be different
      -- col_offset is the byte offset in the utf8 string the parser uses
      attributes (int lineno, int col_offset)

      -- BoolOp() can use left & right?
expr = BoolOp(boolop op, expr* values)
     | BinOp(expr left, operator op, expr right)
     | UnaryOp(unaryop op, expr operand)
     | Lambda(arguments args, expr body)
     | IfExp(expr test, expr body, expr orelse)
     | Dict(expr* keys, expr* values)
     | Set(expr* elts)
     | ListComp(expr elt, comprehension* generators)
     | SetComp(expr elt, comprehension* generators)
     | DictComp(expr key, expr value, comprehension* generators)
     | GeneratorExp(expr elt, comprehension* generators)
     -- the grammar constrains where yield expressions can occur
     | Yield(expr? value)
     | YieldFrom(expr value)
     -- need sequences for compare to distinguish between
     -- x < 4 < 3 and (x < 4) < 3
     | Compare(expr left, cmpop* ops, expr* comparators)
     | Call(expr func, expr* args, keyword* keywords,
         expr? starargs, expr? kwargs)
     | Num(object n) -- a number as a PyObject.
     | Str(string s) -- need to specify raw, unicode, etc?
     | Bytes(bytes s)
     | NameConstant(singleton value)
     | Ellipsis

     -- the following expression can appear in assignment context
     | Attribute(expr value, identifier attr, expr_context ctx)
     | Subscript(expr value, slice slice, expr_context ctx)
     | Starred(expr value, expr_context ctx)
     | Name(identifier id, expr_context ctx)
     | List(expr* elts, expr_context ctx)
     | Tuple(expr* elts, expr_context ctx)

      -- col_offset is the byte offset in the utf8 string the parser uses
      attributes (int lineno, int col_offset)

expr_context = Load | Store | Del | AugLoad | AugStore | Param
```

```
    slice = Slice(expr? lower, expr? upper, expr? step)
          | ExtSlice(slice* dims)
          | Index(expr value)

    boolop = And | Or

    operator = Add | Sub | Mult | Div | Mod | Pow | LShift
             | RShift | BitOr | BitXor | BitAnd | FloorDiv

    unaryop = Invert | Not | UAdd | USub

    cmpop = Eq | NotEq | Lt | LtE | Gt | GtE | Is | IsNot | In | NotIn

    comprehension = (expr target, expr iter, expr* ifs)

    excepthandler = ExceptHandler(expr? type, identifier? name, stmt* body)
                    attributes (int lineno, int col_offset)

    arguments = (arg* args, arg? vararg, arg* kwonlyargs, expr* kw_defaults,
                 arg? kwarg, expr* defaults)

    arg = (identifier arg, expr? annotation)
           attributes (int lineno, int col_offset)

    -- keyword arguments supplied to call
    keyword = (identifier arg, expr value)

    -- import name with optional 'as' alias.
    alias = (identifier name, identifier? asname)

    withitem = (expr context_expr, expr? optional_vars)
}
```

Figure D.1: AST for Python 3.4 [59]

Appendix E

Example source code: time constraints

E.1 Mongoose: Application 1

```
import MOSFoundation

def app1():
        HOSDigitalSetup(4)
        while(True):
                HOSDigitalOff(4)
                MOSSleep(1000)
                HOSDigitalOn(4)
                MOSSleep(1000)
```

E.2 Mongoose: Application 2

```
import MOSFoundation

def app2():
        HOSDigitalSetup(3)
        while(True):
                HOSDigitalOff(3)
                MOSSleep(100)
                HOSDigitalOn(3)
                MOSSleep(100)
```

E.3 ArdOS with Arduino IDE

```
#include <kernel.h>
#include <mutex.h>
#include <queue.h>
#include <sema.h>


void task1(void *p)
{
  pinMode(4, OUTPUT);
  while(1)
  {
    digitalWrite(4, HIGH);
    OSSleep(1000);
    digitalWrite(4, LOW);
    OSSleep(1000);
  }
}

void task2(void *p)
{
  pinMode(3, OUTPUT);
  while(1)
  {
    digitalWrite(3, HIGH);
    OSSleep(100);
    digitalWrite(3, LOW);
    OSSleep(100);
  }
}


#define NUM_TASKS    2

void setup() {
  OSInit(NUM_TASKS);
  OSCreateTask(0, task1, NULL);
  OSCreateTask(1, task2, NULL);
  OSRun();
}
```

Appendix F

Example source code: responsive time

F.1 Mongoose: Application 1

```
import MOSFoundation

def app1():
        HOSDigitalSetup(4)
        while(True):
                HOSDigitalOff(4)
                MOSSleep(1000)
                HOSDigitalOn(4)
                MOSSleep(1000)
```

F.2 Mongoose: Application 2

```
import MOSFoundation

def app2():
        HOSAnalogSetup()
        HOSDigitalSetup(3)
        HOSDigitalListen(2,buttonListener)
        while(True):
                MOSSleep(1000)
                result = HOSAnalogRead(0)
                MOSSleep(1000)
```

```
def buttonListener():
        for x in range(4):
                HOSDigitalOn(3)
                MOSSleep(500)
                HOSDigitalOff(3)
                MOSSleep(500)
```

F.3 ArdOS with Arduino IDE

```
#include <kernel.h>
#include <mutex.h>
#include <queue.h>
#include <sema.h>


void task1(void *p)
{
  pinMode(4, OUTPUT);
  while(1)
  {
    digitalWrite(4, HIGH);
    OSSleep(1000);
    digitalWrite(4, LOW);
    OSSleep(1000);
  }
}


void task2(void *p)
{
  while(1)
  {
    OSSleep(1000);
    unsigned int val=analogRead(0);
    OSSleep(1000);
  }
}

OSSema sem;
void int0handler()
{
  OSGiveSema(&sem);
```

```
}

void task3(void *p)
{
  pinMode(3, OUTPUT);
  while(1)
  {
    OSTakeSema(&sem);
    uint8_t counter = 4;
    for (counter = 0; counter <4; counter++)
    {
        digitalWrite(3, HIGH);
        OSSleep(500);
        digitalWrite(3, LOW);
        OSSleep(500);
    }
  }
}


#define NUM_TASKS    3

void setup() {
  OSInit(NUM_TASKS);
  OSCreateSema(&sem, 0, 1);
  attachInterrupt(0, int0handler, RISING);
  OSCreateTask(0, task1, NULL);
  OSCreateTask(1, task2, NULL);
  OSCreateTask(2, task3, NULL);
  OSRun();
}
```

Appendix G

Example source code: inter process communication

G.1 Mongoose: Application 1

```
import MOSFoundation

def app1():
        HOSAnalogSetup()
        while(True):
                MOSSleep(100)
                result=HOSAnalogRead(1)
                MOSSendMessage(result,1)
                MOSReadChannel(3)
```

G.2 Mongoose: Application 2

```
import MOSFoundation

def app2():
        HOSAnalogSetup()
        while(True):
                MOSSleep(100)
                result=HOSAnalogRead(2)
                MOSSendMessage(result,2)
                MOSReadChannel(1)
```

G.3 Mongoose: Application 3

```
import MOSFoundation

def app3():
        HOSAnalogSetup()
        while(True):
                MOSSleep(100)
                result=HOSAnalogRead(3)
                MOSSendMessage(result,3)
                MOSReadChannel(2)
```

Appendix H

Example source code: complex project with a Fourier transform

H.1 Mongoose: Application 1

```
import MOSFoundation

def app1():
        HOSAnalogSetup()
        while(True):
                MOSSleep(100)
                result=HOSAnalogRead(1)
                MOSSendMessage(result,1)
                MOSReadChannel(3)
```

H.2 Mongoose: Application 2

```
import MOSFoundation

def app2():
        HOSAnalogSetup()
        HOSDigitalSetup(3)
        HOSDigitalListen(2,buttonListener)
        while(True):
                MOSSleep(1000)
                result = HOSAnalogRead(0)
                MOSSendMessage(result,0)
```

```
def buttonListener():
        for x in range(4):
                HOSDigitalOn(3)
                MOSSleep(500)
                HOSDigitalOff(3)
                MOSSleep(500)
```

H.3 Mongoose: Application 3

```
import MOSFoundation
import math

# based on this source code :
# http://paulbourke.net/miscellaneous/dft/


def app3():
    iRe =[]
    iIm =[]
    oRe =[]
    oIm =[]
    while(True):
        for x in range(5):
            result = MOSReadChannel(0)
            iRe[x] = result
            iIm[x] = 0
        MOSAtomicEnter()
        DFT(iRe, iIm, oRe, oIm)
        MOSAtomicExit()

def intToFloat(n : int) -> float:
        return n/1.0


 def DFT(iRe : list, iIm : list, oRe :list, oIm: list):
   n = 5
   for w in range(n):
        oRe[w] = 0
        oIm[w] = 0
        a  = 0.0
        cosa = 0.0
        sina = 0.0
```

```
        for x in range(n):
            a = (2 * M_PI * w * x )/ intToFloat(n)
            cosa = cos(a)
            sina = sin(a)
            oRe[w] = oRe[w]  + iRe[x] * cosa - iIm[x] * sina
            oIm[w] = oIm[w] + iRe[x] * sina + iIm[x] * cosa

        oRe[w] = oRe[w]/n
        oIm[w] = oIm[w]/n
```

Bibliography

[1] https://hal.inria.fr/hal-00768685v3/document

[2] https://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf

[3] http://cdn.intechopen.com/pdfs-wm/12464.pdf

[4] http://electronicdesign.com/embedded/
understanding-protocols-behind-internet-things

[5] http://micrium.com/iot/iot-rtos/

[6] http://www.libelium.com/top_50_iot_sensor_applications_ranking/

[7] https://www.arduino.cc/en/Guide/Introduction

[8] https://bitbucket.org/ctank/ardos-ide/downloads/The\%20ArdOS\%20Quick\
%20Start\%20Guide\%20and\%20Tutorial.pdf

[9] http://www.cse.msu.edu/~zhangy72/docs/CSE812Project.pdf

[10] http://www.dunkels.com/adam/dunkels04contiki.pdf

[11] https://itea3.org/project/workpackage/document/download/1246/
09034-ISN-WP-1-ContikiandTinyOS%28D16%29.pdf

[12] http://www.net.in.tum.de/fileadmin/TUM/NET/NET-2012-08-2/
NET-2012-08-2_02.pdf

[13] http://www.cs.berkeley.edu/~culler/papers/ai-tinyos.pdf

[14] Brian Stuart, *Principles of Operating Systems: Design & Applications*. Delmar Cengage Learning, 1st edition, 2008.

[15] http://web.cse.ohio-state.edu/~heji/TinyOSTutorial_Mar2013.pdf

[16] http://courses.cs.washington.edu/courses/cse466/07wi/lectures/
10-tinyos.pdf

[17] http://sing.stanford.edu/pubs/sensys09-threads.pdf

[18] http://www.riot-os.org/docs/riot-infocom2013-abstract.pdf

[19] http://www.riot-os.org/#about

[20] https://developers.google.com/brillo/

[21] http://thenextweb.com/google/2015/05/28/
brillo-is-googles-operating-system-for-the-internet-of-things/

[22] http://davepowell.org/media/teaching/operating-systems/handouts/
COMP354-Lec02-KernelComparisons.pdf

[23] Andrew S. Tanenbaum, *Modern Operating Systems*. Pearson Education, third edition, 2008.

[24] http://www.freertos.org/about-RTOS.html

[25] http://www.nasa.gov/sites/default/files/482489main_4100_-_RTOS_101.pdf

[26] http://www.jameco.com/Jameco/workshop/JamecoBuilds/arduinocircuit.
html?CID=arduinobuild

[27] http://datasheet.octopart.com/A000066-Arduino-datasheet-38879526.pdf

[28] http://techterms.com/definition/sram

[29] http://www.computerhope.com/jargon/e/eeprom.htm

[30] http://www.atmel.com/Images/doc8161.pdf

[31] http://jeelabs.org/2012/06/23/low-power-uas-in-perspective/

[32] http://www.appelsiini.net/2011/simple-usart-with-avr-libc

[33] http://maxembedded.com/2011/06/port-operations-in-avr/

[34] http://www.elecrom.com/2008/02/12/avr-tutorial-2-avr-input-output/

[35] http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-8
datasheet_Complete.pdf

[36] http://thewanderingengineer.com/2014/08/11/
arduino-pin-change-interrupts/

[37] http://playground.arduino.cc/Code/Interrupts

[38] https://www.arduino.cc/en/Tutorial/AnalogInputPins

[39] http://www.engblaze.com/microcontroller-tutorial-avr-and-arduino-timer-interrupts/

[40] http://www.atmel.com/Images/doc2505.pdf

[41] http://maxembedded.com/2011/06/avr-timers-timer1/

[42] Muhammad Ali Mazidi, Sarmad Naimi, Sepehr Naimi, *The AVR Microcontroller and Embedded Systems: Using Assembly and C*. Prentice Hall, first edition, 2010.

[43] http://www.atmel.com/images/atmel-0856-avr-instruction-set-manual.pdf

[44] http://www.atmel.com/webdoc/AVRLibcReferenceManual/malloc_1malloc_intro.html

[45] http://cs.gmu.edu/~zduric/cs262/Slides/teoX.pdfs

[46] http://www.wseas.us/e-library/conferences/2015/Dubai/CEA/CEA-24.pdf

[47] http://www.freertos.org/implementation/a00016.html

[48] http://www.freertos.org/implementation/a00017.html

[49] http://jeelabs.org/2011/05/22/atmega-memory-use/

[50] www.electro-tech-online.com/attachments/avrrtos-c.73611/

[51] www.electro-tech-online.com/attachments/avrrtos-h.73610/

[52] http://beru.univ-brest.fr/~singhoff/cheddar/publications/audsley95.pdf

[53] https://csperkins.org/teaching/rtes/lecture03.pdf

[54] http://www.nngroup.com/articles/response-times-3-important-limits/

[55] http://cs.lmu.edu/~ray/notes/mutualexclusion/

[56] http://www.cs.mtu.edu/~carr/papers/jcsc02.pdf

[57] http://cython.org

[58] http://web.cse.ohio-state.edu/software/2231/web-sw2/extras/slides/21.Abstract-Syntax-Trees.pdf

[59] https://docs.python.org/3/library/ast.html