POLITECNICO DI TORINO

Master Degree in Computer Engineering

Master Degree Thesis

Implementation of a batteryless environmental datalogger

Supervisors
Prof. Massimo Poncino
Prof. Daniele Jahier Pagliari
Dr. Matteo Risso

Candidate
Andrea Sergio Ferraris
ID 277498

March 2026

† In memory of my dear Dad

Abstract

Batteryless systems, as the name says, do not host a battery and base their operations solely on energy that is harvested from the surrounding environment through light, motion, temperature gradients, radio frequencies, or other sources, temporarily stored in a supercapacitor. Companies producing electronic systems or delivering their services may have benefits in adopting this approach, especially when targeting IoT applications. Among many benefits, they might save money and avoid the problem of replacing batteries.

In this experimental thesis we implement an application whose main actor is a batteryless environmental datalogger able to transmit data and communicate remotely with a server via the long-range and ultra-low power LoRaWAN® radio communication standard. We implement the hardware of our datalogger using commercial devices. It is an MCU-based smart indoor temperature and humidity monitoring station powered only by a small photovoltaic panel. Then, we implement the firmware core running on the target MCU, which is organized around a custom scheduler whose role is to manage relevant tasks. Later, we implement a simple REST Web service capable of managing and storing data provided by the datalogger, and a LoRaWAN® gateway. The role of the gateway is to deal with incoming messages from datalogger and perform HTTP requests, as a client, to the Web service. In practice it works as a data concentrator. We also allow a user to reach the Web service through a Browser in order to display collected data. The last part of the thesis deals with the handling of intermittent computation in the datalogger, i.e., with mechanisms to suspend computation safely when a power failure occurs, and then resume it when new energy becomes available. To manage these scenarios, we resort to the Compute Through Power Loss (CTPL) API by Texas Instruments to save and restore the context or current execution state on the MCU internal non-volatile ferroelectric RAM (FRAM). We also test and evaluate different computing strategies that ensure a proper and correct execution even in the intermittent regime.

Finally, we run an experiment by deploying the overall application. Results demonstrate that the batteryless datalogger is able to save and restore the context correctly.

Acknowledgements

My special thanks go to Prof. Massimo Poncino, Prof. Daniele Jahier Pagliari and Dr. Matteo Risso for their constant support throughout the development of this thesis. Their precious advices and suggestions led me to high-quality results. The developed application, which is the core of this work, was extremely interesting, formative, sometimes challenging, but absolutely gratifying. I could not imagine nothing better: combining passion and work.

In addition, I would like to thank Politecnico di Torino, a prestigious and well-organized university. I studied in a healthy and friendly multicultural environment. I developed projects and performed laboratory sessions with people from other countries. These experiences enriched me both from the technical and human point of view.

Finally, I will always be grateful to my family for patience and understanding as well as for supporting me in every moment of my life.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Electronic systems have to be powered. It is a well-known fact because we are used to easily powering them through *wall-sockets* and *batteries*. Their suitable power source depends on requirements which are drawn up during the development. In general terms, we may say these are the two main ways of powering electronic systems, although most of portable devices are rechargeable battery-based requiring chargers that will be eventually connected to wall-sockets.

Actually, there is another option. Thanks to technological improvements over the past few years, developers can target those that are known as *batteryless systems* [1] whose main feature is to work without hosting any battery and harvest energy from the surrounding environment. This alternative is becoming particularly attractive for companies producing electronic systems with the aim of delivering their own services in different areas such as the *Internet of Things*, home automation or, more in general, the industry. Indeed, the absence of batteries and their maintenance may reduce costs significantly.

This thesis work focuses on the *implementation of a batteryless system* as case study. Specifying, designing and implementing a system is already a challenge. Developers usually resort to best practices and methods to develop a dependable solution meeting the specifications. A dependable solution performs its *mission* as you would expect because there are no failures. The assignment is harder when you target a *batteryless system* as its behavior is typically *intermittent* [7] due to the low available energy. In practice, the system must be able to save the *context* when a power failure occurs and restore it when new energy is available.

To enter the topic, we will now briefly introduce the project we developed. Then, we will discuss about motivations and possible contributions of our work. Finally, the executive summary will be presented to guide the reader through the entire document.

1.1 The project

The core of this thesis work is a project, an *application* involving hardware and software components. The case study is the implementation of a **batteryless environmental datalogger**, a MCU-based smart indoor temperature and humidity monitoring station powered only by a small photovoltaic panel, which transmits data and communicates remotely with

a server via the long-range and ultra-low power LoRaWAN® radio communication standard.

Above all, we studied the problem and focused on its complexity in order to explore different solutions. We implemented our solution resorting to a simplified *development process* which allowed us to face the complexity with a *strategy*. In particular, several activities were taken into account: *requirements*, *design*, *implementation*, *testing*, *deployment* and *managing*. Basically, the process was iterative and incremental with the goal of targeting a proof-of-concept *batteryless system* for experimental purposes.

After drawing up the requirements, the very first step was to perform a *make-or-buy* decision. We decided to select and buy some commercial devices from the market:

- a MSP-EXP430FR5994 LaunchPad™ Development Kit by Texas Instruments (TI) [22], designed around the MSP430FR5994 MCU [23], a device operating in a batteryless setting, thanks to the presence of a non-volatile Ferroelectric Random Access Memory (FRAM). Support libraries allow an application to save and restore critical system components when a power loss is detected;
- a Click board™ module by MIKROE [28], to measure temperature and humidity;
- a Grove Wio-E5 module by seeed studio [39], to work as LoRaWAN® transceiver;
- a Solar Development Kit by PowerFilm® [35], to harvest energy from the environment;
- a Wio-E5 mini development board by seeed studio [42], to implement a LoRaWAN® gateway.

Then, we started the development process. In particular, we implemented three main *actors*:

- the **datalogger**: we used a matrix board to host and connect the modules as well as some additional electronics. We wrote the firmware running on the target MCU in C/C++ thanks to Code Composer Studio (CCS) by TI (§A.1), a modern and powerful IDE. Moreover, we extensively used some libraries by TI to write a portable and robust code;
- the **gateway**: we connected the Wio-E5 mini development board to a host PC to receive data from the datalogger. We also implemented a software component to deal with incoming messages from the datalogger and perform HTTP requests, as a client, to the Web service. We developed the code in Python thanks to Visual Studio (VS) Code IDE by Microsoft (§A.2);
- the **server**: we developed a simple REST Web service capable of managing and storing data sent by gateway and datalogger. We chose the CherryPy, a *pythonic*, object-oriented web framework to build the service itself [4]. Furthermore, we selected SQLAlchemy to store and JSON format to exchange data between the gateway and the Web service. Here as well, we developed the code thanks to VS Code.

Although the application involves three main actors, we mainly focused on the datalogger. Our main goals were to implement a *batteryless system* as well as correctly handle its *intermittent* behavior.

1.2 Motivations

Implementing a *batteryless* and *intermittent* electronic system poses several challenges both from the hardware and software point of view. Developers must solve several problems for which they might not find a *feasible* solution.

One of the key points relates to energy harvesting. The hardware must include one or more harvesters capable of scavenging energy from the surrounding environment. This energy is usually stored in a *super-capacitor* temporarily. A relevant problem is to convert it to power the entire system correctly. In particular, the available energy must satisfy the requirements, that is, allow the system to perform its *mission*. Actually, harvesters can target low-power applications ranging from 10 nW to 100 mW [32], although different technologies could be evaluated according to application needs. Of course, the energy conversion process is not perfect, because part of it is lost due to the non-ideality of components. From a practical point of view, there are strict and critical relationships between energy, power and time. These relationships are a challenge to face in the field and can even be disruptive to well-designed projects. What we want to say is that some theoretical applications might fail or be easily invalidated after the implementation. Implementing a *batteryless system* means solving the problems we have briefly mentioned and many others.

The other key point is about intermittency. This kind of electronic systems are characterized by an *intermittent* computation requiring mechanisms to suspend it safely when a power failure occurs. Furthermore, they must resume the computation when new energy becomes available. In practice, the challenge is to correctly save and restore the *context* or current execution state involving CPUs, peripherals and resources. Actually, the context relates to physical and logical aspects as we will see in the next chapters. However, developers have to test and evaluate different *computing strategies* according to their needs. In particular, at least one *feasible* computing strategy should be targeted.

In this thesis, we addressed the above two challenges as follows:

- we used the Solar Development Kit by PowerFilm® to harvest energy from the environment. The kit also contained an indoor solar panel and a super-capacitor. The kit gave us the possibility to harvest, store, convert and provide energy to the rest of the system;

- we resorted to the Compute Through Power Loss (CTPL) API by TI to save and restore the context or current execution state on the selected MCU internal non-volatile FRAM;

- we implemented, tested and evaluated different computing strategies, finally converging towards one solution that we called As Soon As Possible (ASAP). Basically, since each computing strategy was implemented by running the required *tasks* thanks to a *scheduler*, we promoted the fast execution of *tasks* by minimizing the role of the *controller* and highly exploiting the *interrupt service routines*. Hence the name ASAP.

We would like to point out that our thesis work has limitations and lack of experimental validation. However, results demonstrated that the *batteryless* datalogger is able to save and restore the context correctly. Appropriate validation and verification tests would be necessary to demonstrate that the system *always* satisfies the requirements. They should

be conducted in standard environmental conditions so that results can be compared with other systems.

1.3 Contributions

This work represents a drop in the ocean. With this statement we do not want to minimize our contributions, but point out the complexity and size of the subject.

The first part of the document focuses on features and challenges of batteryless systems with a reasonable level of detail. In particular, there is a part devoted to how to cope with the intermittent behavior which represents one of major design problems. We selected and organized relevant material that might be used by the reader, as a *background*, to start creating a new project after understanding the basics of the topic.

Another contribution of this work relates to the implementation itself. Studying and understanding how components were interconnected and programmed to achieve the goal, is a good starting point for students, engineers and researchers who want to implement their own solutions. A suitable example might be the implementation of a new *computing strategy* to better control and manage the datalogger. Moreover, the system might be scaled and improved or further optimized starting from the results we achieved.

This work also deals with the basics of LoRa® technology and LoRaWAN® protocol. Such knowledge might be exploited for future projects. The same consideration holds for Web services. The thesis faces the design and implementation of a Web service with the REST spirit that can be easily reused, because this way of building distributed applications is well-known and widespread.

The main contribution relates to the intermittent behavior of the datalogger. We based our solution on libraries by TI and on a *feasible* computing strategy, but this was not enough. The hardware played a key role to make the firmware working properly. The use of such libraries in not a simplification, but a powerful way to build a robust and portable code. In the thesis we explain how to set up, configure and use the libraries to solve a complex problem, that is, saving and restoring the *context*. The reader might appreciate this solution after having learned how tricky are these operations. Obviously, ours is one solution out of many, but it is effective. The reader will be able to evaluate the quality of our project by the end of the document and try to think about advantages and disadvantages. Our spirit was to face the experimental project with open mind with the goal of improving our skills, knowledge and insight in the field.

1.4 Executive summary

To complete this introduction, we will briefly present the next chapters with the aim of guiding the reader through the entire document.

Chapter 2, *Batteryless systems*, is the starting point. It can be considered a sort of mini-guide thanks to which developers can start specifying, designing and implementing electronic systems harvesting energy from the surrounding environment through different sources. Particular emphasis is given to main features and design challenges.

Chapter 3, *Project overview*, has two main goals. First of all, it defines the datalogging process and thus the role of dataloggers. Then, it presents our project. In particular, it

describes a simplified system development process which allowed us to face the complexity of developing an application involving hardware and software components. The idea was to resort to best practices and methods to implement a modular and scalable application for experimental purposes. A section is devoted to each activity in the process: requirements, design, implementation, testing, deployment and managing.

Chapter 4, *Datalogger: hardware implementation*, follows. It explains how we implemented the hardware of the datalogger. Each module implementing a particular subsystem is presented as well as its main features and interfaces. By the end of the chapter, the reader is instructed to properly handle the datalogger to reduce the risk of damages. These recommendations are crucial, therefore a section is devoted to this topic.

Chapter 5, *Datalogger: firmware implementation*, is strictly related to the previous one. It explains how we implemented the firmware running on the target microcontroller. Essentially, our solution is based on a custom scheduler whose role is to manage relevant tasks. A task is the implementation of a functional requirement.

Chapter 6, *Server*, follows. It explains how we implemented a simple and reliable REST Web service capable of managing and storing data sent by datalogger and gateway. The design and implementation of a Web service should be faced before creating any client. This is required, because developers must decide how the service can be consumed by a client.

Chapter 7, *Gateway*, covers several topics. After introducing the LoRaWAN® protocol and the hardware module we selected from the market, it explains how we implemented a software component to deal with incoming messages from the datalogger and perform HTTP requests, as a client, to the Web service. The last section presents how to reach the service through a Browser.

Chapter 8, *Datalogger: computing strategies and optimizations*, is the most relevant one. The goal of this chapter is to explain how we managed the context and tried different computing strategies to maximize and control the intermittent behavior of the datalogger. It also explains how we performed some optimizations aiming at reducing the power consumption as much as possible.

Chapter 9, *Experimental Results*, follows. It reports the results we achieved thanks to this work. In particular, it compares the contents of the requirements document with the outcome of an experiment.

Chapter 10, *Conclusions and Future Works*, ends the journey. It suggests possible future works to developers who will be going to face a similar project or will be able to update this project. Lastly, it draws conclusions.

As a final remark, we would like to clarify a crucial aspect about the thesis. The project on which it is based required a considerable amount of time for its overall implementation. Describing all the details would have needed a huge effort and the result would have been a difficult reading to consume. We did our best to organize the material carefully and explain the development process step by step. In a nutshell, the second and third chapters are more fluent due to their contents, whereas the rest of the document follows a user/program manual style. We also included several appendixes to report how to correctly set up the development tools we used, as well as retrieve other relevant information.

Chapter 2

Batteryless systems

In this chapter, we will define and describe *batteryless systems*. We will see how to specify, design and implement an electronic system harvesting energy from the surrounding environment through different sources. Particular emphasis will be given to main features and design challenges.

2.1 Definitions

We are used to powering electronic systems thanks to *wall-sockets*. A **wall-socket** can be seen as end-point of a national electrical distribution grid which is an interconnected network for electricity delivery from producers to consumers [51]. Anytime we plug a power cable into a wall socket, we are feeding the electronic system with alternating current (AC). This type of power source is suitable for *general-purpose* computing systems such as desktop computers, workstations and servers. More in general, any electronic system requiring high currents or having a high-power consumption can be powered in this way.

Portable devices are usually powered thanks to *batteries*. A **battery** is an *energy storage device (ESD)* capable of feeding with direct current (DC). Battery technologies are mature and we can find many types out there. Each technology has benefits and drawbacks, but the goal is always the same: providing energy to the electronic system so that it can perform its *mission* as long as possible. We can distinguish between *primary* and *secondary* batteries. A primary battery is initially fully charged and discarded after use. A secondary battery can be charged, discarded into a *load* and recharged many times. Secondary batteries are *accumulators* as they accumulate and store energy through a reversible electrochemical reaction. Recharging cycles are not infinite, thus at some point, they have to be disposed [55].

Although wall-sockets and batteries are the usual way of powering electronic devices, it is possible to target those that are known as *batteryless systems* for which we can provide the following definition:

Definition 1. *A **batteryless system** is an electronic system that do not host any battery and base its operations solely on energy that is harvested from the surrounding environment through light, motion, temperature gradients, radio frequencies or other sources, temporarily stored in ESDs such as super-capacitors.*

The previous definition will be useful throughout the document, but now we might evaluate the benefits. There are many reasons why companies producing electronic systems may prefer this solution. Let us give an example. A significant number of companies deliver services to their customers by means of *internet of things* applications. An **internet of things (IoT)** is a distributed application relying on a network of embedded system-based physical objects accessing the Internet. Each physical object can be considered a **thing** as it is a *special-purpose* computing system working as end-point. Its role depends on the distributed application and the service to deliver. Data are collected locally by the thing and sent to the *cloud* over the network. The meaning of data will be inferred in the **cloud**, that is, a set of data centers composed by servers running business applications. Decisions are taken in the cloud according to data received and a response is sent back to customers. The power of such approach lies in the capillarity of things. We could imagine that, for some reason, data are sent to the cloud when available as the application does not need a continuous data-flow. In this case companies may choose a batteryless approach to implement their things. They may save money and avoid the problem of replacing batteries. This was just an example to point out the potential of batteryless systems to solve a recurring problem as IoT applications are well-accepted and widespread.

Before starting our journey, it might be useful to clarify the difference between *energy* and *power* concepts to avoid confusions. Indeed, these terms will be named many times throughout the document.

Definition 2. ***Energy (E)*** *[2] is a quantity characterizing the state (position, state of motion, temperature, deformation, etc.) of a body. The energy increases when work is performed on the body, it decreases when work is performed by the body. The work thereby causes a change of the state of the body (displacement, acceleration, increase of temperature, change of shape, etc.).*

In other words, energy measures how much work was put into the body, or was performed by it. Energy has the same SI unit of work: the **joule** [J]. It can be specified only with respect to the reference system. For our purposes, we can say that energy is the ability to cause change. As an example, think of a battery holding a certain amount of energy.

Definition 3. ***Power (P)*** *[2] is the rate energy is moved or used.*

SI unit of power is **watt** [W]. One watt is the power of a machine that performs one joule of work per second. It is useful for the characterization of machines. As an example, think of the rate at which a battery is consumed.

To evaluate how fast the energy is used or transmitted, we can introduce the **average power**:

$$P = \frac{\Delta E}{\Delta T} \tag{2.1}$$

In this case, we divide the amount of energy by the time it took to use it. If the power is time-dependent, the **instantaneous power** is:

$$p(t) = \frac{d}{dt}e(t) \tag{2.2}$$

Of course, dimensional analysis poses:

$$[W] = \frac{[J]}{[s]} \tag{2.3}$$

There is a huge theory behind energy and power concepts. Books and handbooks of Physics usually devote a large space to describe them. Here, we are interested to point out their differences although with a simple recall. However, all the previous definitions will be useful later on.

2.2 Specifications

In the previous section, we have used several times the term *application*. From an engineering point of view, an **application** is the implementation of an idea. The idea may come from, for instance, a customer need, a solution of a recurring problem or an experimental study. The application may involve one or more *actors* exchanging *data* through different *protocols*. We may say an actor is a **system** interacting with other actors, that is, the application is made up of interacting systems. Moreover, a system is usually made up of interacting sub-systems; this means that developers need to describe the complexity of a target application at different levels of abstraction.

In practice, the very first step for developers, is to specify what the target application should perform or its *mission*. The **mission** of the target application should define:

- the *environment* where the application will operate;
- the *function(s)* to be performed;
- the *duration* or operating life cycle.

Electronic and computer engineers are used to coping with kind of specifications when targeting wall-socket or battery-based applications, but those batteryless pose further challenges as we will see later on.

The **environment** plays a key role in batteryless applications. Imagine to develop an IoT application involving hundreds of batteryless things of the same type. Each thing has to measure some physical quantities from the environment via sensors in order to perform its functions. For the same reason, the environment is usually modified by the thing via actuators. Furthermore, the environment will certainly interact with the thing. Temperature, humidity and electromagnetic interference may affect or may have a negative impact on its behavior. The key point is that the interaction with the environment must be exploited to provide energy to the thing thanks to one ore more *energy harvesters*. This energy must guarantee to the thing the possibility of performing its functions correctly.

More in general, since the environment will provide energy to one or more systems of the target application, developers should start answering the following questions in the initial phase of their batteryless projects:

- will the application be indoor or outdoor?
- what kind of energy sources are available in the environment in which the target application will operate?

- are there technologies or devices able to harvest energy from those energy sources?
- which is the order of magnitude of the harvested energy from the environment compared with the one required by the target application?

The answers to these questions allow developers to perform a **feasibility check** of their projects. It is a sort of inception before starting the journey to better investigate the problem and decide whether to develop or drop the application. If the target application is feasible, developers can start writing down the *requirements*. They are the formal description of the target application to develop and resort to **design metrics**: measurable features of a system implementation. Examples of design metrics are *power consumption*, *area*, *performance*, and *cost*. More in detail, the requirements should state something like: "the power consumption must be less than or equal to 100mW". Metrics are not suggestions, but constraints that developers must meet.

The last point concerns the **development**. Developing an application is not a trivial and straightforward process. In most of the cases a *strategy* is required to face the complexity. This statement holds for simple projects too. We should clarify what we mean with *strategy*. For our purposes, we might define a **strategy** as a set of *techniques* and *best practices* guiding developers to *the right application* implementation. Actually, this topic has been highly debated in the past and many solutions have been proposed. Solutions are usually based on *development processes*. As we will see in chapter 3, a *development process* takes into account several *activities* or *disciplines* which are carried out in different *phases*. Kind of activities are: *requirements*, *design*, *implementation*, *testing*, *deployment* and *managing*.

The final message is very simple although crucial. Developers should start developing an application by structuring their mind. We perfectly know that engineers are already well-structured thanks to their study paths. The main point is that each development process has benefits and drawbacks, but allows developers to improve their skills, the ability to make decisions and learn new *techniques* and *best practices*. Furthermore, a development process helps developers to control the time and resources to target the *right* application. This statement is true both for something to sell or an experimental application. These and other concepts will be resumed in chapter 3.

In the rest of the chapter, we will focus on a hypothetical batteryless system to develop as part of an application. Above all, we will start by *modeling* the system. Then, we will see how to *harvest*, *store* and *convert* energy. Finally, we will discuss about hardware and software *implementation strategies* to deal with design *challenges*.

2.3 Modeling

Batteryless systems, as much as wall-socket and battery-based electronic systems, have the goal of powering one ore more *electrical loads*. An **electrical load** is any sub-system, electrical device or component that consumes electrical energy and converts it into another form. As an example, an electric motor converts electrical energy into mechanical energy.

Whatever is its complexity and nature, developers are interested in calculating its *input power* as shown in figure 2.1, where the load is represented by a simple bipole. The electrical parameters associated to a bipole are the current $i(t)$ and the voltage $v(t)$. Thanks to the

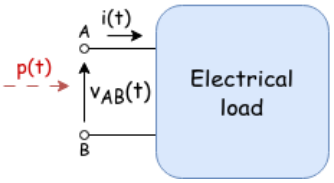


Figure 2.1: Input power of a bipole.

Kirchhoff's laws, we can state that the current entering a terminal is equal to the current exiting the other terminal; the same consideration holds for the voltage. The **input power** of the bipole can be easily calculated as:

$$p(t) = v_{AB}(t)i(t) \tag{2.4}$$

if we consider the current entering the terminal A. This result comes from electromagnetic considerations, but we are not going to enter the details [6].

For the sake of completeness, with the same conventions, we can calculate the **output power** of the bipole:

$$p(t) = -v_{AB}(t)i(t) \tag{2.5}$$

For our discussion, a relevant quantity is the **energy** supplied to the bipole from the $-\infty$ instance to a generic instant t:

$$e(t) = \int_{-\infty}^{t} v(t^{'})i(t^{'})\, dt^{'} \tag{2.6}$$

which characterizes the *passivity* of the bipole. A bipole is said to be **passive** when, whatever the voltage or current on it and for any instant in time t, the energy supplied to the bipole is never negative [6], that is:

$$e(t) \geq 0 \tag{2.7}$$

In the following, we are going to refer to a **passive electrical load** which will be always consuming energy and will never provide it.

To generalize, we can refer to figure 2.2 showing the load as the interconnection of interacting sub-systems. It is a recurring architecture, especially when dealing with IoT

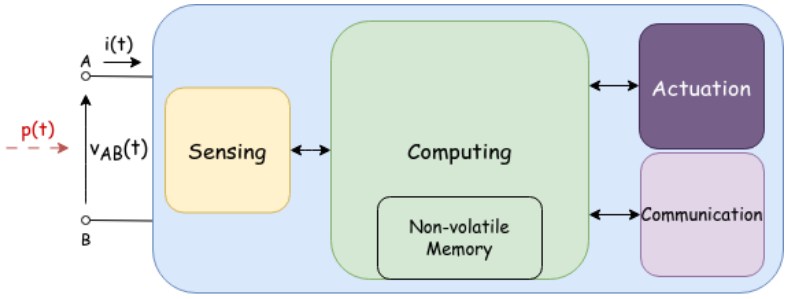


Figure 2.2: Generic architecture seen as an electrical load.

applications, involving:

- the **sensing** sub-system able to *capture* physical quantities from the environment;
- the **computing** sub-system able to perform general or special-purpose computations;
- the **actuation** sub-system able to act or modify the environment;
- the **communication** sub-system able to send and receive *data*.

We are not interested now in focusing on each sub-system as more details and further discussions will be faced later on and in the next chapters. Here, we want to focus on the input power of a generic architecture seen as an electrical load.

Another key point is about the *power supply unit*. A **power supply unit (PSU)** is a circuit that transfers energy from a primary source to the load. Although its role is to power another circuit, it is itself an electronic circuit requiring energy to work. Such detail is crucial when targeting batteryless systems.

Thus, let us introduce the **model** shown in figure 2.3. Energy can be extracted from the environment by one or more *energy harvesters*. An **harvester** or **scavenger** is any system converting a particular form of environmental energy into electrical energy, that is, any system that represents a change of state such as hot to cold, radio signals and light. The activity performed by harvesters is known as **energy harvesting**. As we will see in the next section, energy harvesting targets applications whose power consumption ranges from 10nW to 100mW [32]. Since these are low-power values, the batteryless system needs an

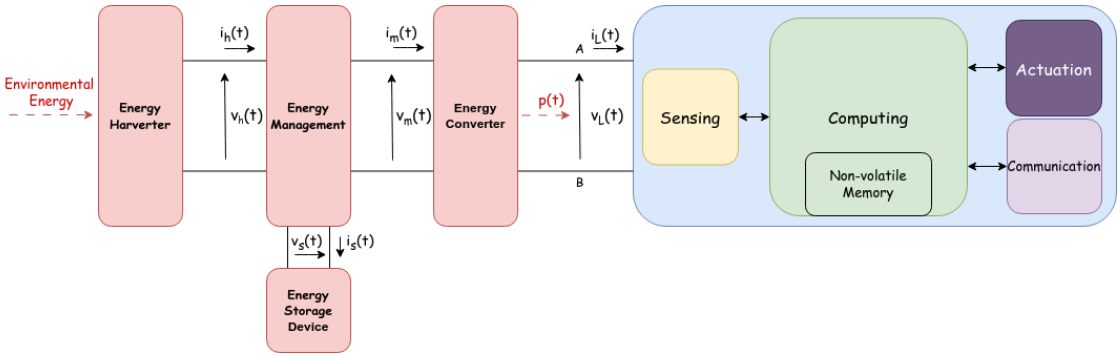


Figure 2.3: Model of a batteryless system.

energy management unit (EMU) to correctly handle harvesters and store the harvested energy into an **energy storage device (ESD)** such as a *super-capacitor* or a *conventional capacitor* working as an energy buffer [1]. When the ESD has reached a particular value, called **on threshold**, the available energy can be used to power the load. A suitable **energy converter** sub-system adapts the voltage and provides the required current to the load which starts working; obviously, the converter is enabled by the energy management unit. At this point, the load is powered and the energy management unit attempts to continuously recharge the ESD thanks to new harvested energy. If the power consumed by the load is higher than the one provided by the PSU, the ESD starts discharging because it cannot sustain the load. The load is powered until the ESD reaches a particular value called **off threshold**. In this case, the manager disables the converter and the load is no more powered.

The entire process is shown in figure 2.4. Notice that the charge-recharge cycle in time represents a normal scenario. Ideally, the ESD would allow a continuous operation, but in reality this is not feasible, at least with the current technology. Actually, an *ultra low-power* consumption batteryless system may allow very long discharge cycles.

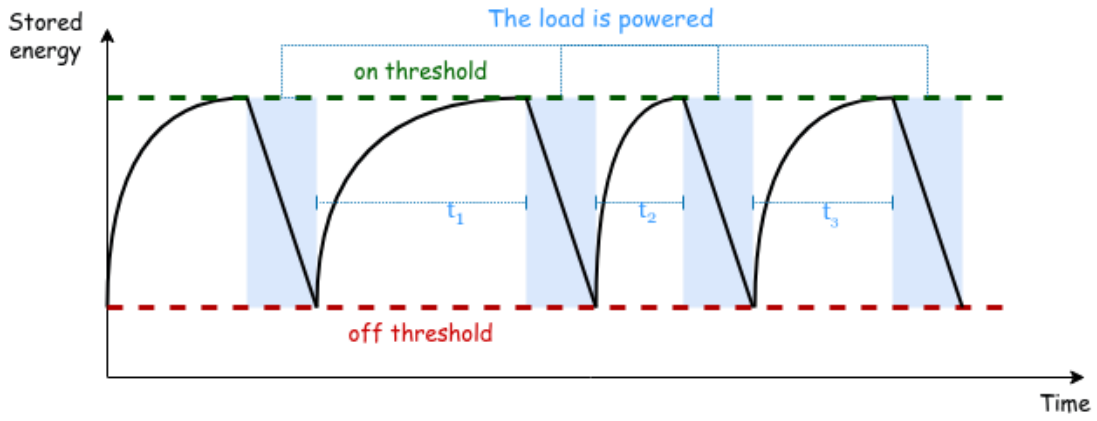


Figure 2.4: Charge-recharge cycle of the ESD [1].

2.4 Hardware implementation

In this section, we will discuss how to implement the hardware of a batteryless system. In particular, we will emphasize the topics which are more relevant for this work of thesis by referring to the model we have presented in the previous section.

2.4.1 Energy harvesters

Before presenting some energy harvesting technologies, we should consider their **application areas** referring to figure 2.5. Energy harvesters target low-power applications, that is, applications whose power consumption ranges from **10 nW** to **100 mW**. Such range could be used by developers as rule of thumb or feasibility check to decide whether energy harvesting is *acceptable* . Actually, developers might decide to use several units of the same technology or different technologies in the same project, but the complexity of the system may increase significantly, especially when implementing the energy management unit, as scavengers, in general, extract very low power and have low conversion efficiency [32].

Although not essential for our discussion, we report that battery-based systems may use energy harvesting to augment or extend the life of a battery. Such solution might be interesting for companies producing IoT devices by exploiting an hybrid technology to target *harvesting-battery-based* electronic systems [32].

In the following, we will briefly present some energy harvesting technologies to have a sufficient background. We will not cover all possible solutions, but some among them. We anticipate that the *light harvesting* technique will be emphasized as relevant in this work of thesis.

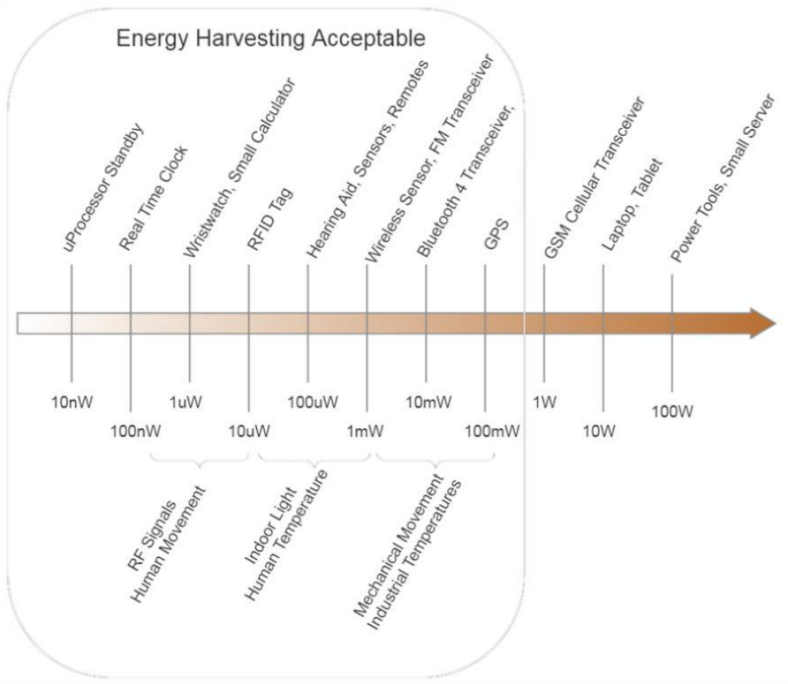


Figure 2.5: Application areas of energy harvesting [32].

Mechanical

A vibration is an energy source. More in general, mechanical strains are converted to energy through motion, vibration and even sound.

The process of converting a vibration to electrical energy can be [32]:

- **inductive**: a coil of wire moves through a magnetic field causing an electric current in wire. The vibration is coupled either to the coil or a magnet. Such method exploits the Faraday's law;
- **capacitive**: a change in capacitance causes either voltage or charge increase. Basically, a vibration causes the distance to change between two capacitive plates. The harvested energy E can be modeled as follows:

$$E = \frac{1}{2}QV^2 = \frac{Q^2}{2C} \tag{2.8}$$

  where Q is the constant charge on the plates, V is the constant voltage and C represents capacitance. Moreover, the capacitance may be represented by the lenght of the plate L_w, the relative static permittivity ϵ_0 and the distance d between plates:

$$C = \frac{\epsilon_0 L_w}{d} \tag{2.9}$$

- **piezoelectric**: a strain in piezoelectric material causes a charge separation (voltage across capacitor). Such solution highly resorts to micro electro-mechanical systems (MEMSs).

Radio Frequency (RF)

Radio frequencies are energy sources. The idea is to harvest energy from broadcast transmissions which are nearly anywhere, with service from television, cell signals, and radio. Such solution requires the selection of an appropriate antenna to capture a frequency band in the 531 kHz to 1611 kHz amplitude modulation (AM) range [32].

Thermal

Temperature is an energy source. Thermal harvesters mainly exploit the Seebeck effect, the potential generated when the junction of two dissimilar metals experiences a temperature difference.

Let S_A and S_B be the Seebeck coefficients of metal A and B as a nonlinear function of temperature, whereas T_1 and T_2 be the temperatures of the two junctions. The potential generated is calculated as:

$$V = \int_{T_1}^{T_2} S_B(T) - S_A(T)dt = (S_B - S_A)(T_2 - T_1) \tag{2.10}$$

Light

Both natural and artificial lights are energy sources. In particular, light harvesters are based on **photovoltaic effect**, which results into electric voltage/current in a material upon exposure to light. The mechanism is quite complex, but can be briefly described as follows. A photon hitting the material will break exactly an electron-hole pair of a P-N junction due to its energy. The process is cumulative because of the high number of photons; a current is generated.

Here, we will take into account *photovoltaic cells* as case study as relevant for this work of thesis. A **photovoltaic (PV) cell** [36] can be modeled, at first approximation, by the circuit shown in figure 2.6 where a light-controlled current source is in parallel with a diode. Th **open circuit voltage (Voc)** represents the forward voltage of the diode at

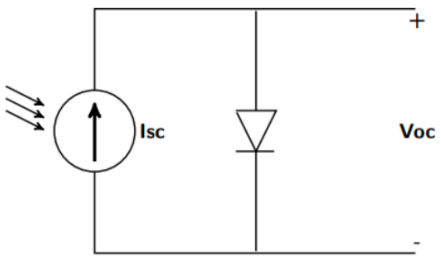


Figure 2.6: PV cell first order model [36].

no load, whereas the **short circuit current (Isc)** is the current delivered by the current source (i.e. when shorting the + and - terminals).

Figure 2.7 reports the typical **I-V curve** of a PV cell for high and low illumination levels which depend on environmental conditions. In particular, **indoor** PV cells, working mainly with artificial light, are not as efficient as **outdoor** PV cells working mainly with sunlight.

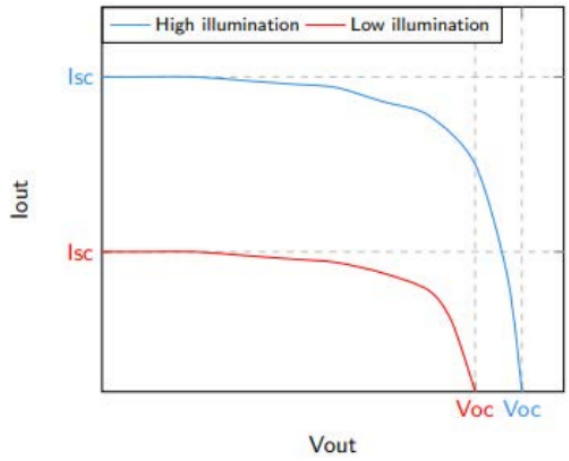


Figure 2.7: Typical I-V curve of a PV cell [36].

Solar cells are rated by their maximum power output in the form of watts [32]. As we have seen in section 2.3, the output power of a bipole is the product between the voltage and the current. The product is positive for power sources, seen as bipoles, as the current is generated and not consumed. Figure 2.8 reports the typical **power-V curve** of a PV cell. The point is, for a given technology, the **maximum power point (MPP)** is achieved at a voltage corresponding to a given ratio of the V_{oc} ranging between 70 % and 90 %. This ratio is, in first approximation, independent of the illumination level. Furthermore, the power significantly decreases with the voltage beyond the optimum V_{mpp}. It is then recommended to configure the V_{mpp}/V_{oc} ratio to be slightly lower than the theoretical optimum and therefore avoid a significant drop of performance [36].

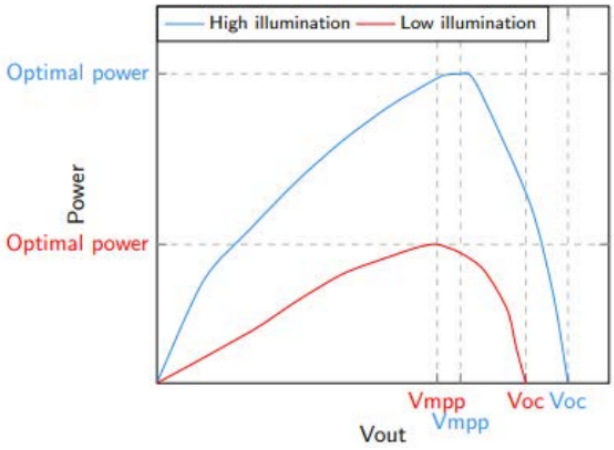


Figure 2.8: Typical power-V curve of a PV cell [36].

The **maximum power point tracking (MPPT)** is a crucial activity as, not extracting at the MPP, strongly reduces the harvester's efficiency. The *tracker* is usually part of the energy management unit as we will see later on.

2.4.2 Energy storage devices

The power management unit uses an **energy storage device (ESD)** working as energy buffer. Here, we will take into account only electrical/electrochemical based ESDs such as inductors and capacitors. Furthermore, although we are examining batteryless systems, we will devote a brief section to batteries for the sake of completeness and comparisons.

There are two qualitative figures of merit to know when dealing with ESDs: the *energy density* and *power density*. The **energy density** measures the amount of stored energy per unit weight/volume, whereas the **power density** measures how fast the energy can be drawn per unit weight/volume. Ideally, we would like both to be as large as possible, but as usual, we have trade-offs. We can compare ESDs thanks to the **Ragone plot**, shown in figure 2.9. It a 2-axis chart plotting the energy density, in Wh/kg, versus power density,

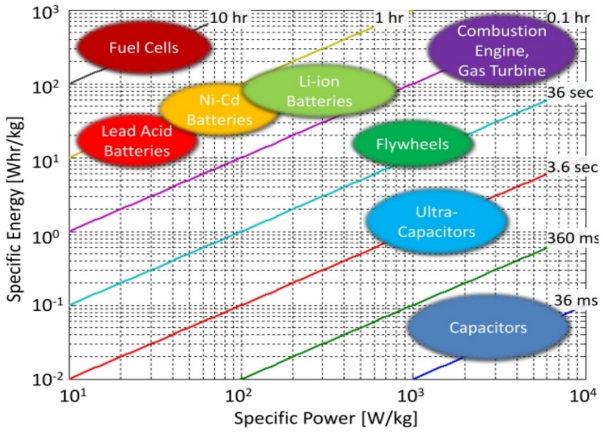


Figure 2.9: Ragone plot [12].

in W/kg; both axes are logarithmic, which allows comparing performance of very different devices [12]. Since:

$$E = E(P) \tag{2.11}$$

colored straight lines represent times, more in detail, the charge/discharge time of a given ESD in hours.

Batteries

At the beginning of this chapter, we have mentioned primary and secondary batteries. We are used to powering any portable device with batteries due to their high energy density.

By looking at figure 2.9, the reader may notice that *Li-ion* batteries have a good trade-off between energy and power densities, that's why, in the following, we are going to take this technology as reference. In a **lithium-ion (Li-ion)** battery, lithium ions physically move from the negative electrode to the positive. During recharge, the ions move back to the negative region. This is known as an *ionic movement* [32]. Since rechargeable, they are secondary batteries.

One of the most relevant figure of merit is the **capacity** whose unit of measure is Ah. If you buy and fully-charge a 1.2 A h Li-ion battery and you feed a load consuming 1.2 A, you

will totally discharge the battery in one hour. As another example, if you have a 200 A h battery and you feed a load consuming 20 A, you will totally discharge the battery in ten hours. In other words, the battery capacity is often specified in Ah at a specific **discharge rate**. An alternative unit to specify the discharge rate is the **C-rate** which is actually used as an equivalent *discharge current*. Now, let us consider again a 1.2 A h Li-ion battery. In this particular case, we say that $1\mathsf{C} = 1.2\,\mathsf{A}$ current; 1C is the **nominal discharge**. The key point is that we are allowed to describe the discharge current with multiples and sub-multiples of the nominal discharge. Indeed, 1C means one hour discharge rate, 2C means half a hour discharge rate, whereas C/2 means 2 hours discharge rate and so on. In practice, we have a double interpretation: time and current.

Inductors

An ideal **inductor** is a passive component characterized by its **inductance (L)** measured in Henry [H] (SI). The relationship between voltage and current can be defined as follows:

$$v(t) = L\,\frac{di(t)}{dt} \tag{2.12}$$

where L is a constant parameter. From the equation, we infer that an inductor is a **linear** bipole with **memory**. Furthermore, if the current flowing in the inductor is constant in time, the voltage between its poles is zero; in this case it behaves as a short circuit [6].

Inductors are **reactive** bipoles as they can store and release energy. The **input power** of an inductor can be easily calculated as:

$$p(t) = v(t)\,i(t) = L\,\frac{di(t)}{dt}\,i(t) = \frac{d}{dt}\left[\frac{1}{2}\,L\,i^2(t)\right] \tag{2.13}$$

which can be considered the temporal derivative of energy:

$$e(t) = \frac{1}{2}\,L\,i^2(t) \tag{2.14}$$

In practice, the input power varies its own energy [6].

Inductors are usually manufactured by wrapping a wire of conductive material around a core which must be made of a suitable material. This technology is not so accurate and cannot be used to target integrated circuits. However, in electronics, many applications use inductors as discrete components such as switching regulators and oscillators [6].

We do not want to discuss too much about inductors here. We are simply informing the reader that they can be used to target a batteryless application with the aim of storing energy, although not shown in figure 2.9.

Conventional capacitors

An ideal **capacitor** is a passive component characterized by its **capacitance (C)** measured in Farad [F]. The relationship between voltage and current can be defined as follows:

$$i(t) = C\,\frac{dv(t)}{dt} \tag{2.15}$$

where C is a constant parameter. From the equation, we can infer that a capacitor is a **linear** bipole with **memory**. Furthermore, if the voltage between its poles is constant in time, its current is zero; in this case it behaves as an *open circuit* [6].

Also capacitors are **reactive** bipoles as they can store and release energy. The **input power** of a capacitor can be easily calculated as:

$$p(t) = v(t)\ i(t) = v(t)\ C\ \frac{dv(t)}{dt} = \frac{d}{dt}\left[\frac{1}{2}\ C\ v^2(t)\right] \tag{2.16}$$

which can be considered the temporal derivative of energy:

$$e(t) = \frac{1}{2}\ C\ v^2(t) \tag{2.17}$$

In practice, the input power varies its own energy [6].

Capacitors are usually manufactured by separating two conductors with appropriate material. This technology is highly precise and can be used to target integrated circuits. Actually, developers can evaluate many technologies out there which are suitable for different applications. Capacitors are practically present in any electronic system as they perform common activities such as filtering, decoupling and by-passing sub-systems [6].

Let us now take a look again to figure 2.9. Conventional capacitors have very low energy density, but very high power density. A typical capacitor will have an energy density of somewhere between 0.01 Wh/kg [32]. This order of magnitude is not enough to target most of batteryless systems, although developers may consider this solution to store energy.

Super-capacitors

A better solution to store more energy is to use *super-capacitors* or *ultra-capacitors*. A **super-capacitor** stores energy at significantly higher volumes than conventional capacitors as its energy density is about 1 to 10 Wh/kg. This order of magnitude is more closer, though still far away, to those of batteries that can be on the order of 200 Wh/kg [32]. Figure 2.9 shows that super-capacitors are more closer to ideal energy storage devices which allow high power and acceptable energy densities.

Like a capacitor, energy is stored electrostatically on a plate. Usually, super-capacitors are made of fairly exotic materials like graphene, which can impact on overall cost. The most relevant advantage is about charging: they can reach their full potential in seconds. Main forms are:

- **electric double-layer capacitors (EDLC)** - use an activated carbon electrode and store energy electrostatically;
- **pseudocapacitors** - use a transition metal oxide and electrochemical charge transfer.

In general, the main issues with super-capacitors are the leakage current and the cost. However, nowadays, they are targeted by most of batteryless applications [32]. We will return to this topic in section 4.5.

To sum up, it could be a good idea to compare Li-ion battery with super-capacitor technologies. As we have seen, Li-ion batteries are widely used in battery-based applications, whereas super-capacitors have a good trade-off between energy and power densities to target batteryless applications. Table 2.1 reports the main data.

Category	Li-ion battery	Super-capacitor
Energy density	200 Wh/kg	8-10 Wh/kg
Charge/Discharge cycles	100 to 1000	Nearly infinite
Charge/Discharge time	1 h to 10 h	ms to s
Operational temperature	−20 °C to 65 °C	−40 °C to 85 °C
Operational voltage	1.2 V to 4.2 V	1 V to 3 V
Power delivery	Constant voltage over time	Linear or exponential decay
Charge rate	Very slow: 40C/x	Very fast: 1500C/x
Operational life	0.5 to 5 years	5 to 20 years
Form factor	Very small	Large
Cost ($/KWh)	Low: $250 to $1000	High: $10000

Table 2.1: Comparison between Li-ion battery and super-capacitor technologies [32].

2.4.3 Energy converters

The **energy converter** sub-system adapts the voltage and provides the required current to the load. In general, PSU developers might target *linear* or *switching* converters whose behavior and main features will be briefly described in this section.

Linear

Electronic engineers often design PSUs. Basically, after plugging a cable into a wall-socket, the alternating input voltage is converted to feed the load with the required output voltage. The process usually involves a sequence of stages: the main switch and fuse, input EMC filter, transformer, rectifier and filter. After the filter, there is an average **constant output voltage** V_{avg} with a **residual ripple** ΔV. If we consider a full wave rectifier and a filter implemented by a simple capacitor, the ripple can be calculated as [5]:

$$\Delta V = \frac{I_{load}}{C} \cdot \frac{T}{2} \tag{2.18}$$

where I_{load} is the current required by the load and T is the period of the input signal. In practice, the ripple mostly depends on the filter capacitor. If the ripple is not acceptable, as it does not satisfy the specifications, you need to add a voltage *regulator*.

Figure 2.10 shows a three terminal **linear regulator**. The input voltage is $V_{in} =$

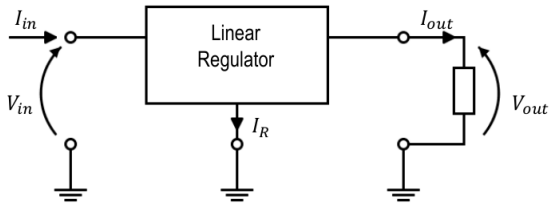


Figure 2.10: Three terminal linear regulator [5].

$V_{avg} + \Delta V$, whereas V_{out} is the regulated voltage. I_{out} is the output current required by the

load. Furthermore, it is always $V_{out} < V_{in}$. The regulator needs the I_R current to work:

$$I_R = I_{in} - I_{out} \tag{2.19}$$

Another important parameter is the **drop-out voltage** that developers can retrieve from the device datasheet. To correctly work, the following condition must hold [5]:

$$V_{DO} = (V_{in} - V_{out}) \geq V_{DO_{min}} \tag{2.20}$$

Silicon manufacturers produce low drop-out linear regulators with the aim of targeting low voltage applications.

The drop-out voltage plays a key role for the **average efficiency** [5]:

$$\eta_{avg} = \frac{P_{out}}{P_{in}} = \frac{V_{out}I_{out}}{V_{in}I_{in}} \simeq \frac{V_{out}}{V_{in}} = \frac{V_{avg} - V_{DO}}{V_{avg}} = 1 - \frac{V_{DO}}{V_{avg}} \tag{2.21}$$

The **maximum efficiency** can be calculated as [5]:

$$\eta_{max} = 1 - \frac{\frac{\Delta V}{2} + V_{DO_{min}}}{V_{avg}} \tag{2.22}$$

Linear regulators are usually less efficient than switching regulators as we will see briefly.

Switching

Given a constant input voltage, a **switching regulator** generates the target output voltage using a switching architecture. Switches are used to create a square wave. The main parameters are the **duty cycle** δ and the **switching frequency** f_{sw}. A filter, composed by reactive components such as inductors and capacitors, extracts the average value. The filter has no power loss and high efficiency. A **feedback network** implements the regulation. Switching are more efficient than linear regulators although sensitivities are worse [5].

The **Buck** or **Step-Down** regulator, shown in figure 2.11, produces an output voltage lower than the input voltage: $V_{out} < V_{in}$:

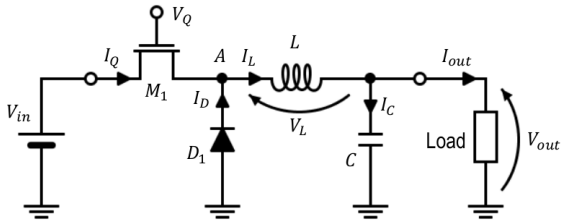


Figure 2.11: Buck (Step-Down) regulator [5].

The **Boost** or **Step-UP** regulator, shown in figure 2.12, produces an output voltage higher than the input voltage: $V_{out} > V_{in}$:

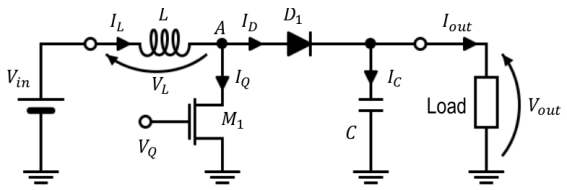


Figure 2.12: Boost (Step-Up) regulator [5].

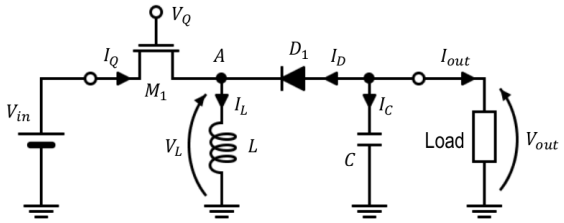


Figure 2.13: Buck-Boost regulator [5].

The **Buck-Boost** regulator, shown in figure 2.13, produces a negative output voltage of any value with a positive input voltage: $V_{out} < 0$.

Actually, the previous circuits implement a **DC-DC converter**; you should add the feedback network to have the voltage regulation. We decided not to show the feedback network for the sake of simplicity. Moreover, we did not describe how to choose components of the switching regulator as it requires a detailed analysis that we cannot face here. However, the reader might consult a power electronics book for this purpose or [5]. We have recalled linear and switching regulators with the goal of better understanding the next topic.

2.4.4 Energy management unit

The **energy management unit (EMU)** is the most complex sub-system to implement. It correctly handles harvesters and stores energy into ESDs with the goal of powering the load. The EMU uses regulators to adapt voltages and currents.

The implementation is not so straightforward. That's why we decided to present a use case: the **AEM10941** integrated EMU by **e-peas**. We will see how to use it in section 4.5, but now we are going to describe its architecture referring to figure 2.14.

The IC integrates several sub-systems whose main are:

- a **finite state machine (FSM)** supervising the system and performing the required operations. The FSM receives input signals for configurations and status and produces output signals to control and enable the other sub-systems;
- a cascade of two **regulated switching converters**, namely the boost converter and the buck converter, both with high power conversion efficiencies;
- two power supplying **LDO regulators** called LVOUT and HVOUT;
- a **MPPT controller** performing the maximum power point tracking.

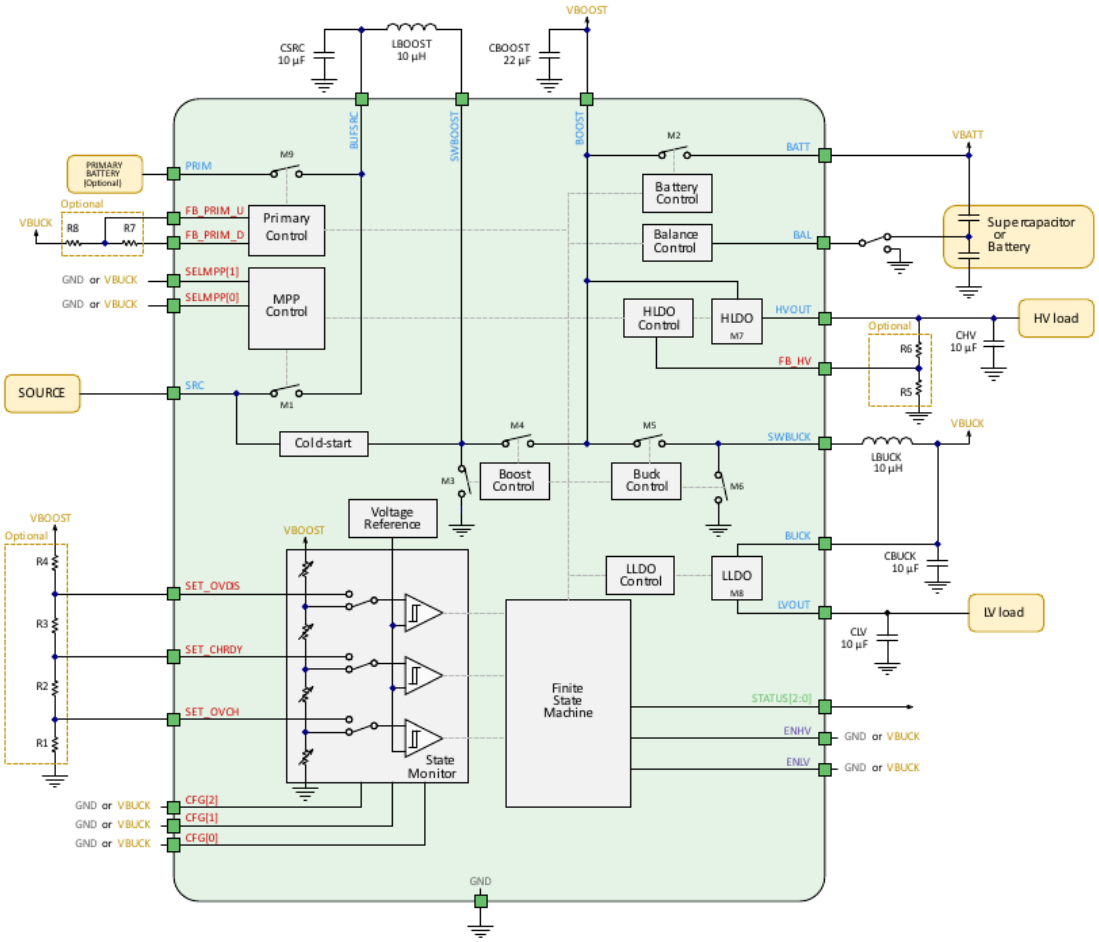


Figure 2.14: AEM10941 functional block diagram [11].

External components are needed to correctly set up and use the IC. Obviously, the IC allows targeting an efficient EMU thanks to the benefits of the integration process, such as reducing distances among components and reducing leakage currents. The AEM10941 is capable of charging an ESD such as a battery or **super-capacitor** connected to BATT from an energy source, such as a **PV cell** connected to SRC, as well as supplying loads at different operating voltages through LVOUT and HVOUT LDO regulators [11].

Although some terms will be referring to figure 2.14, let us focus on figure 2.15 showing a simplified schematic view. The idea is to understand how a load can be powered by a super-capacitor starting from the energy harvested by a PV cell.

The FSM takes decisions according to the current **operating mode** [11]:

- let us consider the initial state in which the super-capacitor, connected to BATT pin, is deeply discharged and there is no available energy to be harvested; the FSM is in **Deep Sleep** mode;
- since the IC integrates a **cold-start circuit**, the FSM enters the **Wake-up** mode as soon as the required voltage of **380 mV** and the required power of **3 µW** becomes

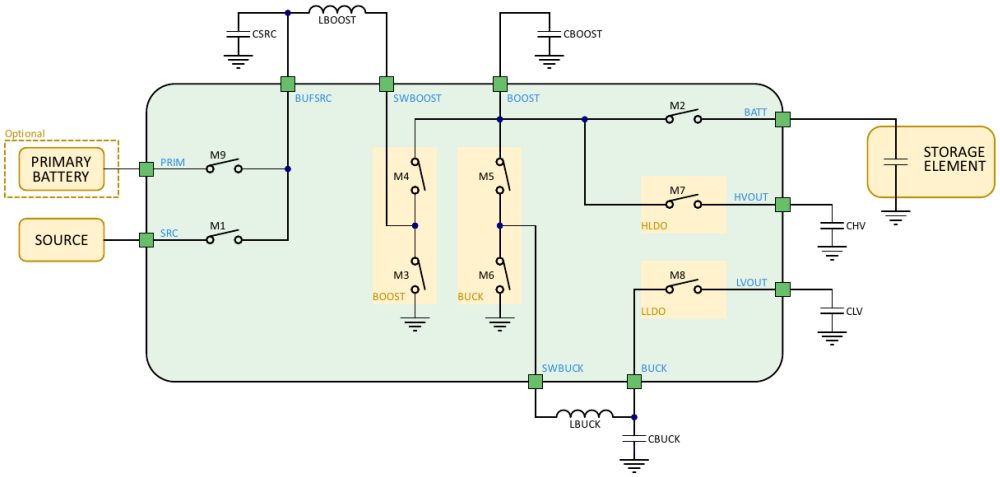


Figure 2.15: AEM10941 simplified schematic view [11].

available on SRC pin, connected to the positive terminal of the PV cell. At this point, the boost converter starts charging the super-capacitor, in the range of 2.2 V to 4.5 V according to the system configuration. The boost converter is internally implemented by M3 and M4 switching transistors, controlled by the FSM, and externally implemented by the L_{BOOST} inductor and C_{BOOST} capacitor. The buck converter lowers the output voltage of the boost converter to 2.2 V. It is internally implemented by M5 and M6 switching transistors, controlled by the FSM, and externally implemented by the L_{BUCK} inductor and C_{BUCK} capacitor. In general, the MPPT control circuit periodically measures the open-circuit voltage of the harvester and defines the optimal SRC voltage regulation. This activity can be performed by disconnecting the PV cell on SRC pin from the BUFSRC pin with the transistor M1. In other words, the FSM allows the boost converter to charge the super-capacitor from the PV cell by modulating the conductance of M1 and M2 switching transistors. Moreover, during the charging, both LDOs are disabled;

- when V_{BATT} reaches V_{CHRDY}, the FSM enters the **Normal** mode and LDOs can be enabled by the user to power the load. The FSM stays in this mode as long as the following condition is met:

$$V_{OVDIS} < V_{BATT} < V_{OVCH} \tag{2.23}$$

- if $V_{BATT} > V_{OVCH}$, the FSM enters the **Overcharge** mode because the PV cell provides more power than the load consumes. Since the super-capacitor charge is complete, the FSM prevents damages to the ESD and internal circuitry;

- if $V_{BATT} < V_{OVDIS}$, the FSM enters the **Shutdown** mode. Both LDO regulators, powering the load, remain enable during the next **600 ms**. If V_{BATT} recovers to V_{OVDIS} within 600 ms the FSM switches back to Normal mode, otherwise it disables LDOs and enters the Deep Sleep mode in order to start from scratch. Figure 2.16 reports the diagram implemented by the FSM that we have described.

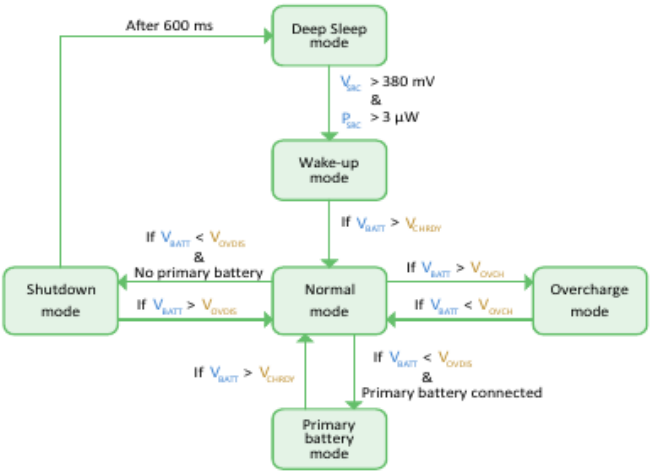


Figure 2.16: AEM10941 FSM [11].

In this section, we have explained how to implement the EMU of a batteryless system resorting to a use case. We wanted to point out how much can be complex the implementation with the aim of achieving good results. In practice, an IC is the best solution compared to a modular approach as the goal is to reduce distances among components and leakage currents. The power consumption is key. We will further present and configure the energy management unit in section 4.5.

2.5 Software implementation

In this section, we will discuss how to implement the software of a batteryless system. In particular, we will present some *computing models* aiming at managing its *intermittent* behavior.

2.5.1 Intermittency

Batteryless systems are characterized by an **intermittent** behavior or execution. Such behavior occurs as harvested energy is highly variable and unpredictable, and ESDs, such as conventinal capacitor and super-capacitors, are small. In practice, **energy failures** are frequent [1] as load power demand exceeds the harvested energy. The ESD undergoes charge-recharge cycles as we have already reported in figure 2.4.

Basically, the intermittent computation requires mechanisms to *suspend* it safely when a power failure occurs. Furthermore, they must *resume* the computation when new energy becomes available. The challenge is to correctly *save* and *restore* the *context*, that is, the *current execution state* involving CPUs, peripherals and resources. Let us better discuss this crucial point. In general, a power failure clears the system's volatile state such as main memory and register file. That's why a batteryless systems must **suspend** the computation and **save** the *context* or *computational progress* in a non-volatile memory when a power failure occurs. When energy is back, the systems must **resume** the computation and

restore the *context* instead of performing a global restart. Thus, developers must write programs bearing in mind the following issues [1]:

- restartability and progress - the concepts we have introduced previously. The key is deciding which volatile state to preserve before an energy failure;
- memory consistency - since non-volatile memory manipulation is non-idempotent, that is operations including WAR dependencies might re-execute, they might leave a different result in non-volatile memory, producing erroneous and inconsistent outputs;
- timeliness - the amount of time between the system resumes and suspends represents a **session** in which the system must be able to perform its mission. The time between sessions can range from hundreds of microseconds to days. Such range might produce unusable results if not correctly evaluated;
- peripherals - a power failure can leave a peripheral in an unrecoverable state.

We should better clarify what we mean with *context*. The **context** represents the current execution state or computational progress during a power failure. As an example, we might imagine an algorithm which is interrupted during its execution. We would like to restore the algorithm, after resuming, exactly at the point where it was interrupted. This issue involves *physical* and *logical* aspects:

- the **physical** context relates to hardware components. We will need to save and restore the CPU and several peripheral states in a non-volatile memory;
- the **logical** context relates to software execution. The program is a representation of an algorithm whose execution steps must be correctly managed. When a power failure occurs, you have a **snapshot** of your program that you have to save; after resuming, you have to restore the algorithm to the correct execution step. Of course, the non-volatile memory plays a key role to cope with data structures.

After introducing the main concepts, we are now going to present some *computing models* that developers might use to write the code targeting a batteryless system.

2.5.2 Computing models

Developers must use a **computing model** or **strategy** to deal with intermittency. Several solutions have been proposed in literature, mostly based on *checkpointing* and *task-based* models [7].

Checkpointing-based

A **checkpointing-based** model captures a system snapshot periodically and, after resume, continues with the execution starting from the state captured by the **checkpoint**. The main issue is trying to preserve the *forward progress* by increasing the size of checkpoints and volatile memory. Such increase grows the program's time and energy overhead. Moreover, there is a possibility that the code operates with inconsistent values, which makes the memory consistency uncertain [7].

Task-based

A better solution is to split the program into different **atomic sub-tasks**, saving the output data in nonvolatile memory after each execution. This is a **task-based** model [7]. Basically, the algorithm can be decomposed into functions each performing an atomic operation such as capturing, computing, saving and transmitting data. In this work of thesis, we will resort to a task-based model to deal with the batteryless datalogger's intermittency, as we will see in chapters 5 and 8.

According to researchers [7], most of task-based models do not consider the energy awareness. Actually, since the computing sub-system is part of the load to be powered, it is possible to alert it, about an upcoming power failure, by adding proper hardware components or exploiting some features of the available electronics. We will return to this topic in chapter 4. We can anticipate that, if the hardware provides some signals to the computing sub-system, developers can write a sub-task whose role is to correctly manage the power failure and perform a safe system shutdown.

2.5.3 Libraries

Nowadays, most of SoC manufacturers, such as NXP, ST Microelectronics and Texas Instruments (TI), provide **libraries** to users which can be used to write a *robust* and *portable* code. The idea is to help developers thanks to SoC architects' skill and knowledge with the goal of correctly managing CPUs and peripherals. **Robustness** of libraries is highly achieved because users can quickly signal a bug that will be solved within the next release. **Portability** is targeted by architects with the aim of applying a few modifications for running the code on a family of SoCs. We resorted to a library by TI to manage the physical context as we will see in chapter 8.

In this chapter, we have defined and described batteryless systems. We have seen how to specify, design and implement an electronic system harvesting energy from the surrounding environment through different sources. Particular emphasis has been given to main features and design challenges. The chapter should have provided a sufficient background. The rest of the document will explain how we implemented a batteryless environmental datalogger.

Chapter 3

Project overview

In the previous chapter, batteryless systems were introduced. Now we will have two main goals. First of all, we will define the *datalogging* process and thus the role of *dataloggers*. Then, we will present our project. In particular, we will describe a simplified *development process* which allowed us to face the complexity of developing an application involving hardware and software components. The idea was to resort to best practices and methods to implement a modular and scalable application for experimental purposes. A section will be devoted to each activity in the process: *requirements*, *design*, *implementation*, *testing*, *deployment* and *managing*.

3.1 Dataloggers

Nowadays, it has become essential, if not crucial, the need of capturing various types of *information* about a specific environment, often gathered remotely over an extended period of time [45]. This need is a key requirement for research, industrial and safety-critical applications.

Let us give an example. Imagine a factory having hundreds of specialized workers each of which performing a particular job into a dedicated area. Many elements may act on the environment surrounding the worker endangering his/her health and safety. We perfectly know that companies usually adopt strict protocols and rules in order to take care about their workers. This goal can be achieved thanks to helmets, gloves, masks and other useful tools, but you could do more. It may be a good idea to monitor each area through an electronic system able to *capture*, *store* and *collect* relevant *data* about the environment surrounding the worker such as temperature, humidity and air quality. *Data* may be sent to a remote control system able to analyze them and make decisions. The remote control system may act on the environment surrounding the worker by activating air conditioners or vacuum cleaners with the goal of leading temperature, humidity and air quality to nominal values. Furthermore, the electronic system monitoring an area, may also detect smoke, gas leaks or the presence of dangerous chemical agents. These events may be notified to the remote control system. The entire process should be performed *regularly* and *consistently*. We have entered the topic with a simple example, but it would be better to clarify the basic concepts and provide some definitions.

3.1.1 Basic concepts and definitions

First of all, we should clarify the difference between *data* and *information* from a computational point of view. In general, **data**[1] are raw numbers that need to be processed and somehow organized. As soon as they are processed, organized, structured or presented in a given context, it is called **information** [9]. As an example, data may be a single measure of temperature in a room, whereas the average temperature is an information we may infer thanks to a set of measures over time.

Another important clarification relates to whom or what can infer the information. In the previous chapter, we have mentioned Internet of Things (IoT) applications. In this scenario, data are *collected* locally by the thing (T) and sent to the cloud over the Internet network. The meaning of data is inferred in the cloud, that is, a set of data centers composed by servers running business applications. Such approach is known as **cloud-computing**. Recently, thanks to technological improvements, things can *collect* and elaborate data locally. Main advantages are latency reduction, latency predictability, energy and data volume reduction and, of course, privacy. Such new trend is known as **edge-computing** [50].

A key point concerns the meaning of *capturing*, *collecting* and *storing* data from a computational point of view. We may say they are three different *tasks*. The first *task* is to **capture** the physical quantity from the target environment. This goal can be achieved thanks to a sensor whose main purpose is to *sense* the physical quantity and provide an analog signal as output. The analog signal is usually conditioned to be adapted to an analog-to-digital converter (ADC) whose output will provide a digital signal on a certain number of bits. In the end, the measure is captured and becomes a data. The second *task* performs several times, *regularly* or not, the previous one. The goal is to **collect** instances of the same measure. The third *task* has the role of saving data that are captured. The goal is to **store** each data in a suitable device for later purposes.

The last basic concepts we would like to report here relate to *tasks* that are executed *regularly* and *consistently* [45]. Although they may seem trivial, these aspects are relevant and crucial for mission-critical applications. A *task* is executed **regularly**, if exists an event able to trigger it. As an example, a timer module may create an event every second or minute or a longer time. A *task* is executed **consistently**, if it performs its actions always in the same conditions. As an example, a *task* whose goal is to capture the temperature, should ensure the same resolution for each sample.

After introducing the basic concepts, we are ready to provide and comment two important definitions that will be useful through the document.

Definition 4. *The **datalogging** process captures, collects and store data over a period of time. It covers an extensive range of purposes and environments [45].*

Definition 5. *A **datalogger** is typically a small, but powerful piece of autonomous electronic measurement equipment able to perform a datalogging process regularly and consistently. Dataloggers are widely used in a diverse range of professional data sampling and analysis tasks [45].*

[1] We remark that today *data* is well-accepted and used in English both as a plural and as a singular mass noun [58].

In practice, datalogging is a process involving several *tasks* that should be carefully defined. However, one of its main roles is to *record* data, therefore the process could be also called *datarecording*. Furthermore, dataloggers are *autonomous* electronic systems and do not necessarily need to interface with a desktop computer or another remote system [45]. These choices or advanced features may have an high impact on design and implementation activities. Figure 3.1 shows two examples of dataloggers you could find in the market.

(a) Single channel [45] (b) Multi-channel [45]

Figure 3.1: Examples of dataloggers.

Most of dataloggers are battery-based electronic systems. Since the objective of this work is the *implementation of a batteryless environmental datalogger*, we will see throughout the document, how we faced the challenge as well as the consequences and limitations. Let us report in the following something more about dataloggers to have a complete picture and a basic knowledge about them.

3.1.2 Applications

Research, analysis, performance and process monitoring applications take advantage of dataloggers [45]. To have an idea of their importance, we are now briefly present some cases.

We have previously introduced a simple example. At the end of the day, it concerns an **industrial health and safety** application. Indeed, dataloggers are currently being used in a number of industries where the health and safety of workers can be at risk because of high temperatures, air quality and other factors [14]. More in general, **environmental monitoring** applications are the ones in which dataloggers are widely-used. They can record and transmit precise temperature, humidity, and other data so that companies and organizations are allowed to better monitor the areas of interest.

Another class of applications relate to **energy efficiency management**. Dataloggers monitor and manage the energy efficiency of a space or building [14]. Modern buildings are based on domotics or home automation systems with the aim of taking under control resources in an efficient manner. All of us know that saving energy means avoiding wastefulness and saving money.

Finally, an interesting class of applications is about **weather tracking, prediction and environmental hazard detection**. In this case, dataloggers are widely used in environmental research and for tracking phenomena such as weather patterns and climate change. [45].

3.1.3 Benefits

There are many benefits to using dataloggers. Here, we are briefly mentioning some of them.

Dataloggers allow measurements to be taken automatically and precisely at set intervals without the need for manpower [47]. This can be considered a **high-efficient** and **accurate** approach saving time and money. A manual approach would be error-prone, not scalable and would require a constant human observation.

Dataloggers are **versatile**. Measures can be captured indoors, outdoors, underwater and in most of critical environments, make them suitable for a variety of applications [47]. Of course, they have to be designed carefully to work in the target environment. Most of them exploit some connectivity with the goal of sending data to a remote application for data inferring and analysis.

Dataloggers are able to notify crucial events thanks to LED alarms, e-mail notifications and alarm boxes [47]. These features are basically exploited to alert users for **safety**, **security** and **cybersecurity** purposes.

Finally, dataloggers are **reliable**. Data are captured, collected and stored into non-volatile memories. They can operate for extended periods of time thanks to long-life batteries [47].

3.1.4 Specifications

A datalogger can be considered an *embedded system* hosting hardware and software components. This statement will be resumed and discussed later in section 3.4.1. It is usually based on a microprocessor or a microcontroller, some form of on-board data storage, one or more sensors and potentially a variety of other resources [45].

The complexity of a datalogger depends on the target application, but its main feature is about the **number of channels**. *Single-channel* dataloggers are usually designed to measure just one type of environmental parameter via a suitable sensor. They are often the cheapest and most straightforward. *Multi-channel* dataloggers are more complex as they can accept various input signals via different types of probes, attachments and sensors [45]. The specifications should also report whether a user can set how frequently a measure is captured at regular intervals or not. Each configuration should be carefully described.

The **power consumption** is a crucial design metric. Dataloggers are usually deployed in the target environment for a very long time. To maximize the life of batteries, among other possibilities, developers may exploit the idle times between measurements and enter a *low-power state*. This approach is possible thanks to *energy management* policies, but we will face the topic in chapter 8.

Just to have an idea, look at table 3.1 referring to 1384 datalogger by RS PRO we have reported in figure 3.1b. It is a 4-channel temperature datalogger working with different sensor types. This model can be easily handled by a human; it means that it was developed targeting important metrics like **size** and **weight**. It is a battery-powered electronic system and the battery-life is roughly two days. This model is not suitable for stand-alone applications due to its nature and purpose. Its main goal is to measure the temperature, thus the specifications report the maximum temperature measurement, accuracy and resolution. It records up to 512 kB of data that a user can view thanks to a suitable LCD

Attribute	**Value**
Measurement Parameters	Temperature
Number of Input Channels	4
Sensor Type	B, C, E, J, K, L, N, R, S, T, U-type thermocouple
Interface Type	USB
Data Logger Type	Temperature
Calibrated	UKAS
Maximum Temperature Measurement	+1112 (U) °F, +1300 (N) °C, +1370 (K) °C, +1400 (E) °F, +1600 (R/S) °C, +1652 (L) °F, +1760 (B) °C, +1760 (C) °C, +1832 (J) °F, +2372 (N) °F, +2498 (K) °F, +2912 (R/S) °F, +3200 (B) °F, +3200 (C) °F, +400 (T) °C, +600 (U) °C, +752 (T) °F, +760 (E) °C, +900 (L) °C, +999.9 (J) °C
Power Source	Battery
Best Temperature Measurement Accuracy	±0.05 (B) %, ±0.05 (E) %, ±0.05 (J) %, ±0.05 (K) %, ±0.05 (R/S) %, ±0.1 (C) %, ±0.1 (L) %, ±0.1 (N) %, ±0.1 (T) %, ±0.1 (U) %
Minimum Operating Temperature	0°C
Maximum Operating Temperature	+50°C
Display Type	LCD
Includes	Battery, Carrying Case, Instruction Manual, Software CD, Type K Thermocouples, USB Cable
Battery Life	2 Days
Temperature Measurement Resolution	0.1 °F, 0.1°C
Battery Type	AA
Alarm	Yes
Model Number p	1384, 1361C
Number of Readings Per Channel	100000 (1 Input), 36000 (4 Input), 99 (Set)
Auto data logging capacity	512 KB
Dimensions	187 mm Length x 73 mm Width x 53 mm Height
Weight	405 g approximately (with batteries)
Cost	£332.65

Table 3.1: Specifications of model 1384 by RS PRO [45].

display. The electronic system can be connected to a Personal Computer by means of a USB interface. A relevant specification relates to the number of readings per channel.

We reported this example to point out that developing a datalogger is a challenge that requires a considerable effort. The effort justifies the **cost** and in particular the *unit cost* and the *non-recurring engineering cost (NRE)*. The unit cost is the *bill of material (BOM)* you need to manufacture a copy of each system. NRE involves all you need to develop the system such as tool licenses or developers' salary. Companies need to enter the market

on time due to **time-to-market** constraints. The technology thanks to which you enter the market should be the best one to avoid losses as much as possible. The cost you can read in the last row of table 3.1 was probably calculated by meeting the *metrics* we have mentioned and based on a market analysis in order to get a fair profit.

3.1.5 Types

We are going to conclude this first part saying that there are many types of dataloggers out there. They can capture almost any measurable characteristic of an environment. Among them, we can mention time, water level, light intensity, wind speed and direction, room occupancy, differential pressure, rainfall and many others. However, after time, **temperature** and **humidity** are the most measured characteristics of an environment [45].

Starting from the next section, we will present the project we developed for this thesis work. The project targeted a *batteryless environmental datalogger* capable of measuring temperature and humidity.

3.2 Development process

After introducing the basic concepts, the datalogging process and dataloggers in general, we can start presenting our project.

Developing a system from scratch is a nightmare if you do not have a strategy. This statement is true both for a product to sell and for experimental purposes. That's why developers usually resort to a **development process (DP)**. The target system may be a software component or a system involving hardware and software components. A DP takes into account several **activities** or **disciplines** which are performed in different **phases**. Best DPs are *iterative* and *incremental*. In this scenario, the development of a system is divided into a number of **iterations** or **cycles**. Developers perform one or more activities in each iteration with a different rate; each iteration results in an *increment*, which is a **release** of the system that contains added or improved functionalities compared with the previous one [57]. Crucial decisions are: the definition of phases, types of activities and number of iterations to be performed in each phase. The worst DPs are *waterfall*. Activities usually coincide with phases and a phase starts when the previous ends. Studies demonstrated that waterfall processes, eventually, lead to the project failure. Companies producing electronic systems, software products or other *artifacts* are aware of this risk, especially when dealing with time-to-market and time-to-prototype constraints.

Developers resorting to a selected DP have many benefits. First of all, there are a large number of process **models** out there among which you can choose, each suitable for a specific purpose. A DP, whose objective is the development of an electronic system, will certainly be different from another one targeting a software component. Another benefit concerns the **best practices**. Since processes are mature and well-documented, you usually do not have to *reinvent the wheel*. Developers reuse the knowledge which has been acquired by their predecessors. Finally, resorting to a DP means having a **strategy**. Developers must think before they do in each step in the process. The development is driven by choices and appropriate considerations in order to prevent flaws, randomness,

confusion and errors. Obviously, there are many other benefits we do not want to list here. The final message is that, thanks to a suitable DP, engineers can achieve high-quality results in less time and monitor the development of the system they are working on. More in general, companies resort to DPs to apply best practices, methods and *design patterns* with the aim of producing **dependable** systems: systems performing their tasks with no failures. As we will see later on, *testing* is a key activity to achieve dependability.

To develop our application we resorted to a **simplified custom development process** as you can see in figure 3.2. It is an iterative and incremental DP whose target is the development of an *application* made up of hardware and software components. The process defines a unique *phase* called **development**. The phase is performed in a certain number of *cycles*. A cycle is a period of time of variable duration. This choice allowed us to perform *activities* without being stressed by rigid deadlines. In every cycle, all the *activities* are performed with a different rate; they are: **requirements**, **design**, **implementation**, **testing**, **deployment** and **managing**.

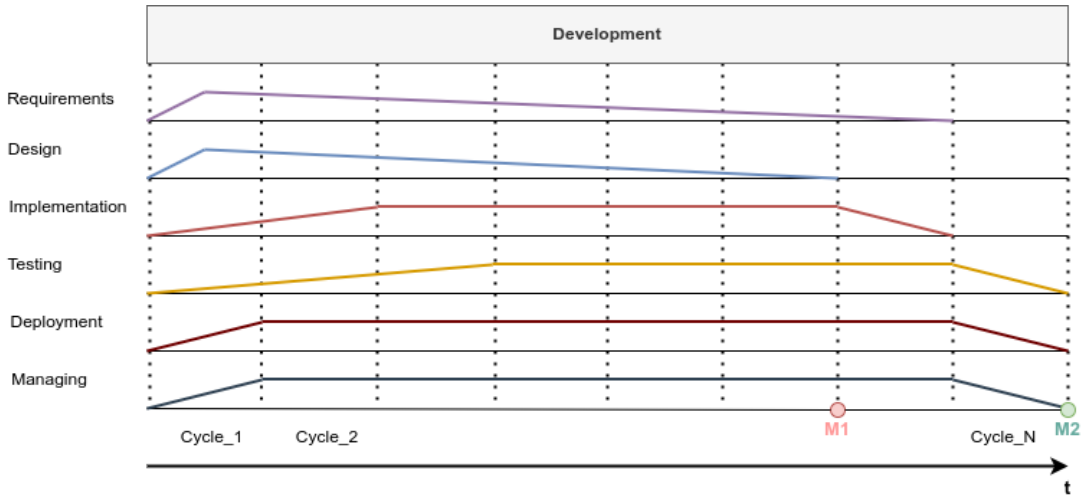


Figure 3.2: Custom development process.

Basically, figure 3.2 shows that activities were performed with a different rate in each cycle during the process. *Requirements* activity was the most relevant in the first cycles as well as *design*. In the middle of the process, *implementation* activity prevailed over the others. *Testing* activity was the most performed in last cycles. *Managing* and *deployment* activities were practically always performed. The reason is that the project management was based on configurations and updates, whereas the nature of the target application required to interact with the target environment continuously.

Finally, the figure also shows M1 and M2 points which are *milestones*. A **milestone** can be defined in many ways, but in our custom development process it represented an *expected outcome*. This topic will be resumed in section 3.7 whose main objective is to describe the *deployment* activity. All the other activities will be discussed in the rest of the chapter.

3.3 Requirements

The goal of *requirements* activity is to describe, in a formal way, the target system or application to develop. In other words, it documents what the target system should do. Its outcome is the **requirements document**. It is the result of a set of techniques guiding developers to the *right product*. We could state that **specifications** are an interpretation of the requirements document. The better the requirements are described, the less is the possibility of flaws, misunderstandings and interpretation errors.

For the sake of convenience, the reader may open and read the contents of our requirements document: the Requirements.md file. Instructions can be found in section 3.8. The document reports four **techniques** we selected among many available in literature: the *abstract*, *story*, *interfaces*, *functional and non-functional requirements*. We are going to present these techniques now.

3.3.1 Abstract

Thanks to an **abstract**, we started describing the target application without thinking of a real implementation. The benefit is about details. It is similar to an abstract you could find in a book helping the reader to enter the topic with the aim of focusing on main aspects. Here is our abstract taken from the requirements document:

> A *batteryless environmental datalogger* is an electronic system able to capture, collect and store temperature and humidity measures of a room. Each measure is stored in its internal memory. The electronic system behaves like an intermittent indoor datalogger. Main feature is the absence of any battery. It is totally powered thanks to energy harvested by means of a small photovoltaic cell. Any power failure is successfully managed in order to save the current state and perform a correct shutdown. When new energy is available, the system resumes, restores the saved state and starts measuring.
>
> Captured, collected and stored data by datalogger are sent to a remote system, called *gateway*, through a wireless interface. After receiving a measure, the gateway returns the timestamp to the datalogger which will be added to the measure. Furthermore, the gateway will send the measure to the *server*.
>
> The server hosts a REST Web service, a simple, but effective API. The service exposes suitable endpoints and documents how resources can be consumed. Data are saved into a database.
>
> The entire application should be scalable. The Web service should be able to deal with more than one gateway each of which should be able to manage more than one datalogger. Plug and Play solutions are preferred.
>
> Users can access data collected by the application thanks to a Browser.
>
> The entire application has experimental purposes. The main goal is to test and evaluate different computing strategies able to manage the intermittent behavior of the datalogger. Gateway and server should be able to perform their tasks without taking into account other aspects such as confidentiality, integrity and availability.

As you can see, the abstract is a short, meaningful and self-explaining text. You can read it and infer a *high level description* of the target system to develop. Here is the first point. In our case the target is an **application** involving hardware and software components. The three main *actors* are the **datalogger**, the **gateway** and the **server**. In particular, thanks to the abstract, we can state that:

- a datalogger is a **batteryless electronic system** hosting hardware and software components, working **indoor** and powered by means of a **small photovoltaic cell**. It must send data to a remote system, called *gateway*, through a **wireless interface**;

- a gateway is a sort of **data concentrator** able to receive data from different dataloggers. The gateway must send data to a *server*. Since the abstract mentions a REST Web service, the gateway should have a software component able to work as a **REST client**;

- the server actually runs a **REST Web service** that must deal with different gateways. Data must be stored into a **database**. A user is allowed to display data thanks to a **Browser**.

Furthermore, the abstract points out the application's main goal. It has **experimental purposes** and focuses on testing and evaluating different **computing strategies** capable to managing the **intermittent behavior** of the datalogger.

After reading the abstract, developers might estimate the complexity of the target application to develop, at least from a high level point of view. To go forward, they need the next technique.

3.3.2 Story

To help understanding the interactions among actors, developers might try to tell a **story**. Stories are mostly used in software engineering to illustrate design solutions from user's point of view and tell user's goals, motivations and actions. This technique has a crucial role as we will discuss later on.

We adapted this technique to requirements as it helped us to better describe, with more details, the behavior of the entire application. You do not need to tell a long story; what matters is the interaction among actors. Of course, its length depends on the complexity of the target application. For high-complexity applications it might be necessary to tell more than one story.

The key point is that:

- actors are **objects** interfacing each other;

- verbs represent **actions** or **tasks**;

- interaction among actors will be better clarified through **interfaces**;

- actors and verbs will be crucial for writing down the **functional requirements**. Other details will be crucial for **non-functional requirements**.

In general, comments are not necessary if the story is complete and told with the desired level of detail. Here is ours:

> A datalogger works intermittently due to its nature. It resumes when the energy harvested thanks to the photovoltaic panel allows the system to be powered. After resuming, it performs an initialization in order to restore the context that was saved before the last shutdown. Then, it captures a temperature and humidity measure. This measure is sent to a known gateway thanks to a long-range wireless interface in the form:
>
> datalogger_id temperature humidity
>
> The gateway receives the measure and returns a timestamp to be added to the measure. The measure is sent back to the datalogger in the form:
>
> datalogger_id timestamp
>
> The datalogger stores the measure in its internal memory. It collects measures regularly and consistently. The datalogger must be notified about a power failure; in this case it runs a shutdown procedure in order to save its context correctly. Since a datalogger works indoor, a suitable photovoltaic panel is selected among those available in the market. Low-power consumption is a key metric. The range of temperature and humidity measures is compatible with indoor ones. A datalogger is easily manageable in order to allow developers to conduct experiments.
>
> A gateway has two main purposes. First of all, it works as a data concentrator thanks to its long-range wireless interface. It can receive measures from any datalogger with a relationship many to one. The second purpose is to send each received measure to a REST Web service. This means it acts as a REST client capable of performing POST HTTP requests to the server. Each gateway can be recognized thanks to either IPv4 and IPv6 addresses. IPv4 address is mandatory.
>
> A server hosts a REST Web service exposing suitable endpoints. It is able to deal with more than one gateway with a relationship many to one. The Web service saves each measure into a database. The database contains tables storing data about gateways, dataloggers and measures. In particular, the server knows each gateway, dataloggers managed by each gateway and measures performed by each datalogger managed by a gateway. Data can be retrieved thanks to the REST spirit and thanks to a Browser. The Browser allows users to display and analyze data with the goal of maximizing the user experience.

3.3.3 Interfaces

Logical and physical interactions among actors can be easily described thanks to **interfaces** technique. The information can be inferred by reading the abstract and the story. Our interfaces are shown in table 3.2.

This technique looks like useless, but it is not. Developers can easily identify the actors involved in the application as well as their interfaces. A high-complexity application may have one or more tables with hundreds of rows. This technique is a smart and efficient way of documenting data that would be difficult to retrieve after a long time.

Actor	Logic Interface	Physical Interface
Datalogger	Data sender	Wireless module
Gateway	Data concentrator, REST Client	Wireless module, Internet link
Server	REST API	Internet link
Browser	Graphical User Interface (GUI)	PC, Smartphone

Table 3.2: Interfaces of the application.

Furthermore there is more. Writing *wireless module* as item, simply means that you are going to use a wireless module whose type will be decided later, maybe during the *design* activity. At this point developers are writing down the requirements, but some details have not been defined yet or you are not sure about the final choice. Details may be added later on after receiving the information. This is a classic example of iteration. The document can be updated in any cycle because the process is iterative and incremental.

3.3.4 Functional and non-functional requirements

By reading the story, developers can infer or better define the **tasks** performed by each actor as well as other relevant and important information. This technique allowed us to write down the *functional* and *non-functional requirements* which are reported in tables 3.3 and 3.4.

A **functional requirement (FR)** defines a *function* or *task* that the target application must perform. As you can see, an actor manages *tasks* which are described by some verbs in the story. A first level FR defines the actor. It represents a software or hardware component performing the required tasks. Each FR has an identifier (ID). The ID will be crucial during the *implementation* activity.

A **non-functional requirement (NFR)** is an aspect of the application which is not related to its functions. It is a relevant aspect that the application must satisfy. It could be a metric, that is, a measurable feature of a system implementation. It is a constraint that the application must meet. An example of metric may be: *the power consumption must be less than or equal to* 100 mW. Developers must guarantee that the system will be able to meet any metric described in the requirements document.

Here is an important point. If you are developing an experimental application, you may not have some metrics. Metrics depend on developers' choices and goals. Obviously, you would develop a low-power, low-size and high-performance target system. Furthermore, low time-to-prototype, low unit and non-recurring engineering cost would be preferable. The problem is that you usually have *trade-offs* among metrics and not always *optimizations*. A **trade-off** means that if you are improving a metric, there is at least another metric which is worsening. An **optimization** is another story; if you are improving a metric, the other metrics may improve or remain stable. Developers try to optimize, but trade-offs are more common. We will resume this topic in chapter 8.

We can conclude this section with an important consideration. There are many other techniques we could have used to improve and enrich our requirements document. As an example, we could have applied the *use case diagram* and *use cases*, the *glossary*, the *system diagram* and *deployment diagram*, just to mention a few. The point is that we never

ID	Description
FR1	Handle Datalogger
FR1.1	Restore the context
FR1.2	Capture a measure
FR1.3	Store a measure in memory
FR1.4	Collect measures
FR1.5	Send a measure to a gateway
FR1.6	Save the context
FR1.7	Manage workload
FR1.8	Monitor power failures
FR1.9	Manage shutdown
FR2	Handle Gateway
FR2.1	Receive a measure from a datalogger
FR2.2	Send a measure to the Web service
FR3	Handle Web service
FR3.1	Add a gateway
FR3.2	Retrieve a gateway
FR3.3	Retrieve all gateways
FR3.4	Add a datalogger to a gateway
FR3.5	Retrieve a datalogger of a gateway
FR3.6	Retrieve all dataloggers of a gateway
FR3.7	Add a measure of a datalogger
FR4	Handle Browser
FR4.1	Display measures

Table 3.3: Functional requirements.

ID	Description
NFR1	A datalogger is a batteryless system
NFR2	A datalogger harvests energy thanks to a photovoltaic cell
NFR3	A datalogger works indoor
NFR4	A datalogger performs measures regularly and consistently
NFR5	A datalogger reduces as much as possible the power consumption
NFR6	A gateway is mostly available
NFR7	The Web service does not take into account availability
NFR8	The Web service does not take into account confidentiality
NFR9	The Web service does not take into account integrity
NFR10	The entire system works with a Plug and Play strategy
NRF11	Decimal numbers use .(dot) as decimal separator
NRF12	Unit of measure for temperature is Celsius. Range is at least [-20, +50]
NRF13	The percentage of relative humidity is required

Table 3.4: Non functional requirements.

forgot our goal: creating the requirements document for an experimental application and not for a product to sell. In practice, we were satisfied about the level of details achieved thanks to the selected techniques.

3.4 Design

The *design* activity gets the requirements document as input and produces a **model** as output. It models the target application as architects would do. Architects usually use scale models to design buildings, bridges and other complex structures; electronic and computer engineers usually use diagrams, software and hardware languages to model their applications.

By reading our requirements document, in particular the story and interfaces, we drew the global scheme of the target application shown in figure 3.3. Here actors became **systems** interacting each other. The model reports more information that will be discussed soon: the *design choices*.

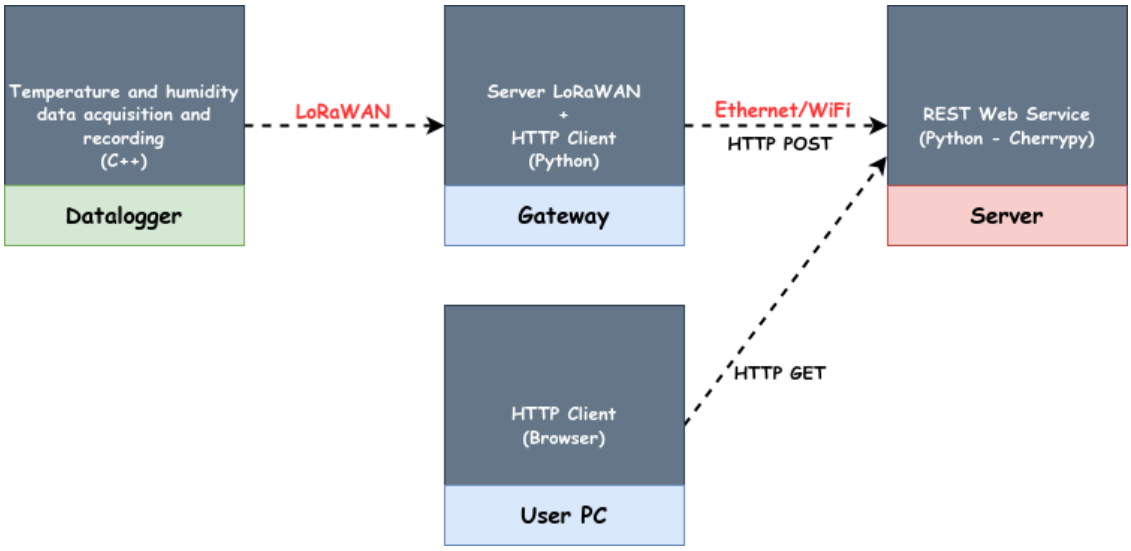


Figure 3.3: Global scheme of the target application.

After keeping in mind the application scalability, the scheme became the one shown in figure 3.4.

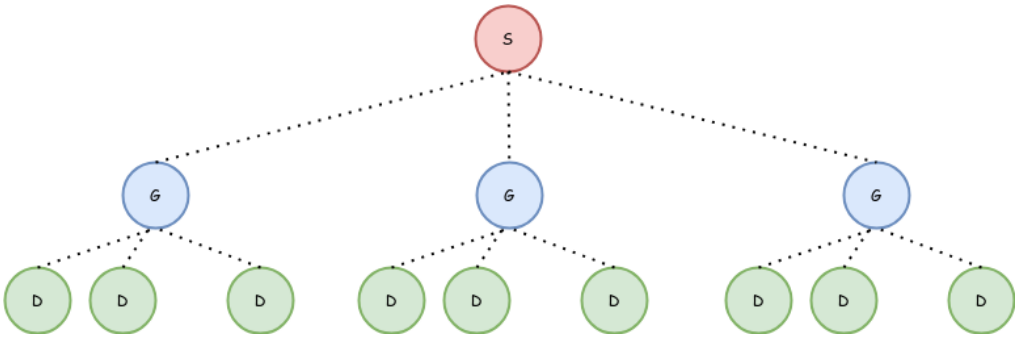


Figure 3.4: Scalable global scheme of the target application.

Here, a server hosts a REST Web service dealing with one ore more gateways each of which deals with one ore more dataloggers. Each datalogger is a batteryless autonomous intermittent electronic system exchanging data with a remote gateway. The datalogger knows the identifier (ID) of the remote gateway; the gateway knows the ID of each associated datalogger. The Web service knows the Internet Protocol (IP) address of each gateway. In a more complex scenario a *server farm* would be used to distribute the workload.

In the following, we will refer to the global scheme shown in figure 3.3 in order to focus on *design choices*. The discussion will not affect the scalable solution.

3.4.1 Datalogger

In the first part of this chapter, we have defined *datalogging* as the process of capturing, collecting and storing data over a period of time. Furthermore, we have defined a *datalogger* as an autonomous measurement equipment performing a datalogging process. Since the purpose of such a system is not general, but *specific*, we can say a datalogger is an *embedded system*.

An **embedded system (ES)** is a *special-purpose computing system* able to control a complex object or a target environment via sensors and actuators often with real-time constraints. It is an electronic system hosting hardware components, software components and communication interfaces. Sensors and actuators type depend on the application. They are usually connected to ES via connectors, although in some cases, they could be placed into the printed circuit board (PCB) thanks to suitable *footprints*. Real-time constraints are met by writing smart and efficient pieces of code whose purpose is to drive specific peripherals. Furthermore, there are many real-time operating systems (RTOS) out there which have been designed for embedded systems. Their main goal is to deal with *deadlines* which represent the main differences between general-purpose and special-purpose computing systems. The former must be *interactive* with users, the latter must be *reactive* to external stimuli.

As we have already discussed, developers create a model of the target application starting from the requirements document. The result of our modeling regarding a datalogger is shown in figure 3.5. We modeled a datalogger as an embedded system working with two different views: the *functional view* and *non-functional view*. The **functional view** describes what the system does in terms of functionality; connections represent *flow of data*. The functional view can also be decomposed in three main sub-systems: the *sensing sub-system*, *computing sub-system* and *communication sub-system*. The **non-functional view** describes how the system is powered; connections represent the *flow of energy/power* which involves further components. The key point is that functional components are generic *loads* from the power flow perspective, as we saw in section 2.3.

Functional view

The **sensing sub-system** captures temperature and humidity measures. We focused on suitable **low-power sensors** able to capture measures which were compatible with the mission. In particular, the sensing sub-system had to meet NFR12 and NFR13 non-functional requirements.

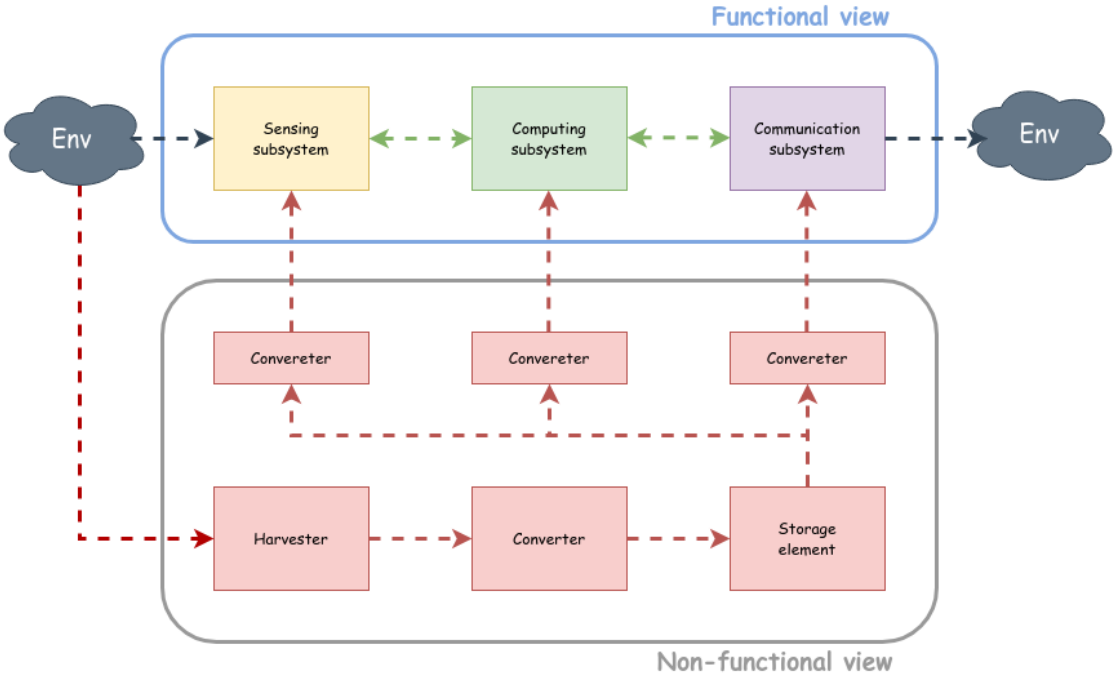


Figure 3.5: Model of the datalogger.

The **computing sub-system** manages the entire ES. It had to meet FR1 class functional requirements and NFR4, NFR5 non-functional requirements. Our choice was to base the computing sub-system on a suitable **low-power MCU** to reduce the power consumption as much as possible. This choice played a key role because, as you know, MCUs are complete computing systems with many resources and interfaces. Furthermore, they can enter low-power state modes which are exploitable for different purposes. We would like to point out that MCUs are fine, but have some limitations such as, for instance, the maximum operating frequency. As a consequence, they may not be able to run complex algorithms or high-data volume based applications [3]. The *firmware* of most MCUs can be written in **C/C++** which are *medium-level* and versatile languages you might use to write operating systems, software applications, drivers and libraries. The firmware written in C/C++ masks the need of using the *assembly* language and its low-level details [3].

The **communication sub-system** targets a **low-power/long-range** transmission protocol. Our choice was to use a LoRaWAN® transceiver. LoRaWAN® is a medium access control (MAC) layer protocol built on top of LoRa®. In other words, LoRa® represents the physical layer of a LoRaWAN® network [32]. The topic is too complex to face here. We will present LoRa Alliance® and LoRaWAN® in chapter 7. We can anticipate that main advantages of adopting this technology are low power, long range, high security, no need for Internet, easy to deploy and low cost. Of course, there are disadvantages like low bandwidth, low transmission rate and spectrum interference; the consequence is that the technology is not exploitable by some applications.

Non-functional view

The **energy harvester sub-system** had to meet NFR2 and NFR3 non-functional requirements. A key point is about the strict relationship between them. Indeed, there are photovoltaic panels which are designed to work *indoor* or *outdoor*. Our goal was to select a suitable **indoor solar cell**.

The **converter sub-systems** convert the energy/power for different purposes. First of all, we had to store the energy harvested by the photovoltaic cell somewhere; as a consequence we needed to convert energy to be stored into an *energy storage device*. Furthermore, the available energy had to be converted to power all the functional sub-systems requiring different voltages and currents.

The **energy storage device (ESD)** had to meet NFR1 non-functional requirement. Since we were dealing with a batteryless system, our goal was to select a suitable **supercapacitor**. All these topics will be resumed in chapter 4.

3.4.2 Server

A *server* is any powerful-enough general-purpose computing system capable of running pieces of code with the aim of creating *distributed applications*. This statement is simplified and would require an in-depth discussion.

Our project required a **REST Web Service** or **REST API** able to meet FR3 class functional requirements and NFR7, NFR8, NFR9 non-functional requirements. A REST Web service exposes *endpoints*. An **endpoint** is a public Unified Resource Identifier (URI) that client applications use to access a *resource* of a Web service [38]. A **resource** is any data available in the Web service that can be accessed and manipulated with HTTP requests to the REST API. Our choice was to create the set of endpoints reported in table 3.5.

API endpoint	HTTP	Description
/gateways	GET	Retrieve all gateways
/gateways	POST	Create a new gateway
/gateways/{id}	GET	Retrieve a single gateway
/gateways/{id}/dataloggers	GET	Retrieve all dataloggers
/gateways/{id}/dataloggers	POST	Create a new datalogger
/gateways/{id}/dataloggers/{id}	GET	Retrieve a single datalogger
/gateways/{id}/dataloggers/{id}/measures	GET	Retrieve all measures
/gateways/{id}/dataloggers/{id}/measures	POST	Create a new measure
/gateways/{id}/dataloggers/{id}/measures/{id}	GET	Retrieve a single measure

Table 3.5: Endpoints of the REST Web service.

Another crucial point was about the technology. We decided to use the **Python** programming language. More in detail, we selected a technology known as *CherryPy*. **CherryPy** is a *pythonic*, object-oriented web framework allowing developers to build web applications in much the same way they would build any other object-oriented Python program. In practice, developers are able to develop smaller source code in less time [4]. According to creators,

> "*CherryPy is now more than ten years old, and it has proven to be fast and reliable. It is being used in production by many sites, from the simplest to the most demanding* "[4].

The other point was about how to store data. We decided to use **SQLite**, a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine [44].

It is important to point out that a REST Web service is any service that adheres to REST architecture constraints. Our service needed an IP *address* and a *port* in order to be reachable. For sure the topic is extended and requires more details. We will resume the topic in chapter 6.

3.4.3 Gateway

Our project required a *gateway* able to receive data from the datalogger to be forwarded to the Web service. In particular, a gateway had to meet FR2 class functional requirements and NFR6 non-functional requirements.

From the specifications, we inferred a gateway was the combination of a hardware component and a software component. The hardware component had the role of managing the LoRaWAN® protocol; incoming messages from the datalogger were received and passed to the software component. The software component had the role of creating a *timestamp* to be added to the incoming measure to send back to datalogger. It should also work as a **REST client** to send the measure to the Web service. Our choice was to use the **Python** language with the aim of creating a software component able to interface with the hardware component and send data to the Web service thanks to HTTP requests. We will resume this topic in chapter 7.

3.4.4 Browser

Our project required a *Browser* able to meet FR4 class functional requirements. A **Browser** is a computer program with a graphical user interface for displaying and navigating Web pages. We obviously did not need to design a Browser from scratch, but simply use one out of many available to access the Web service thanks to HTTP requests. Also this topic will be resumed in chapter 7.

3.4.5 Data format

According to specifications, we should be able to exchange data between the datalogger and the gateway. Furthermore, the gateway had to exchange data with the Web service. We also had to consider the NFR11 non-functional requirement.

The datalogger is an embedded system working autonomously. An ID was required in order to be recognized. It had to send each measure to the gateway with the following format:

datalogger_id temperature humidity

The gateway provided a *timestamp* to be added to the measure which was sent back to datalogger in the form:

datalogger_id timestamp

We decided to send data from the datalogger to the gateway in a **raw** format. This topic will be resumed in chapter 5.

The gateway had two interfaces. The one needed for managing the LoRaWAN® protocol and the one for performing HTTP requests. This means it needed an ID and an IP *address*. We decided to use the **JSON** format to exchange data between the gateway and the Web service. This topic will be discussed in chapter 6.

3.5 Implementation

In the previous section, we have introduced the design activity. The reader should have understood the role of each system, the interconnections and involved communication protocols. At this point our design choices should be clear.

The *implementation* activity gets the model provided by the design activity as input and produce an *artifact* as output. An **artifact** is an object made by a human being that, in our case, was a software or hardware component. Since the implementation is the main topic of this work, we would like to sum up the contents of chapters regarding this activity.

Chapter 4, *Datalogger: hardware implementation*, will explain how we implemented the hardware of the datalogger. Each module implementing a particular sub-system will be presented as well as its main features and interfaces. By the end of the chapter, the reader will be instructed to properly handle the datalogger to reduce the risk of damages. These recommendations are crucial therefore a section will be devoted to this topic.

Chapter 5, *Datalogger: firmware implementation*, will be strictly related to the previous one. It will explain how we implemented the firmware running on the target microcontroller. Essentially, our solution is based on a custom scheduler whose role is to manage relevant tasks. A task is the implementation of a functional requirement.

Chapter 6, *Server*, will explain how we implemented a simple and reliable REST Web service capable of managing and storing data provided by datalogger and gateway. The design and implementation of a Web service should be faced before creating any client. This is required, because developers must decide how the service can be consumed by a client.

Chapter 7, *Gateway*, will cover several topics. After introducing the LoRaWAN® protocol and the hardware module we selected from the market, it will explain how we implemented a software component to deal with incoming messages from the datalogger and perform HTTP requests, as a client, to the Web service. The last section will present how to reach the service through a Browser.

Chapter 8, *Datalogger: computing strategies and optimizations*, will be the most relevant one. The goal of the chapter will be to explain how we managed the context and tried different computing strategies to maximize and control the intermittent behavior of the datalogger. It will also explain how we performed some optimizations aiming at reducing the power consumption as much as possible.

Furthermore, we did our best effort to organize the material carefully and explain the implementation activity step by step. The reader may perceive or experience a user/program manual reading style.

3.6 Testing

Testing is the process aiming at identifying *faulty products*: products that do not work according to specifications. A product working as expected, with no faults, is said to be *dependable*. Actually, testing involves many operations among which we could mention *test logic verification*, *memory testing* and *memory retention*. Companies usually spend a lot of money to write **test programs** performing a set of operations. These operations are executed in a particular sequence to reach the dependability of the target system.

Since ours was an experimental project, we focused on a very subset of all possible tests we could have performed to evaluate the dependability of an application involving hardware and software components. Our goal was to:

- ensure the correctness of measured data by sensors of the datalogger (chapter 5);
- ensure that the REST Web service was working properly (chapter 6);
- ensure that the gateway was working as a REST client (chapter 7).

It might be considered useless or unreasonable performing such tests for experimental purposes. However, we wanted to ensure that the basic functionalities were correctly executed by our application.

3.7 Deployment

The *deployment* activity is the action of bringing resources into effective action. To be practical, we imagined two different scenarios.

3.7.1 Development scenario

When you develop a new application, you should focus on relevant aspects without being stressed because of details and configurations. To achieve this goal, we performed the following set of steps:

- we developed a datalogger. The main point here was the possibility of using the LoRaWAN® *module* in TEST mode. This topic will be discussed in chapter 7. We initially developed the datalogger as it were not batteryless; indeed, we powered it via USB cable. This topic bill be discussed in section 4.8.1;
- we developed a gateway, as the composition of a hardware and a software component. The hardware component is a LoRaWAN® *module* able to receive data from the datalogger. Also in this case we were working in TEST mode and we connected the *module* to a PC via USB cable. On the same PC, we were running a Python application able to interface with the *module* and forward incoming data to the Web service;
- we developed a REST Web service, a Python application running on the same PC used for the gateway. The **loopback** interface simplified the interaction between the gateway and Web service software components over the Internet network.

The deployment was very quick. First of all, we had to run the Web service waiting for gateway connections on 127.0.0.1:8080. The 8080 port was the default one and never changed. Then, we should run the gateway to start receiving data from the datalogger. If the datalogger was powered, it could send measures to the gateway.

3.7.2 Working application scenario

Our development process defined M1 and M2 milestones, as we have seen in section 3.2, which represented the **expected outcomes**.

M1 milestone

The objective of M1 milestone was to have a **working solution** capable of closing the chain datalogger-gateway-server. In particular:

- the datalogger had to be batteryless and had to work intermittently and autonomously. Data had to be sent to the gateway by using the TEST mode;
- the gateway's LoRaWAN® *module* had to be connected to a target PC via USB interface. A Python program had to be able to control the LoRaWAN® *module* and work as **REST Client** performing HTTP requests to the Web Service;
- a Python **REST Web service** had to be reachable on the same PC;
- each system had to be **easily configurable**.

M2 milestone

The objective of M2 milestone was to have an **optimized application**. In particular:

- the power consumption of the datalogger had to be optimized to maximize its intermittent working sessions;
- a user could display the available data by running a Browser.

3.8 Managing

Managing is a complex activity involving a set of actions. In our development process we simply addressed several issues of a project management.

Basically, we were interested in **version control**. The goal could be *easily* achieved thanks to Git and GitHub. A private *repository* was created to store and update the project. Version control systems are also fine for iterations in the development process as they help developers meeting deadlines and defining milestones.

To conclude this chapter, we would like to present the contents of the project that the reader may easily inspect after cloning the repository (§A.1.5). We created four different folders into the Project root folder:

- Docs: contains relevant material about the project such as the requirements document;

- Datalogger: contains several sub-folders. The Node folder contains the firmware implementation which is based on a custom-scheduler. We will refer to this folder in chapters 5 and 8. The reader will also be able to retrieve datasheets and schematics. In particular, we will refer to Schematics and Board folders in chapter 4;
- Server: contains the Python implementation of the REST Web service. We will refer to this folder in chapter 6;
- Gateway: contains the Python implementation of the gateway. We will refer to this folder in chapter 7.

In this chapter we have defined the datalogging process and the role of dataloggers. Then, we have presented the project. In particular, we have described the simplified development process we resorted to. We have seen that the idea was to apply best practices and methods with the aim of implementing a modular and scalable application for experimental purposes. From the next chapter, we will start explaining how we implemented our application.

Chapter 4

Datalogger: hardware implementation

In this chapter, we will explain how we implemented the hardware of the datalogger. Each *module* implementing a particular sub-system will be presented as well as its main features and interfaces. By the end of the chapter, the reader will be instructed to properly handle the datalogger to reduce the risk of damages. These recommendations are crucial therefore a section will be devoted to this topic.

4.1 The make-or-buy decision

Let us start by the model of the datalogger we have discussed in the previous chapter. For the sake of simplicity we report it here again in figure 4.1.

Here is the main point. Imagine to implement this embedded system for experimental purposes from scratch. First of all, you should select the components and create one or more schematics describing your system. Then, you should create a printed circuit board (PCB) or an experimental board to host and solder each electronic component or connector. The main problem is that most of components and modern integrated circuits (IC) are surface mounted devices (SMD) and their interconnections might be complex; you need proper tools and skills to deal with them. Furthermore, designing a PCB is not a trivial job. You have to start with schematics, design the board by selecting the right *footprint* for each component and *route* wires; you usually need more than one *layer* to perform the job. You should provide the manufacturer the Gerber files of your project and wait for the PCB to be manufactured. The final cost depends on the quality and whether you decide to exploit a service able to pick, place and solder each component for you. Among other problems, we can mention those related to transceivers. These devices are antenna-based. Many of them use *patch* antennas that must be carefully designed; circuits with antennas are challenging, even for expert developers. A similar problem takes place for microcontroller units. By-pass capacitors must be carefully placed close to power pins and developers have to comply with datasheets suggestions and recommendations strictly. What we are taking into account here is just a subset of all possible issues you may experience.

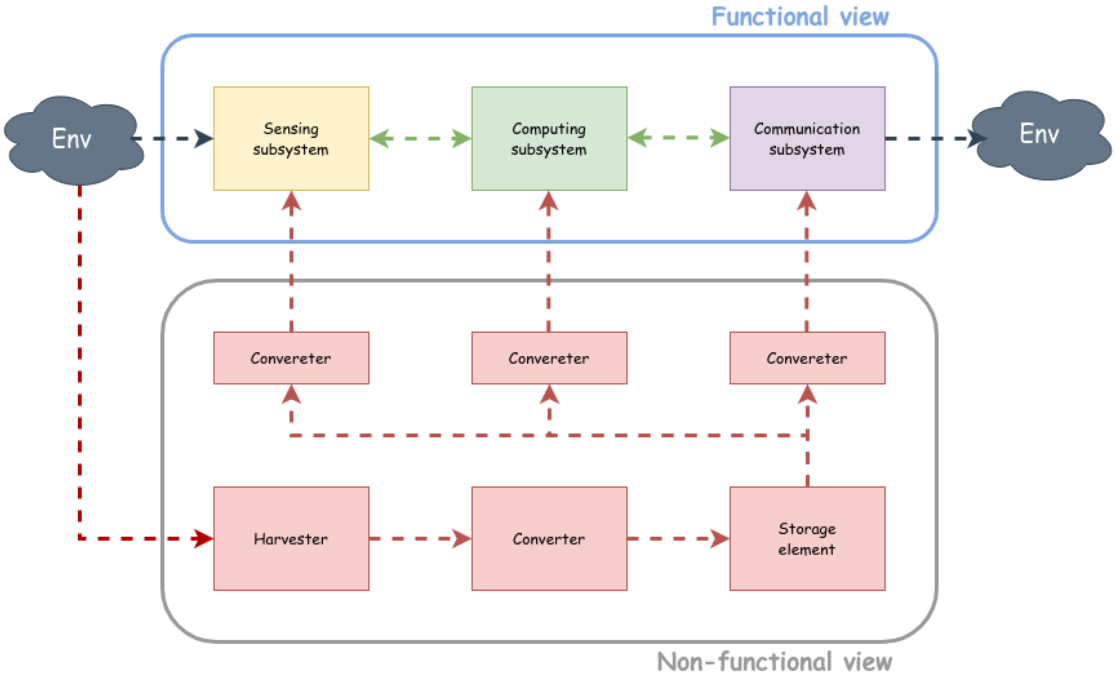


Figure 4.1: Model of the datalogger.

At the beginning, we tried to understand if it were better to start implementing everything from scratch or exploring the market to find ready-to-use solutions. The question is under the roof of a well-known matter, that is, *make-or-buy* decisions. A **make-or-buy** decision is an act of choosing between manufacturing a product in-house or purchasing it from an external supplier [25].

Our decision was to buy. In particular, we decided to buy four low-power consumption **modules** to implement each sub-system of the datalogger. The result of this choice is shown in figure 4.2. Best practices here are *modularity* and *scalability*. **Modularity** is the degree to which a system's component may be separated and recombined, often with the benefit of flexibility and variety in use. The concept of modularity is used primarily to reduce complexity by breaking a system into varying degrees of interdependence and independence sub-systems [54]. **Scalability** is the property of a system to handle a growing amount of work. In our case, as an example, we could have added more than one sensor module. The number of modules is not relevant; what matters is their interface.

Obviously, *buy* solutions imply several drawbacks. First of all, buying means performing careful choices, but the market is large and offers thousands of possibilities. Developers should perfectly know their targets to reduce the field of choice and find the best options for their purposes. Another aspect regards the design metrics. Modules you can find in the market are not optimized for any application. Developers have to take into account many trade-offs among metrics, such as area, performance and power consumption. Another relevant drawback is about cost. Each module may cost more than expected and the choice may seriously affect the unit cost of the embedded system. There are many other aspects we might discuss here, but we can skip as, in our case, benefits overcame drawbacks.

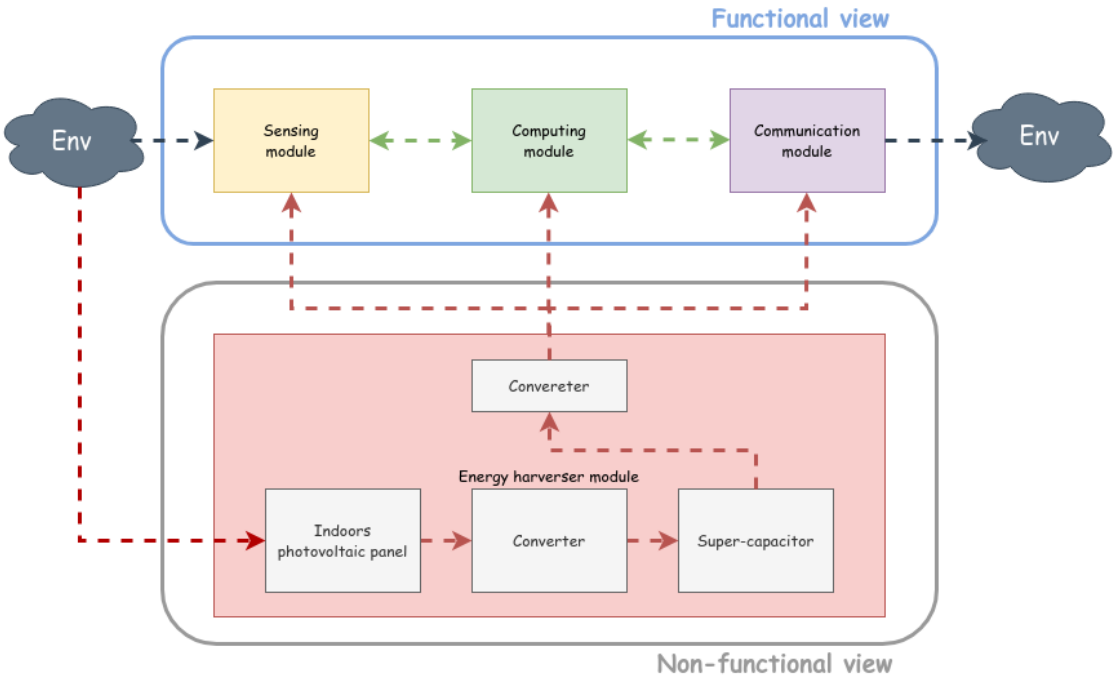


Figure 4.2: Implementation of the datalogger based on modules available in the market.

To describe the hardware implementation shown in figure 4.2, we are going to present each module as well as its main features and interfaces. Then, we are going to discuss how modules were interconnected.

4.2 Computing

In section 3.4.1, we stated that the computing sub-system is responsible of managing the entire embedded system. We may say that the *smartness* of the ES depends on its computing elements. For this reason we focused on a suitable *microcontroller unit*.

Microcontrollers (MCUs) are ICs usually containing a random access memory (RAM), a read only memory (ROM) and various peripherals, which allow them to run *tasks* independently. We will see how to run *tasks* in section 8.4. The point is that MCUs contain several interfaces typical of sensors or other devices and might have one or more analog-to-digital (ADC) and digital-to-analog (DAC) converters. They have limited functionalities and modest computational power, but they are meant for special-purpose computing systems. Most of them are **low-power consumption** devices and their cost is cheap.

After a careful market analysis and according to our specifications, we selected the **MSP430FR5994 MCU** by **Texas Instruments (TI)**.

Companies manufacturing MCUs and other complex components, such as system-on-chips (SoC) or field-programmable-gate-arrays (FPGA), sell *evaluation modules* in order to

promote their products. An **evaluation module (EVM)** is used by developers to determine whether a semiconductor component is a good or the best fit for the target application. We bought the **MSP-EXP430FR5994 LaunchPad™ Development Kit** (figure 4.3). According to the manufacturer,

> "it is an easy-to-use evaluation module for the MSP430FR5994 MCU. It contains everything needed to start developing on the ultra-low-power MSP430FR5994 FRAM MCU platform, including onboard debug probe for programming, debugging and energy measurements"[22].

We implemented the computing sub-system by using this evaluation module.

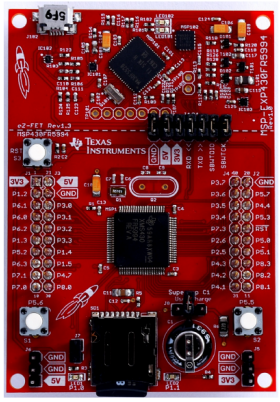


Figure 4.3: MSP-EXP430FR5994 LaunchPad™ Development Kit [22].

4.2.1 The module

Let us have an overview of the module looking at figure 4.4.

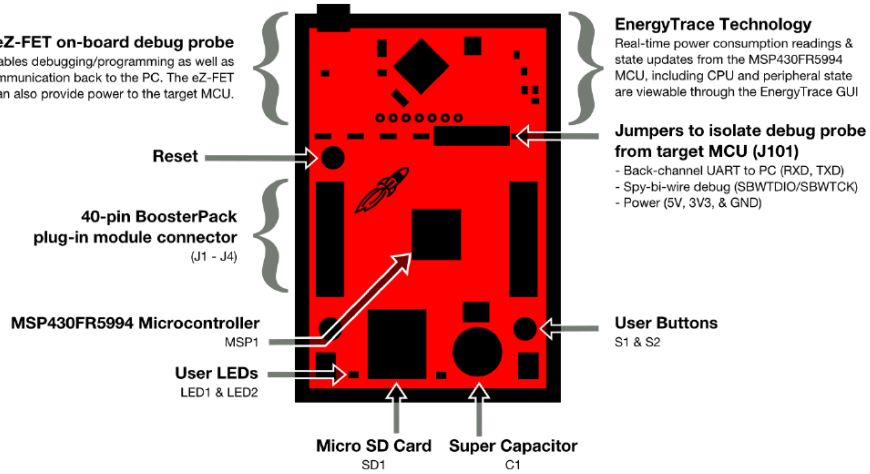


Figure 4.4: MSP-EXP430FR5994 Overview [22].

Its core is the MSP430FR5994 16-bit MCU to be evaluated. The MCU can be easily restarted thanks to the onboard reset button. The module hosts further two onboard buttons and two LEDs for user interaction, a microSD card port to interface with microSD cards and a super-capacitor to enable stand-alone applications without an external power supply. The module includes an onboard eZ-FET debug probe with EnergyTrace++™ technology. Finally, it hosts a set of jumpers and a 40-pin plug-in module. For completeness, when you buy the module, a micro-USB cable and a quick start guide to start developing are included.

4.2.2 Isolation Jumper Block

The module is logically divided into two regions or domains: the eZ-FET debug probe domain and the MSP430FR5994 target domain. The regions can be also physically disconnected each other thanks to the **isolation jumper block (J101)**. Its role is to connect or disconnect signals that cross from the eZ-FET domain into the MSP430FR5994 target domain (figure 4.5). This includes eZ-FET Spy-Bi-Wire signals, application UART signals, 3.3 V and 5 V power supply.

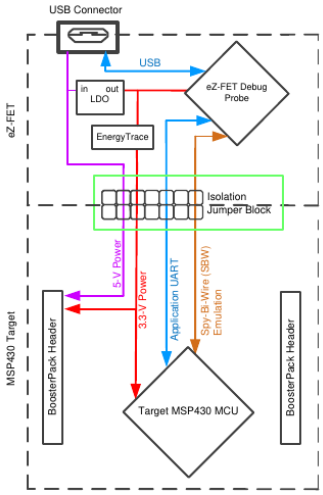


Figure 4.5: MSP-EXP430FR5994 eZ-FET Isolation Jumper Block Diagram [22].

The reader might wonder why opening the connections. There are good reasons. In particular [22]:

- *to remove any and all influence from the eZ-FET debug probe for high accuracy target power measurements*;
- *to control* 3.3 V *and* 5 V *power flow between the eZ-FET and target domains*;
- *to expose the target MCU pins for other use than onboard debugging and application UART communication*;
- *to expose the programming and UART interface of the eZ-FET so that it can be used for devices other than the onboard MCU.*

Jumper	Description
GND	Ground
5V	5 V VBUS from USB
3V3	3.3 V rail, derived from VBUS in the eZ-FET domain
RXD «	Backchannel UART: The target MSP430FR5994 receives data through this signal. The arrows indicate the direction of the signal
TXD »	Backchannel UART: The target MSP430FR5994 sends data through this signal. The arrows indicate the direction of the signal
SBW RST	Spy-Bi-Wire debug: SBWTDIO data signal. This pin also functions as the RST signal (active low)
SBW TST	Spy-Bi-Wire debug: SBWTCK clock signal. This pin also functions as the TST signal

Table 4.1: Isolation Block Connections [22].

For completeness, table 4.1 reports the role of each jumper. The isolation jumper block had a key role in our project as we will discuss later on.

4.2.3 eZ-FET Onboard Debug Probe With EnergyTrace++ Technology

The module integrates an onboard **eZ-FET debug probe** which eliminates the need for expensive programmers. In particular, it is a simple and low-cost debugger that supports all MSP430 device derivatives. The eZ-FET also provides a backchannel UART-over-USB connection with the host, which can be very useful during debugging and for easy communication with a PC. The module features full **EnergyTrace++ technology™** which can be used to measure the current draw of the MSP430FR5994 and see energy profiles through integrated GUI into development tools [22]. We will exploit these features in chapter 9. The eZ-FET debug probe domain is shown in figure 4.6.

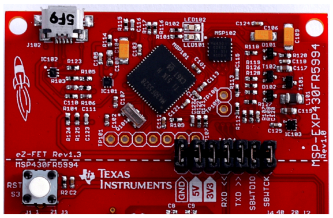


Figure 4.6: MSP-EXP430FR5994 eZ-FET Debug Probe [22].

4.2.4 BoosterPack Plug-in module

The module complies with the 40-pin LaunchPad™ development kit pinout standard. This standard was created to aid compatibility between LaunchPad™ development kits and **BoosterPack** plug-in modules across the TI ecosystem [22]. We were not interested in connecting any of these TI modules; our goal was to exploit its presence to reach the MCU.

We were not allowed to access all the pins, but a subset, as you can see in figure 4.7. It was not a limitation for our project because all we needed was available. We will resume the topic in section 4.7.

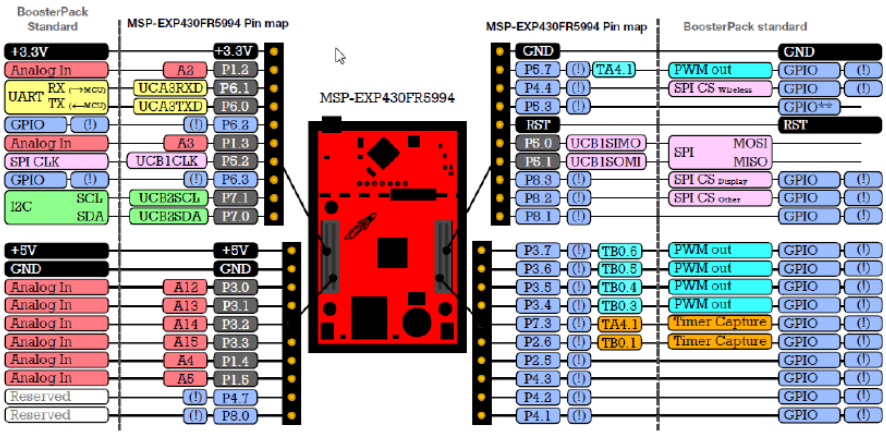


Figure 4.7: LaunchPad™ to BoosterPack Plug-in Module Connector Pinout [22].

4.2.5 MSP430FR5994 MCU

The module hosts a MSP430FR5994 MCU. It is a 16-bit reduced instruction set computer (RISC) architecture. The clock frequency can go up to 16 MHz. The supply voltage range extends from 1.8 V to 3.6 V.

The complete architecture will be presented in chapter 5. Nonetheless, we anticipate here that the MCU features 256 kB of embedded **ferroelectric random access memory (FRAM)**, a nonvolatile memory known for its ultra-low power, high endurance and high speed write access. The device also features 8 kB of SRAM. Furthermore, other key features are [22]:

- 16-channel 12-bit ADC;
- 16-channel analog comparator;
- six 16-bit timers with seven capture/compare registers each;
- 6-channel direct memory access (DMA);
- 128-bit or 256-bit AES;
- 32-bit hardware multiplier (MPY);
- 68 GPIOs.

We have already stated that we were not allowed to access all the MCU pins via the plug-in module headers, but it was not a limitation. However, we report the complete pinout in figure 4.8. MSP430 MCUs follow a particular **convention** to share multiple functions through a pin. Let us give an example. Pin 4 is P3.0/A12/C12. It means the

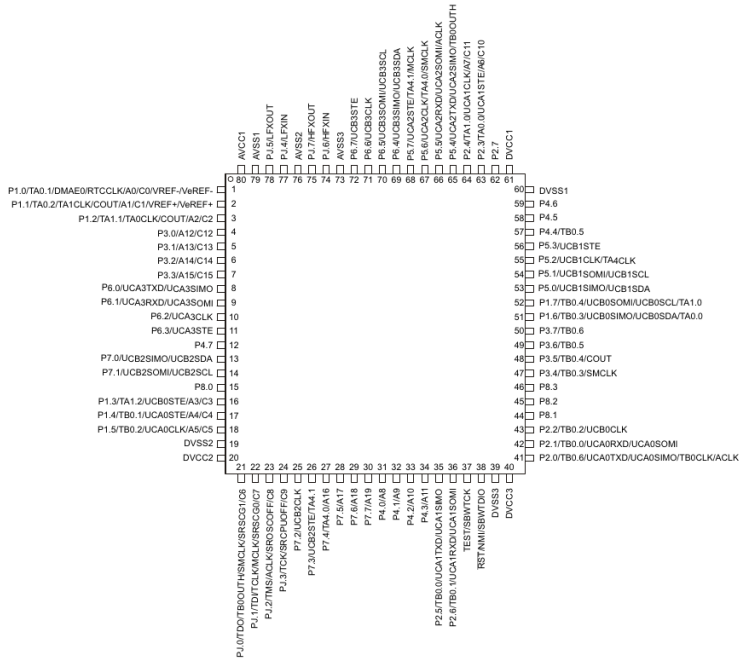


Figure 4.8: MSP430FR5994 Pinout [22].

pin can be used as *general-purpose input/output* P3.0 (first bit of port 3), as *analog input* A12 or as *comparator input* C12. The function can be selected thanks to a proper port configuration [3]. The role of each pin can be understood from the MCU datasheet.

4.2.6 A clarification

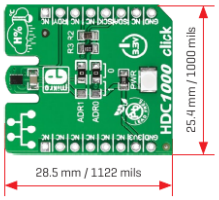
We would like to conclude this section with a clarification. Among the available resources of the module, we exploited the reset button and LEDs. We never exploited the user buttons and the microSD card port. In particular, we did not exploit the presence of the super-capacitor. The *energy storage element* is a topic discussed in section 4.5.

4.3 Sensing

To measure temperature and humidity, we selected a module by **MIKROE**. It is a Serbian development tools company dedicated to standardization and time-saving in the embedded industry [30]. It produces and sells not only compilers, development boards, starter boards, accessories, kits, MCU cards and much more, but also particular modules called *Click boards*^{TM}. According to manufacturers,

> "**Click boards**$^{\mathbf{TM}}$ are made to save developers time because, by using them, they do not have to worry about designing a new board just to test some idea or concept. They are a perfect solution for users which require tested and affordable hardware, supported with free libraries and technical support. They simplify integration and later upgrade of the project or products"[30].

Our choice was to buy a **HDC1000 click™**, a temperature and humidity measurement click™ board carrying the **HDC1000** sensor from **Texas Instruments** (figure 4.9). The HDC1000 click™ communicates with the computing module through *mikroBUS™* I2C lines, SCL and SDA, plus the INT pin, used here for DRDY (Data Ready). The board is designed to use a 3.3 V power supply only. We implemented the sensing sub-system by using this module.



	mm	mils
LENGTH	28.5	1122
WIDTH	25.4	1000
HEIGHT*	4	157.5

* without headers

Figure 4.9: HDC1000 click™ [28].

4.3.1 The module

Let us have a look to the schematic of the module referring to figure 4.10. The module is powered thanks to 3.3 V and GND *mikroBUS™* pins.

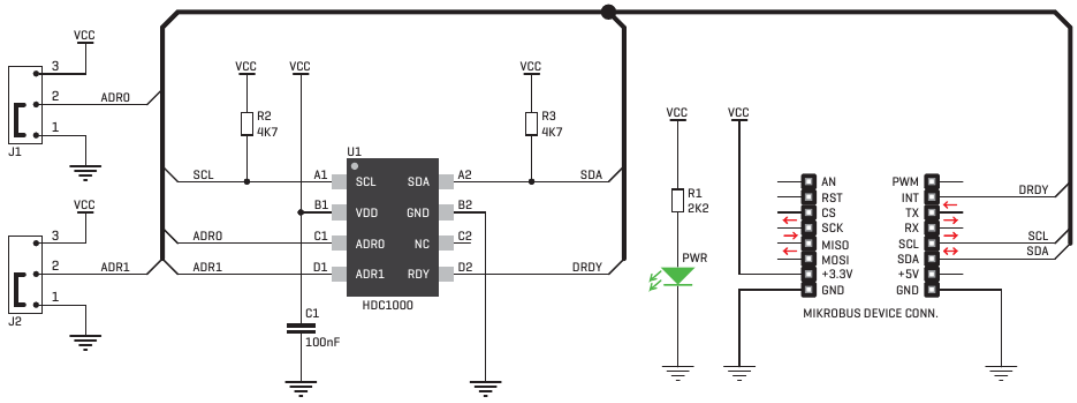


Figure 4.10: HDC1000 click™ schematic [28].

The HDC1000 IC has 8 pins. Pin B1 is VDD; it is a power pin whose role is to provide the supply voltage. B2 is GND and must be connected to ground. The HDC1000 requires a voltage supply within 2.7 V to 5.5 V. A multilayer ceramic by-pass X7R capacitor of 0.1 µF between VDD and GND pins is recommended according to the datasheet. Pin A1 is SCL. It is an input pin which serves a serial clock line for **Inter Integrated Circuit (I2C)**; since it is open-drain, it requires a pull-up resistor to VDD. Pin A2 is SDA. It is an input/output pin which serves a data line for I2C; since it is open-drain, it requires a pull-up resistor to VDD. Designers chose 4.7 kΩ pull-up resistors for SCL and SDA pins. Pin D2 is DRY. It is an output active low pin whose purpose is to inform that data are ready; since it

is open-drain, it requires a pull-up resistor to VDD. If not used, it must be tied to GND. C1 and D1 are respectively ADR0 and ADR1. They are input address select pins and are hardwired to GND or VDD. They are needed because the HDC1000 sensor acts as a **slave** on the I2C interface. Two jumpers (zero-ohm resistors) on the click™ board let you specify the slave address byte. Default is GND. Finally, C2 pin is NC. It is not connected or could be connected to GND [17].

We would like to point out that the presence of a LED is fine to verify that the module is correctly powered, but it represents a wasted current. We measured $V_{AK} = 1.86\,\text{V}$ across the diode and we calculated the current:

$$I_{AK} = \frac{V_{DD} - V_{AK}}{R_1} = \frac{3.3\,\text{V} - 1.86\,\text{V}}{2.2\,\text{k}\Omega} = 0.65\,\text{mA} \tag{4.1}$$

We may say that the presence of a by-pass jumper would have been appreciated. We will be facing this kind of topics in chapter 10.

4.3.2 mikroBUS™

MIKROE created an open standard called **mikroBUS™**. It defines mainboard sockets and add-on boards used for interfacing microcontrollers or microprocessors (mainboards) with integrated circuits and modules (add-on boards). According to creators,

> "the standard specifies the **physical layout** of the mikroBUS™ pinout, the communication and power supply pins used, the size and shape of the add-on boards and the positioning of the mikroBUS™ socket on the mainboard. Finally, the silkscreen marking conventions for both the add-on boards and sockets are specified. Its main purpose is to enable easy hardware expandability with a large number of standardized compact add-on boards, each one carrying a single sensor, transceiver, display, encoder, motor driver, connection port or any other electronic component"[29].

Such feature was not relevant in our project, but it could be highly exploited for modularity and scalability purposes. This topic will be resumed in chapter 10.

4.3.3 The sensor

According to the datasheet,

> "the HDC1000 is a digital **humidity** sensor with integrated **temperature** sensor that provides excellent measurement accuracy at **very low power** and long term.
>
> The sensing element of the HDC1000 is placed on the bottom part of the device which makes the HDC1000 more robust against dirt, dust and other environmental contaminants. Measurement results can be read out through the I2C compatible interface. The **resolution** is based on the measurement time and can be 8, 11, or 14 bits for humidity; 11 or 14 bits for temperature. The HDC1000 is functional within the full −40 °C to 125 °C temperature range. The percentage of relative humidity is measured"[17].

These features met NFR12 and NFR13 non functional requirements of our project.

> "One of the key features of the HDC1000 is its low power consumption which makes the device suitable in battery or **power harvesting applications**. In these applications the HDC1000 spends most of the time in **sleep mode** with a typical 110 nA of current consumption. The averaged current consumption is minimal"[17].

More information can be retrieved from its datasheet, but we will return to the sensor in chapter 5.

4.4 Communication

From requirements and design activities, we needed to target a **low-power** and **long-range** wireless interface. In particular, in section 3.4.1 we stated our choice was based on a LoRaWAN® transceiver. As implementation choice we selected the **Grove Wio-E5** module by **seeedstudio** (figure 4.11).

Figure 4.11: Grove Wio-E5 module [39].

The module embedded with Wio-E5 **STM32WLE5JC**, powered by ARM Cortex M4 ultra-low-power MCU core and Wio SX126x, is a wireless radio module supporting LoRa® and LoRaWAN® protocols on the EU868 and US915 frequency and (G)FSK, BPSK, (G)MSK, LoRa® modulation [39]. We implemented the communication sub-system by using this module.

4.4.1 The module

The module is extremely compact and its core is the STM32WLE5JC IC (figure 4.12). A

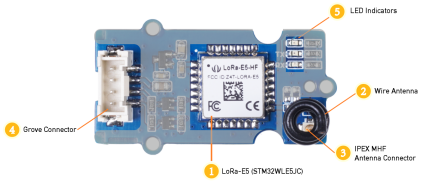


Figure 4.12: Grove Wio-E5 module main components [39].

wire antenna is mounted on a suitable connector. Useful LEDs are mounted to display its

activities. The module interfaces to the outside world thanks to the Grove Connector. This connector allows the module to be powered thanks to 3.3 V power supply and controlled by *AT commands* through a UART connection.

4.4.2 STM32WLE5JC

The STM32WLE5JC is the main component. According to manufacturers,

> "it is designed for IoT device development, testing and long-distance, **ultra-low power consumption** IoT scenarios like smart agriculture, smart office, and smart industry. In particular, it is designed with industrial standards with a wide working temperature at −40 °C to 85 °C, high sensitivity between −116.5 dB m to −136 dB m, and power output between 10 dB m to 22 dB m"[39].

Table 4.2 reports the STM32WLE5JC main features.

Feature	**Description**
Core	32-bits Arm Cortex-M4 CPU, up to 48 MHz
LoRaWAN stack	Built-in AT Command Firmware
Package	12 mm x 12 mm, 28 pins SMD
Interfaces	UARTx3, I2Cx1, ADC(12-bit)x1, SPI 1, GPIOx6
Sensitivity	−116.5 dB m (SF5), −121.5 dB m(SF7), −136 dB m (SF12)
Modulation	LoRa®, (G)FSK, (G)MSK and BPSK
Certificate	FCC and CE (EU868/US915)
Power Supply	1.8 V to 3.6 V
RF Output Power	up to 20 dB m at 3.3 V

Table 4.2: STM32WLE5JC main features [39].

Our main goal was to control the module via *AT commands*. It was possible because the IC hosts a built-in AT Command firmware. More information can be retrieved from its datasheet [40]. We will return to the IC and AT commands in chapter 7.

4.5 Energy harvesting

Computing, sensing and communication modules had to be powered. The key point was to meet NFR1, NFR2 and NFR3 non functional requirements:

- a datalogger is a **batteryless system**;
- a datalogger harvests energy thanks to a **photovoltaic panel**;
- a datalogger works **indoors**.

First issue was about the power supply required by each module to work. Table 4.3 summarizes the data we retrieved from the datasheets. Actually, data were related to main components such as the MCU or the sensor IC, but the modules were designed to be powered with **3.3 V**. This for us was the target voltage.

Module	Required voltage
Computing	1.8 V to 3.6 V
Sensing	2.7 V to 5.5 V
Communication	1.8 V to 3.6 V

Table 4.3: Power supply required by modules.

To meet non functional requirements, we carefully selected a solution from the market: the Solar Development Kit by PowerFilm® [35]. The kit involves several components: a module, an indoor photovoltaic cell, an outdoor photovoltaic cell, a Li-ion battery and two super-capacitors. For our project we focused on:

- the module (2EAEM10941C0011);
- the indoor photovoltaic cell (LL200-2.4-37);
- the dual-cell super-capacitor (DMT3N4R2U224M3DTA0).

We implemented the harvesting sub-system by using these three components.

4.5.1 The module

The module by e-peas, shown in figure 4.13, is design around the **AEM10941** IC, in particular, it manages its power, configuration, control and status signals.

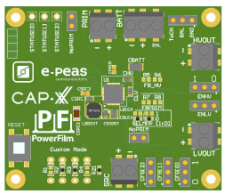


Figure 4.13: 2EAEM10941C0011 energy harvester module [36].

The module can be used for different purposes, but in our project we were interested in using the circuit shown in figure 4.14. That is:

- connecting the photovoltaic cell to SRC connector to harvest energy;
- connecting the dual-cell super-capacitor to BATT connector to store energy;
- connecting the sensing, computing and communication modules to HVOUT connector to power loads.

The module had to be configured as we are going to discuss shortly. It provides the user three logic output status signals, but their values had to be converted; this topic will be discussed in section 4.6.

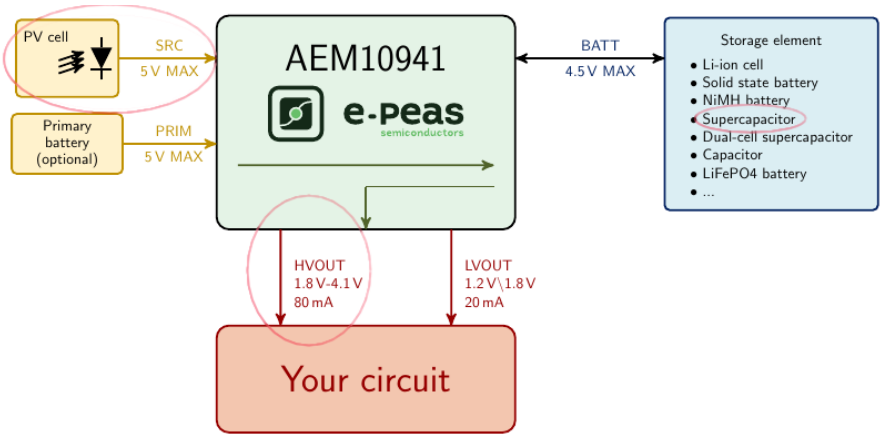


Figure 4.14: 2EAEM10941C0011 energy harvester module setup [11].

AEM10941

We introduced this EMU in section 2.4.4 as case study. According to manufacturers,

> "the AEM10941 is an integrated energy management unit (EMU) that extracts DC power from up to 7-cell solar panels to simultaneously store energy in a rechargeable element and supply the system with two independent regulated voltages. The AEM10941 allows to extend battery lifetime and *ultimately eliminates the primary energy storage element in a large range of wireless applications*, such as industrial monitoring, geolocation, home automation, e-health monitoring and *wireless sensor nodes*"[11].

The AEM10941 harvests the available input current up to 110 mA. It integrates an ultra-low power boost converter to charge a storage element, such as a Li-ion battery, a thin film battery, a super-capacitor or a conventional capacitor. The boost converter operates with input voltages in a range from 50 mV to 5 V. With its unique cold-start circuit, it can start operating with empty storage elements at an input voltage as low as 380 mV and an input power of just 3 µW.

The low-voltage supply typically drives a microcontroller at 1.2 V or 1.8 V. The high-voltage supply may typically drive a radio transceiver at a configurable voltage between 1.8 V and 4.1 V. Both are driven by highly-efficient low drop-out (LDO) regulators for low noise and high stability [11].

Configuration

Let us describe how we configured the module by extracting data from the datasheet and taking into account our needs. We will refer to figure 4.15.

The AEM10941 (AEM) is an energy management unit able to charge a storage element, battery or super-capacitor connected to BATT, from an energy source connected to SRC, as well as to supply loads at different operating voltages through two powers supplying LDO regulators LVOUT and HVOUT [11]. In our project, we were interested in charging a super-capacitor from a photovoltaic cell, as well as supplying loads at 3.3 V thanks to

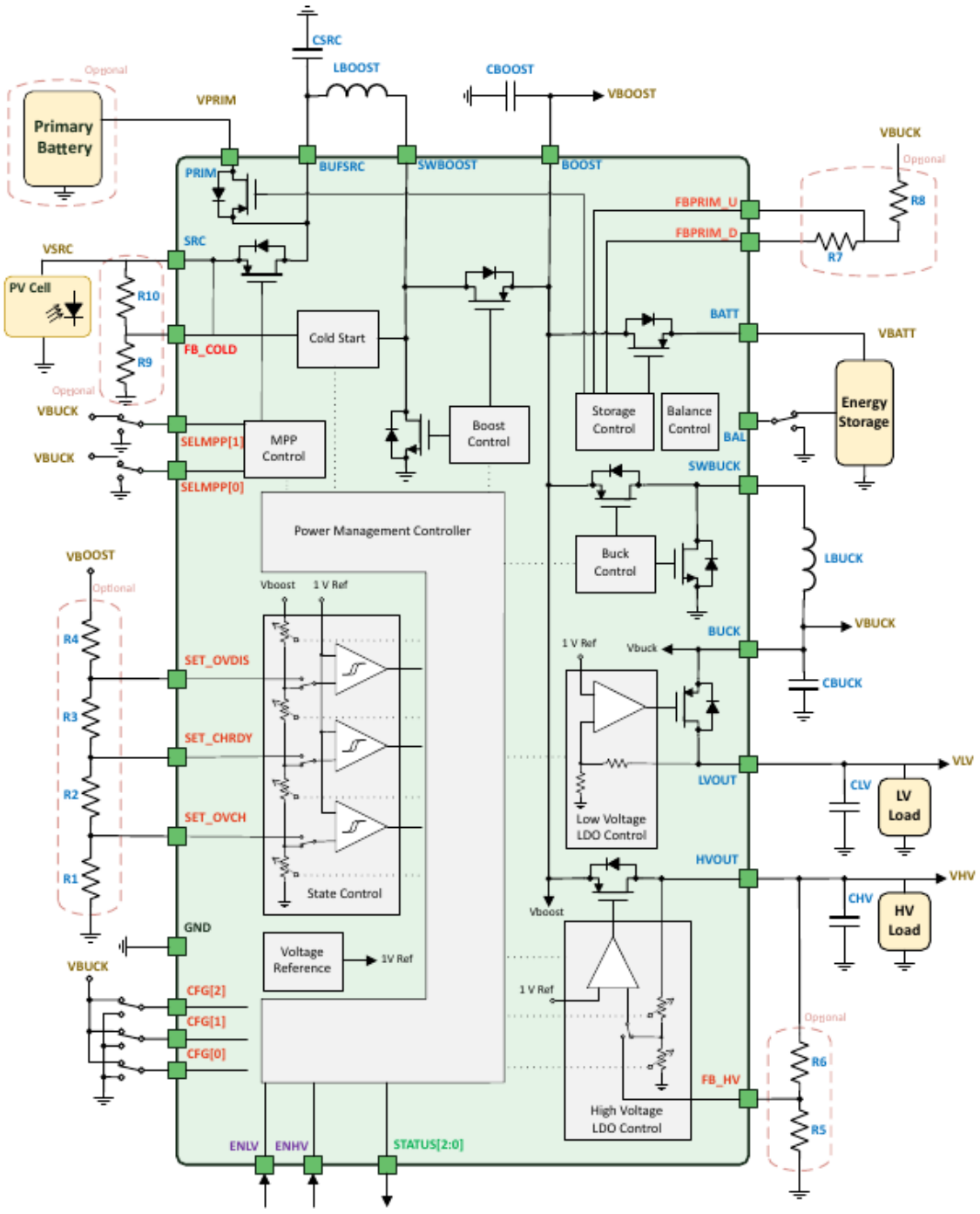


Figure 4.15: AEM10941 functional block diagram [11].

the HVOUT regulator. Indeed, the heart of the AEM10941 is a cascade of two regulated switching converters, namely the boost converter and the buck converter with high-power conversion efficiency, as already discussed in section 2.4.4.

At first start-up, as soon as a required **cold start voltage** (V_{cs}) of 380 mV and a scant amount of power of just 3 µW available from the photovoltaic cell, the AEM cold starts. After the cold start, the AEM can extract the power available from the source as long as the input voltage is comprised between 50 mV and 5 V. Note that the minimum voltage for the cold start may be set by adding resistors R9 and R10 [11]. By default the module does not mount these resistors, thus the cold start is set to the minimum value of 380 mV. However, the user may increase the value by defining:

$$R_c = R_9 + R_{10} \quad (100\,\text{k}\Omega \leq R_c \leq 10\,\text{M}\Omega) \tag{4.2}$$

$$R_9 = 0.38\,\text{V} \cdot R_c / V_{cs} \tag{4.3}$$

$$R_{10} = R_c - R_9 \tag{4.4}$$

Through three configuration pins **CFG[2:0]**, the user can select a specific operating mode from a range of seven modes that cover most application requirements without any dedicated external component. These operating modes define the LDO output voltages and the protection levels of the storage element [11]. For our project, we were interested in using the **dual-cell super-capacitor** operating mode. This configuration could be easily performed thanks to suitable jumpers mounted on the module. Figure 4.16 reports the available configurations and the selected one. Let us better describe the configuration.

Configuration pins			Storage element threshold voltages			LDOs output voltages		Typical use
CFG[2]	CFG[1]	CFG[0]	Vovch	Vchrdy	Vovdis	Vhv	Vlv	
1	1	1	4.12 V	3.67 V	3.60 V	3.3 V	1.8 V	Li-ion battery
1	1	0	4.12 V	4.04 V	3.60 V	3.3 V	1.8 V	Solid state battery
1	0	1	4.12 V	3.67 V	3.01 V	2.5 V	1.8 V	Li-ion/NiMH battery
1	0	0	2.70 V	2.30 V	2.20 V	1.8 V	1.2 V	Single-cell supercapacitor
0	1	1	4.50 V	3.67 V	2.80 V	2.5 V	1.8 V	Dual-cell supercapacitor
0	1	0	4.50 V	3.92 V	3.60 V	3.3 V	1.8 V	Dual-cell supercapacitor
0	0	1	3.63 V	3.10 V	2.80 V	2.5 V	1.8 V	LiFePO4 battery
0	0	0	Custom mode - Programmable through R1 to R6				1.8 V	

Figure 4.16: AEM10941 configuration pins [11].

Vovch, Vchrdy and Vovdis are threshold voltage levels and work as **protection levels**. In particular:

- Vovch is the maximum voltage accepted on the super-capacitor before disabling the boost converter. For us was **4.50 V**;
- Vchrdy is the minimum voltage required on the super-capacitor after a cold start before enabling the LDOs. For us was **3.92 V**;
- Vovdis is the minimum voltage accepted on the super-capacitor before considering the storage element as depleted. For us was **3.60 V**.

In practice, LDOs were disabled until the super-capacitor reached 3.92 V. The super-capacitor could continue charging up to 4.50 V maximum value. In case of discharge, the LDOs were enabled till the 3.60 V critical value. The two LDOs output voltages are called Vhv and Vlv for the high and low-output voltages, respectively. This configuration allowed

us to get $\mathbf{Vhv = 3.3\,V}$. For the sake of completeness, we would like to report here how to perform a custom configuration to define the element protection levels and the output voltage of the high-voltage LDO. First of all, you define:

$$R_T = R_1 + R_2 + R_3 + R_4 \quad (1\,\mathrm{M\Omega} \leq R_T \leq 100\,\mathrm{M\Omega}) \tag{4.5}$$

Then, you can set the value of R1, R2 R3 and R4:

$$R_1 = R_T \cdot (1\,\mathrm{V}/Vovch) \tag{4.6}$$

$$R_2 = R_T \cdot (1\,\mathrm{V}/Vchrdy - 1\,\mathrm{V}/Vovch) \tag{4.7}$$

$$R_3 = R_T \cdot (1\,\mathrm{V}/Vovdis - 1\,\mathrm{V}/Vchrdy) \tag{4.8}$$

$$R_4 = R_T \cdot (1 - 1\,\mathrm{V}/Vovdis) \tag{4.9}$$

Next step is to define Vhv thanks to R5 and R6:

$$R_V = R_5 + R_6 \quad (1\,\mathrm{M\Omega} \leq R_V \leq 40\,\mathrm{M\Omega}) \tag{4.10}$$

$$R_5 = R_V \cdot (1\,\mathrm{V}/Vhv) \tag{4.11}$$

$$R_6 = R_V \cdot (1 - 1\,\mathrm{V}/Vhv) \tag{4.12}$$

In general, the resistors should have high values to make the additional power consumption negligible. Moreover, the following constraints must be adhered to ensure the functionality of the chip:

$$Vchrdy + 0.05\,\mathrm{V} \leq Vovch \leq 4.5\,\mathrm{V} \tag{4.13}$$

$$Vovdis + 0.05\,\mathrm{V} \leq Vchrdy \leq Vovch - 0.05\,\mathrm{V} \tag{4.14}$$

$$2.2\,\mathrm{V} \leq Vovdis \tag{4.15}$$

$$Vhv \leq Vovdis - 0.3\,\mathrm{V} \tag{4.16}$$

The maximum power point tracking (MPPT) ratio can be configured using two configuration pins $\mathbf{SELMPP[1:0]}$. The pins allow to select the MPP tracking ratio based on the characteristic of the photovoltaic cell. On the board these pins are hardwired to ground, which implies MPPT is set to 70 % [11].

Two logic control pins are provided, ENLV and ENHV, to dynamically activate or deactivate the LDO regulators that supply the low and high-voltage load, respectively. The status pin STATUS[0] alerts the user that the LDOs are operational and can be enabled. This signal can also be used to enable an optional external regulator. In our project, we activated LDOs thanks to suitable jumpers mounted on the board. Figure 4.17 reports the possible configurations and the selected one.

With our previous configuration, when the super-capacitor reached the 3.60 V critical value, the LDOs were power gated and the controller was no longer supplied by the super-capacitor to protect it from further discharge. Around 600 ms before the **shutdown** of the AEM, the status pin STATUS[1] alerted us for a clean shutdown of the system.

The status of the MPPT controller was reported with one dedicated status pin: STATUS[2]. The status pin was asserted when an MPPT calculation was being performed.

ENLV	ENHV	LV output	HV output
1	1	Enabled	Enabled
1	0	Enabled	Disabled
0	1	Disabled	Enabled
0	0	Disabled	Disabled

Figure 4.17: AEM10941 LDOs enabling [36].

We did not need a primary battery. This condition had to be signaled to the board thanks to two jumpers called NoPrim. In our project, we set the jumpers according to specifications.

Finally, our project used a dual-cell super-capacitor as storage element. On the board there is a jumper whose role is to set the **balun circuit**. The balun circuit balances the internal voltage in the dual-cell super-capacitor to avoid damaging it because of excessive voltage on one cell. To enable the circuit, we set up the jumper so that the BAL pin was connected to toCn pin.

It is important to point out that more information can be retrieved from AEM10941 datasheet [11]. What we have reported here is a subset of the entire documentation. In particular, many pictures and diagrams may be suitable for future work.

4.5.2 The supercapacitor

The **dual-cell super-capacitor** is from **CAP-XX** (figure 4.18), a world leader in the design and manufacture of thin, flat super-capacitors and energy management systems used in portable and small-scale electronic devices. Table 4.4 reports its main features.

Figure 4.18: DMT3N4R2U224M3DTA0 super-capacitor [36].

The notable feature of CAP-XX super-capacitors is their very **high power density** and **high energy storage capacity** in a space-efficient prismatic package. These attributes are essential in power-hungry consumer and industrial electronics, and deliver similar benefits in automotive and other transportation applications.

Voltage(V)	**Capacitance(mF)**	**ESR(mΩ)**	**Dimensions(mm)**	**Height(mm)**
4.2	220	300	21 x 14	2.2

Table 4.4: DMT3N4R2U224M3DTA0 super-capacitor main features [36].

We could have soldered the super-capacitor directly on the module as its hosts a suitable footprint. However, we did not use this possibility and we preferred connecting the super-capacitor via BATT connector.

4.5.3 The photovoltaic cell

The **photovoltaic cell** is by **PowerFilm** (figure 4.19). This company designs and manufactures custom solar cells, modules, panels and power solutions for energy harvesting and portable power applications using proprietary **thin-film silicon technology** [34]. Table

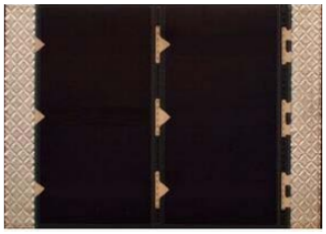


Figure 4.19: LL200-2.4-37 photovoltaic cell [36].

4.5 reports its main features. Vmp, Impp, and Pmp represent guaranteed minimum performance. Other values are average or typical behavior. The cell is tested for performance under a cool white LED spectrum.

Illuminance(lx)	Voc(V)	Isc(µA)	Vmp(V)	Impp(µA)	Pmp(µW)
200	2.4	60	1.6	43	69
1000	2.9	200	2.1	214	449

Table 4.5: LL200-2.4-37 photovoltaic cell main features [36].

According to the manufacturer,

> "the PV material is inherently sensitive to collecting low intensity light, making it ideal for **low power indoor/outdoor** IoT and energy harvesting applications where light conditions are always changing. In particular, indoor panels have excellent performance under artificial lighting, whether that is a dim warehouse (200 lx) or a bright retail store (1000 lx)"[36].

The photovoltaic cell could be easily connected to the module via SRC connector.

4.6 Additional electronics

In the previous section, we have learned that the energy harvester module provides the user three logic output status signals: STATUS[2], STATUS[1] and STATUS[0]. If unused, they can be left floating, otherwise they can be exploited to monitor the module.

The problem to solve was about their **logic level** because they needed a conversion. STATUS pins high level is Vbatt, that is, the voltage of the super-capacitor. In particular, the values should be adapted to interface with MCU general-purpose input/output pins. Actually, we decided to use and adapt only STATUS[1] pin, the one asserted if the super-capacitor voltage falls under Vovdis. More details will be given in chapter 8 and chapter 10.

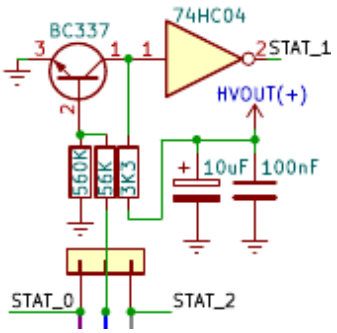


Figure 4.20: Voltage adapter for energy harvester module STATUS[1] pin.

We adapted the STATUS[1] pin voltage thanks to the circuit shown in figure 4.20. As you can see, the circuit transforms the V_s analog voltage to V_{stat_1} CMOS output digital voltage. Since V_{cc} coincides with V_{HVOUT}, that is the output voltage regulator, the V_{stat_1} high value will be 3.3 V, whereas its low value will be 0 V. The BC337 BJT transistor works as a switch; from its datasheet we obtained $H_{FE_{min}} = 100$. We also knew that $V_{s_{min}} = 3.60\,V$ and we set $I_{c_{SAT}} = 1\,mA$. By studying the circuit, we could write:

$$V_s - V_{BE} = R_1 I_B \quad \Rightarrow \quad I_B = \frac{V_s - V_{BE}}{R_1} \tag{4.17}$$

Furthermore:

$$V_{cc} - V_{CE} = R_2 I_C \quad \Rightarrow \quad R_2 = \frac{V_{cc} - V_{CE}}{I_C} \tag{4.18}$$

From the second equation, we could calculate the value of R_2:

$$R_2 = \frac{V_{cc} - V_{CE_{SAT}}}{I_{C_{SAT}}} = \frac{3.3\,V - 0.2\,V}{1\,mA} = 3.1\,k\Omega \simeq 3.3\,k\Omega\ (0.25\,W) \tag{4.19}$$

With this value, $I_{C_{SAT}} \simeq 0.93\,mA$. To calculate R_1 we set:

$$I_{B_{SAT}} > \frac{I_{C_{SAT}}}{H_{FE_{min}}} \simeq 10\,\mu A \tag{4.20}$$

To be sure the BJT would have been in saturation region, we set $I_{B_{SAT}} = 50\,\mu A$. Therefore:

$$R_1 = \frac{V_s - V_{BE_{SAT}}}{I_{B_{SAT}}} = \frac{3.6\,V - 0.8\,V}{0.05\,mA} = 56\,k\Omega\ (0.25\,W) \tag{4.21}$$

We also added the R_3 resistor so that the base of the BJT transistor was not floating:

$$R_3 \geq 10R_1 = 560\,k\Omega\ (0.25\,W) \tag{4.22}$$

The rest of the circuit is quite straightforward. We simply connected a 74HC04 CMOS inverter gate to correctly get a suitable and stable output digital voltage. We also added capacitors to IC power pins with the aim of removing noise e further stabilize the voltage.

4.7 Interconnections

After presenting all the modules, we are now ready to discuss how they were interconnected to implement the entire datalogger. Let us refer to figure 4.21 showing the practical schematic.

Our choice was to use a standard 160 mm x 160 mm x 1.6 mm **experimental board** whose outline is shown in dark yellow. Its size allowed us to easily place the four modules, the additional electronics and connectors. We mounted the computing and sensing modules by exploiting strip connectors; this solution ensured a solid mechanical stability as well as the access to the board bottom layer for electrical interconnections. We mounted the communication and energy harvesting modules by using plastic spacers; in this case, we were forced to access the bottom layer via connectors.

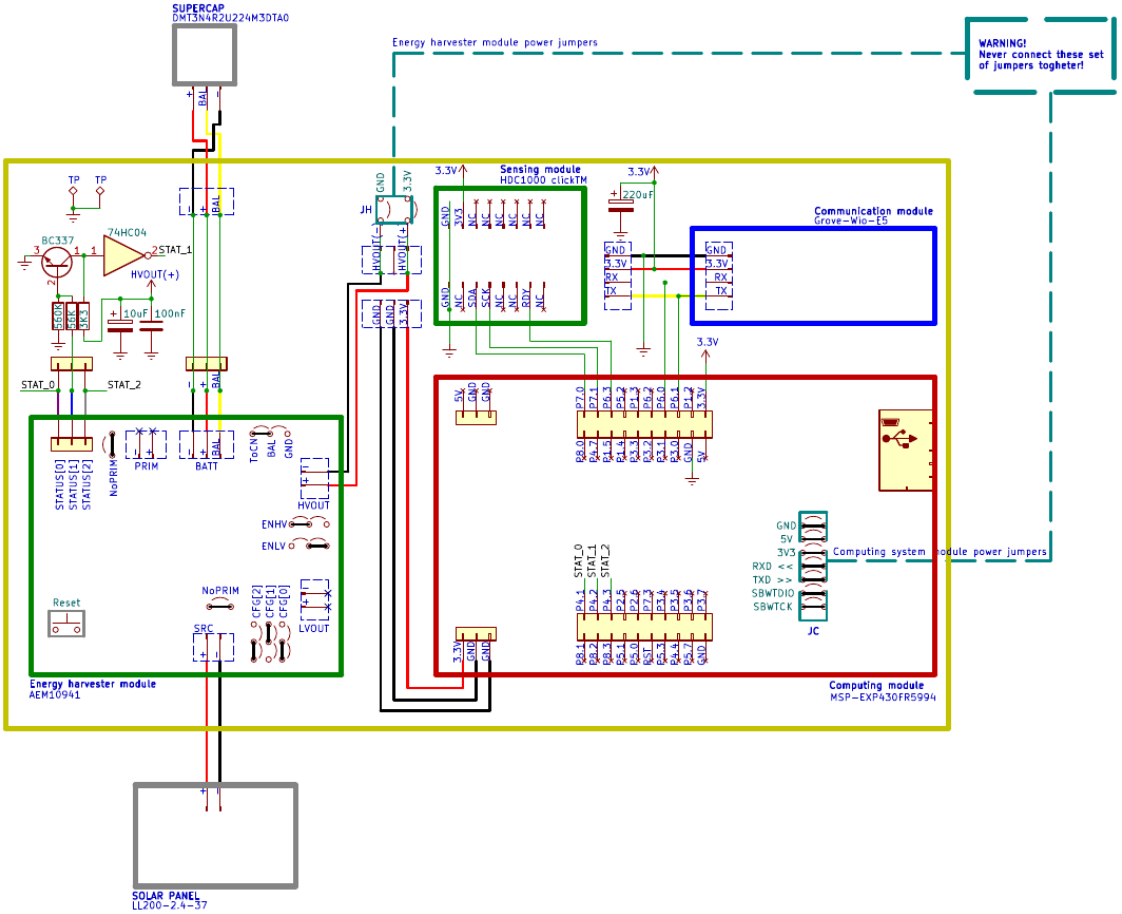


Figure 4.21: Practical schematic of the datalogger.

After cloning the project, the reader can retrieve schematics, datasheets and documentation of each module as well as the practical schematic we are analyzing. Computing, sensing and communication modules are functional units playing the role of loads in the system. These loads can be powered in two different ways which are mutually exclusive:

via USB connector or via energy harvesting module. To easily switch to a particular mode, two set of jumpers were exploited: JH (mounted on the board) and JC (mounted on the computing module §4.2.2). Such technique allowed us not to touch any connector on the board and change modality with as little effort as possible. We will discuss on how to deal with jumpers in the next section.

As we have already stated, we could have soldered the super-capacitor on the bottom layer of the energy harvester module, but we decided to use a connector to easily connect or disconnected it to the board when needed. The solar panel could be managed in the same way thanks to a screw connector which is hosted on the energy harvesting module.

The number of interconnections were not huge. The sensing module was connected to the computing module via I2C protocol and the ready signal, whereas the communication module was connected to the computing module via UART protocol thanks to a single connector. We have discussed the additional electronics in the previous section. Ground test pointers were added to facilitate the usage of probes to perform measurements. Most of the job was to separate the USB from energy harvesting module **power flow**. This feature required an interconnection strategy on the bottom layer by creating suitable tin areas.

4.8 Recommendations

This section is devoted to crucial recommendations. In particular, it is aimed at readers who will be handling the datalogger of this project.

As you know from the previous section, we built the datalogger on an experimental board. Although all the modules were firmly tied to the board and good quality connectors were used, the user should take into account the following suggestions:

- always place the board on a clean and tidy surface. This will prevent to accidentally get in touch with objects that could damage it;
- always handle wires, connectors and jumpers in a gentle way. They are robust, but the mechanical stress may weaken the connections;
- most of connections at the bottom layer were created resorting to tin soldering, but some of them are wired. These wires were carefully soldered, but the mechanical stress or vibrations may weaken the connections;
- always pay attention to exposed electronics, especially if you are wearing rings or bangles.

These set of recommendations might seem trivial, but practical electronics always require attention and care especially when dialing with experimental boards. Moreover, voltages and currents could play tricks even for low power applications. These tricks may cause serious damages to the board or worse to the user.

In the following, we are going to instruct the user to correctly perform the board programming, deploying and storing operations.

4.8.1 Programming

The programming operation modifies the firmware running into the MCU. In this case, the energy harvesting module is not involved and the datalogger is totally powered thanks to the computing module via USB cable. To perform the operation correctly, please refer to the following set of steps in order:

1. ensure the photovoltaic cell and super-capacitor are not connected;
2. disconnect JH jumpers;
3. connect all JC jumpers;
4. plug the USB cable.

After programming, the datalogger will be working as if it were not batteryless.

4.8.2 Deploying

The deploying operation allows the datalogger to work into the target environment. In this case, the datalogger is totally powered thanks to the energy harvesting module. To perform this operation correctly, please refer to the following set of steps in order:

1. unplug the USB cable (if plugged);
2. disconnect all JC jumpers;
3. reset the energy harvesting module by pushing its reset button;
4. connect the super-capacitor;
5. connect JH jumpers;
6. connect the photovoltaic cell.

After deploying, the super-capacitor will be starting storing energy according to environmental conditions.

4.8.3 Storing

The storing operation avoids damaging the datalogger after its deployment. To perform this operation correctly, please refer to the following set of steps in order:

1. disconnect the photovoltaic cell;
2. disconnect JH jumpers;
3. disconnect the super-capacitor;
4. reset the energy harvesting module by pushing its reset button.

After storing, the datalogger will not be powered anymore, neither by the computing system nor by the energy harvesting module.

In this chapter we have explained how we implemented the hardware of the datalogger. Each module implementing a particular sub-system has been presented as well as its main features and interfaces. We also have seen how modules were interconnected and how to properly handle the board. The next chapter will explain the firmware implementation of the datalogger and will take into account other significant aspects about the modules we have presented here.

Chapter 5

Datalogger: firmware implementation

In this chapter, we will explain how we implemented the firmware running on the target microcontroller. Essentially, our solution is based on a custom scheduler whose role is to manage relevant tasks. A task is the implementation of a functional requirement.

5.1 Microcontroller architecture

In section 4.2, we have presented the evaluation board we selected to implement the computing sub-system of the datalogger. It has been designed around the MSP430FR5994 MCU whose functional block diagram is shown in figure 5.1. Developers usually do not use all the sub-systems and peripherals that a microcontroller features. They normally choose a target MCU as it provides functions which are suitable for the application to develop. For our purposes, we focused and used the target blocks which are colored in red in the figure.

For the sake of completeness, we provide the reader a general description about the architecture that we extracted from its datasheet [23] and user's guide [21]. Such knowledge might be exploited in a large number of applications. However, we will describe with more details the peripherals we used in our project later in this chapter and chapter 8.

The core of the MCU is a **MSP430X CPU** (CPUXV2) with 1 MB memory access. It is a 16-bit RISC CPU working up to **16 MHz** clock.

The main feature of this MCU is the *LEA* sub-system for digital signal processing. The **low-energy accelerator (LEA)** delivers 40x the performance of Arm Cortex-M0+ MCUs to help developers efficiently process data using complex functions such as FFT, FIR, and matrix multiplication. Implementation requires no DSP expertise with a free optimized DSP Library available.

The MCU has an internal 256 kB *FRAM* memory and 8 kB *SRAM* memory space. The **ferroelectric random access memory (FRAM)** is another key feature. It allows ultra-low-power and fast writes at 125 ns per word (64 kB in 4 ms), flexible allocation of data and application code in memory and 10^{15} write cycle endurance. It is radiation resistant and nonmagnetic. FRAM technology combines the low-energy fast writes, flexibility, and endurance of RAM with the nonvolatile behavior of flash.

Both CPU and the 6-channels **direct memory access (DMA) controller** interface with peripherals through the **bus control logic**. The bus control logic manages the **memory address bus (MAB)** and the **memory data bus (MDB)**. All sub-systems and peripherals are connected to these buses as well as the **JTAG** interface.

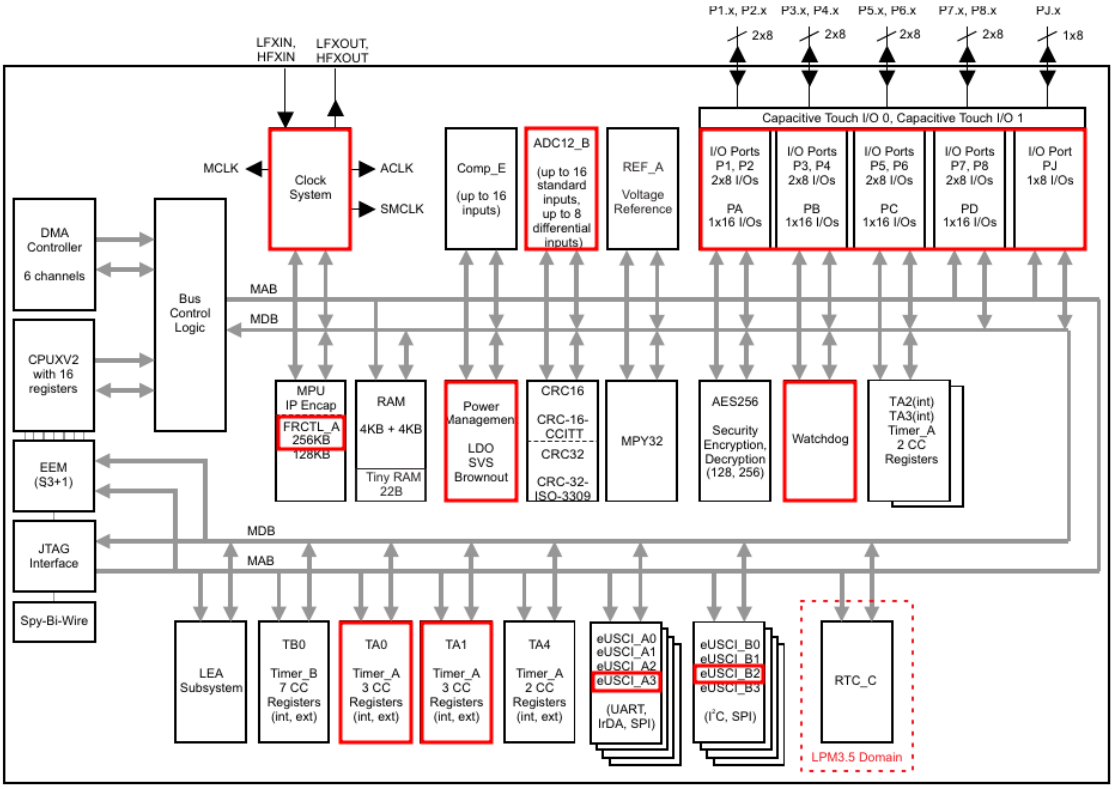


Figure 5.1: MSP430FR5994 target blocks of the project [23].

The architecture includes intelligent digital peripherals. There is a 32-bit hardware **multiplier (MPY)**, a 6-channel internal DMA we have already mentioned and a **real time clock (RTC)** with calendar and alarm functions. Moreover, there are six 16-bit **timers** with up to seven capture/compare registers each as well as 32 and 16-bit **cyclic redundancy check (CRC)**.

The architecture includes high-performance analog sub-systems such as a 16-channel analog **comparator** and 12-bit **analog-to-digital converter (ADC)** featuring window comparator, internal reference and sample-and-hold, up to 20 external input channels.

The MCU interfaces to outside world thanks to multifunction input/output **ports**. All pins support capacitive-touch capability with no need for external components. All ports are pull-up and pull-down programmable. They can be accessed bit-, byte-, and word-wise (in pairs). All ports are edge-selectable wake from **low-power modes (LPM)**. We will discuss about LPM in chapter 8.

The MCU allows to perform code **security** and **encryption**. Indeed, there is a 128 or 256 bit **AES** security encryption and decryption coprocessor. It is possible to generate random number seed for random number generation algorithms. The IP encapsulation

protects memory from external access.

The MCU allows enhanced **serial communication**. There are up to four eUSCI_A serial communication ports; in particular, we have **UART** with automatic baud-rate detection as well as IrDA encode and decode. There are also up to four eUSCI_B serial communication ports; in particular, we have **I²C** with multiple-slave addressing.

The **clock system** is flexible and can be used for a variety of purposes as we will see later on.

5.2 Driver Library

It is the time to introduce the approach we used for implementing the firmware. The reader should be aware of this before inspecting the code.

Developers usually resort to **low-level** programming methods with the aim of describing complex *tasks*. This choice allows developers to control any aspect of the code as well as the details. Let us give an example. Imagine to write an **assembly** program for the target microcontroller capable of blinking a LED. Its structure could be the following [3]:

```
        .cdecls C, LIST,"msp430.h"

        .def RESET
        .text
        .retain
        .retainrefs

RESET:
         mov.w #__STACK_END, SP           ; Stack init.
         mov.w #WDTPW | WDTHOLD, &WDTCTL  ; Hold the watchdog timer
         bic.b #LOCKLPM5, &PM5CTL0        ; Clear LOCKLPM5 bit
         bis.b #0x01, &P1DIR              ; P1DIR
         bis.b #0x01, &P1OUT              ; P1OUT
_loop:
         xor.b #0x01, &P1OUT              ; XOR
         mov.w #0xF400, R10               ; R10 <-0xF400
_lp1:
         dec.w R10                        ; Decrement
         cmp.w #0x00, R10                 ; Compare
         jne _lp1                         ; Conditional branch
         jmp _loop                        ; Branch
         nop
         .global __STACK_END
         .sect .stack
         .sect ". reset"
         .short RESET
```

Listing 5.1: Programming at low-level.

If the program is successfully built and flashed, the red LED mounted on the MSP-EXP430FR5994 LaunchPad™ Development Kit will blink. We are not now interested in examining the code, although easy to understand. The point is this approach would have allowed us to describe the behavior of the datalogger with the maximum level of detail, but the code would have been difficult to manage. However, such approach is relevant for crucial peaces of code and sometimes is necessary.

A better approach takes into account the **register level** programming. The idea is to control registers directly to gain the access of the MCU. Developers understand low-level hardware and software aspects of an embedded system by working with a higher level of abstraction. Most of register level programs are based on **C/C++** programming languages [3]. As an example, let us rewrite the previous code resorting to the register level style. Its structure could be the following [3]:

```
#include <msp430.h>

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;    // Hold the watchdog timer
    PM5CTL0 &= ~LOCKLPM5;        // Clear LOCKLPM5 bit
    P1DIR |= 0x01;               // Set the direction, P1.0
    P1OUT |= 0x01;               // Set the output value

    while (1) {
        P1OUT ^= 0x01;           // Toggle, P1.0
        __delay_cycles(250000); // Delay
    }
    return 0;
}
```

Listing 5.2: Programming at register-level.

As you can see, the code is more programmer-friendly and developers are able to better control their algorithms with the goal of describing complex *tasks*.

Actually, another possible solution is to work with a higher level of abstraction with two main purposes: reducing the effort of describing complex *tasks* and allowing code portability. Texas Instruments provides the **MSP430 Peripheral Diver Library** which supports a higher-level programming method and can be considered a *middleware* layer [3]. Since this is one of the recommended methods in writing a program for an MSP430FR5994 MCU, we followed the recommendation. As an example, let us write the blinking program resorting to the library. Its structure could be the following [3]:

```
#include <msp430.h >
#include "driverlib.h"

int main (void) {
    // Hold the watchdog timer
    WDT_A_hold(WDT_A_BASE);
    // Clear LOCKLPM5 bit
    PMM_unlockLPM5();
    // Set P1.0 as output
    GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN0);

    while (1) {
        GPIO_toggleOutputOnPin(GPIO_PORT_P1, GPIO_PIN0); // Toggle, P1.0
        __delay_cycles(250000);                          // Delay
    }
    return 0;
}
```

Listing 5.3: Programming with MSP430 peripheral driver library.

In practice, the MSP430 Peripheral Diver Library is a set of drivers for accessing the peripherals, such as GPIOs, UART and timers, found on the MSP430 FR5xx/FR6xx family of microcontrollers . Among many advantages, the library is well-documented and drivers are written entirely in C. Although, drivers are reasonably efficient in terms of memory and processor usage, they might not be so efficient for particular applications and do not support the full capabilities of the hardware [20]. Such limitations should be known by developers. However, users have a huge variety of examples available out there that can be used to write a robust and portable code. The library must be included in your project, but the operation is straightforward as, Code Composer Studio (§A.1) and other IDEs, allow developers to create a driver library based project from scratch. That's why the reader will find a driverlib folder into the root Node folder of the datalogger implementation. It was included when we created the project to start using the library.

Finally, there is a further higher level of abstraction. Indeed, it is possible to implement a **real time operating system (RTOS)**-based solution. A RTOS might simplify and accelerate the development as it eliminates the need of creating basic software functions. Texas Instruments provides the TI-RTOS(§A.1.4) whose core is a real-timer kernel [3]. The usage of a RTOS may allow new implementation choices and strategies. We will return to this topic in chapter 10.

5.3 Developed code

In this section, we are going to describe the code we developed to implement the firmware of the datalogger. We will refer to Node folder, that is, the root folder of our Code Composer Studio project (§A.1.5). Actually, part of the code will be discussed in chapter 8. The reader will understand the reason by the end of the section.

The root folder contains four main sub-folders:

- **driverlib**: it was added when we created the project from scratch. Our goal was to use the MSP430 Peripheral Diver Library by TI we have introduced in the previous section. Developers should never touch or change its contents;
- **fram-utilities**: it contains the FRAM Utilities by TI, a collection of embedded software utilities that leverage the ultra-low power and virtually unlimited write endurance of FRAM. In particular, we took into account the Compute Through Power Loss (CTPL), a utility API that enables ease of use with LPMx.5 *low-power modes* and a powerful *shutdown mode* that allows an application to *save* and *restore* critical system components when a *power loss* is detected [22]. This topic will discussed in chapter 8;
- **application**: it contains the file units we developed to implement the firmware. Its contents will be discussed in this section;
- **html**: it contains the files created by **Doxygen** tool with the aim of documenting the project. This topic will be discussed in section 5.4.

The other folders were managed by Code Composer Studio. The root folder also contains the main.c and main.h files, we are going to discuss later on; moreover, it contains the ctpl_lnk_msp430fr5994.cmd, ctpl_msp430fr5994.c and ctpl_pre_init.c files that will be presented in chapter 8.

In the following, we will describe the code by construction with a *bottom-up* approach. We point out that we developed a **layered code** whose structure is shown in figure 5.2. Each layer has a particular role, although the *separation of concerns* was not always easy to achieve.

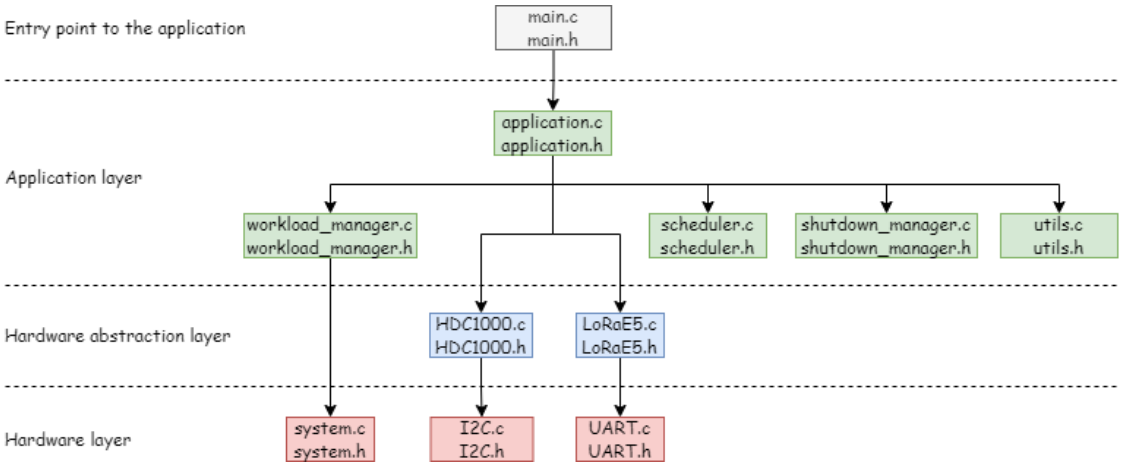


Figure 5.2: Structure of the layered code.

5.3.1 Hardware layer

The **hardware layer** sets up the MCU's hardware. Its role is crucial as wrong configurations deeply impact on the overall behavior of the microcontroller, especially when dealing with *clock sources*.

Initialization

The system.c and system.h files supervise the hardware initialization of the MCU. The System_init function performs the initialization of peripherals with a particular sequence:

```
1  void System_init(uint32_t frequency)
2  {
3      /* Halt the watchdog timer */
4      WDT_A_hold(WDT_A_BASE);
5
6      /* Initialize GPIO */
7      System_initGPIO();
8
9      /* Initialize clock sources and clocks */
10     System_initClocks(frequency);
11
12     /* Initialize timers */
13     System_initTimers();
14
15     /* Initialize ADC monitor */
16     System_initADCMonitor(frequency);
17
```

```
18      /* Initialize I2C */
19      System_initI2C(frequency);
20
21      /* Initialize UART */
22      System_initUART(frequency);
23
24      /* Enable general interrupts */
25      __enable_interrupt();
26  }
```

Listing 5.4: MCU: hardawre initialization.

As a very first step, we configured the *watchdog timer*. The **watchdog timer** is usually used to trigger a system reset process if the program is in a certain hardware or software failure condition [3]. We stopped or halted the watchdog timer, but we will return to this topic in chapter 8.

Then, we initialized the **general purpose input/output (GPIO)** thanks to the System_initGPIO function. The GPIO is a peripheral that can be used for interfacing to the outside world. As we saw in section 4.2.5, the target MCU has 80 pins many of which are connected to internal GPIO circuits. Furthermore, most of them may offer different other functions [3]. The MSP430FR5994 has eight digital I/O ports implemented, **P1** to **P8**. A further port exists called **PJ**. Most ports contain eight I/O lines; however, some ports may contain less. In particular, PJ contains only four I/O lines. Port pairs P1/P2, P3/P4, P5/P6 and P7/P8, are associated with the names **PA**, **PB**, **PC**, **PD** respectively. This means that individual ports can be accessed as byte-wide ports or can be combined into word-wide ports and accessed via word formats. Each I/O line is individually configurable for input or output direction as well as each can be individually read or written. Moreover, each I/O line is individually configurable for pullup or pulldown resistors [21]. The Driver Library provides several macros and a set of functions to deal with GPIO. First of all, we configured all the **pins as output** on a selected port. As an example:

```
1 GPIO_setAsOutputPin(GPIO_PORT_P1, GPIO_PIN_ALL8);
```

A good practice is to set all outputs to low value in order to **minimize the power consumption**:

```
1 GPIO_setOutputLowOnPin(GPIO_PORT_P1, GPIO_PIN_ALL8);
```

Of course, we set up some **pins as input**, that is:

- P4.1 - connected to Status[0] pin of the energy harvester module (see section 10.1);
- P4.2 - connected to Status[1] pin of the energy harvester module (see section 4.6);
- P4.3 - connected to Status[2] pin of the energy harvester module (see section 10.1);
- P6.3 - connected to DRDYn pin of the sensor module (see section 4.3).

Each pin required to be set as input, select the interrupt edge and clear the interrupt. Furthermore, the interrupt should be enabled. The following code reports the configuration for P4.2:

```
GPIO_setAsInputPin(GPIO_PORT_P4, GPIO_PIN2);
GPIO_selectInterruptEdge(GPIO_PORT_P4, GPIO_PIN2,
    GPIO_LOW_TO_HIGH_TRANSITION);
GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN2);
GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN2);
```

P6.3 also required to be set as input with **pull-up resistor**. The value of the internal resistor ranges from 20 kΩ to 50 kΩ (typically 35 kΩ):

```
GPIO_setAsInputPinWithPullUpResistor(GPIO_PORT_P6, GPIO_PIN3);
```

Finally, we performed an important configuration. When the MCU starts up, I/O pins might remained locked. To be effective for the port register configuration, the LOCKLPM5 bit needs to be cleared in PM5CTL0 register [3]. The PMM_unlockLPM5 library function performs this configuration behind the scenes:

```
PM5CTL0 &= ~LOCKLPM5;
```

The crucial step was to configure **clock sources** and **clock signals** thanks to the System_initClocks function. The MSP430FR5994 MCU has a complex, but straightforward clock system as you can see in figure 5.3. According to the documentation,

> "the clock module supports *low system cost* and *low power consumption*. Using three system clock signals, the user can select the best balance of performance and power consumption. The clock module can be configured to operate without any external components, with one or two external crystals, or with resonators, under full software control"[21].

The MSP-EXP430FR5994 LaunchPad™ Development Kit board mounts a 32-KHz Epson crystal (FC-135R) working as LFXTCLK source. The crystal allows for low power LPM3 sleep currents than do the other low-frequency clock sources. Therefore, the presence of the crystal allows the full range of *low-power modes* to be used [22]. We did not use this clock source, however the user should:

- set PJ.4 and PJ.5 as *Primary Module Function Input*:

  ```
  GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_PJ, GPIO_PIN4 +
      GPIO_PIN5, GPIO_PRIMARY_MODULE_FUNCTION);
  ```

- set the external clock frequency to 32.768 kHz:

  ```
  CS_setExternalClockSource(32768, 0);
  ```

- turn on the circuit:

  ```
  CS_turnOnLFXT(CS_LFXT_DRIVE_3);
  ```

We also did not take into account an external high-frequency oscillator, HFXTCLK source, as it was not mounted on board by default. This was not a problem as we exploited the **internal digitally controlled oscillator (DCO)** with selectable frequencies. Our solution was to set the DCOCLK frequency to **16 MHz**:

```
CS_setDCOFreq(CS_DCORSEL_1, CS_DCOFSEL_4);
```

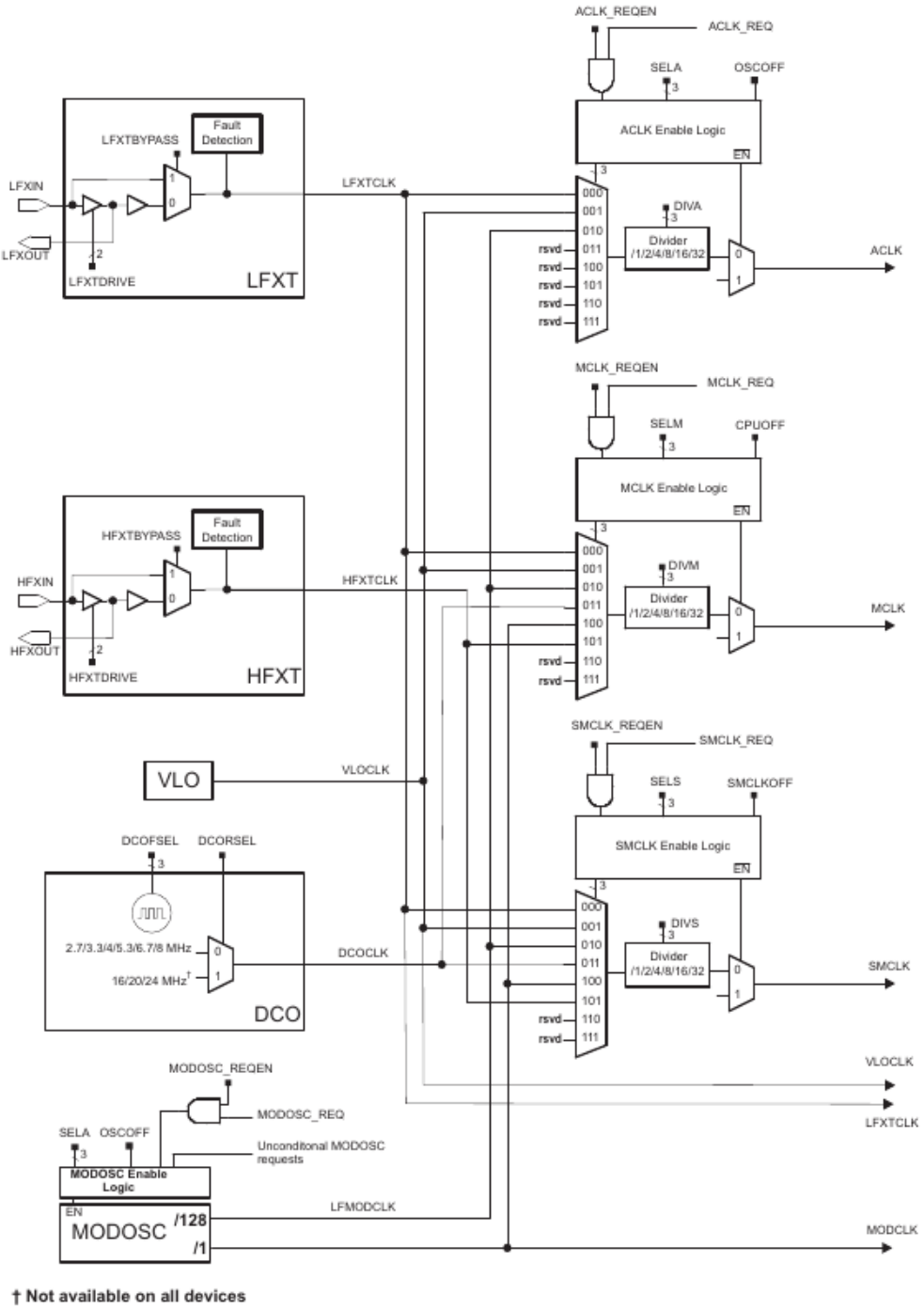


Figure 5.3: MSP430FR5994 Clock System [21].

Moreover, we did not take into account the internal low-power oscillator LFMODCLK, but we used the **internal very-low-power low-frequency (VLOCLK)** oscillator as well

see briefly. After setting the clock sources, we moved to clock signals. We mainly focused on the **master clock (MCLK)**, **sub-system master clock (SMCLK)** and **auxiliary clock (ACLK)**. The MCLK is used by the CPU and system, whereas SMCLK and ACLK are software selectable by individual peripheral modules. Here is a crucial point. Before configuring the clock signals, the user should configure one FRAM wait state for MCLK operation beyond 8 MHz as required by the datasheet. That's why we introduced the following code:

```
if (frequency >= SYSTEM_FREQUENCY_8MHZ) {
    FRCTL0 = FRCTLPW | NWAITS_1;
}
```

Our main goal was to set MCLK and SMCLK clock signals with different selected frequencies as we will see in chapter 8. Selected frequencies were **16**, **8**, **4** and **1 MHz**. As an example, to set these signals to 1 MHz, the following code has to be executed:

```
case SYSTEM_FREQUENCY_1MHZ:
    // Initialize master clock (MCLK=DCO)
    CS_initClockSignal(CS_MCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_16);
    // Initialize subsystem master clock (SMCLK=DCO)
    CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_16);
    break;
```

Finally, we set ACLK to be sourced by the internal very-low-power low-frequency VLOCLK source thanks to the following code:

```
/* Set auxiliary clock (ACLK=VLOCLK) */
CS_initClockSignal(CS_ACLK, CS_VLOCLK_SELECT, CS_CLOCK_DIVIDER_1);
```

The next step was to configure the **Timer_A1 (TA1)** peripheral thanks to the System_initTimers function. Main goal of the timer was to trigger a new measure at regular intervals. Our choice was to set TA1 in **continuous mode** sourced by SMCLK. Here is the code:

```
Timer_A_clearTimerInterrupt(TIMER_A1_BASE);
Timer_A_initContinuousModeParam param = {0};
param.clockSource = TIMER_A_CLOCKSOURCE_SMCLK;
param.clockSourceDivider = TIMER_A_CLOCKSOURCE_DIVIDER_1;
param.timerInterruptEnable_TAIE = TIMER_A_TAIE_INTERRUPT_ENABLE;
param.timerClear = TIMER_A_DO_CLEAR;
param.startTimer = false;
Timer_A_initContinuousMode(TIMER_A1_BASE, &param);
```

The library defines a Timer_A_initContinuousModeParam structure whose role is to configure all the necessary parameters. We do not want to enter the details, but the reader may inspect the datasheet of the MCU and the library documentation to easily understand how to configure a timer in continuous mode. A remark: the function just initializes the timer; it does not start or stop it. The timer is started or stopped by the *workload manager* as the reader will understand in chapter 8.

Another important step was to configure the **ADC12_B** peripheral whose role was crucial to save and restore the *context*. We prefer to postpone the discussion in chapter 8 to have a linear and clear explanation.

The System_init function performs other two important initializations we needed in our project. The System_initI2C function allows the initialization of the eUSCI_B2 peripheral

thanks to which we targeted an **I2C** interface. This topic will be discussed in the next section. The System_initUART function allows the initialization of the eUSCI_A3 peripheral thanks to which we targeted an **UART** interface. This topic will be discussed in section 5.3.1.

The very last initialization step was to enable the **general global interrupt** that could be easily done thanks to the following code:

```
__enable_interrupt();
```

I2C

The I2C.c and I2C.h files target an I2C interface of the MCU. As we saw in section 4.3.1, the temperature and humidity sensor measurement results can be read out through its I2C compatible interface. The hardware layer here has the role of initializing a selected I2C peripheral, that is **eUSCI_B2**, for this purpose.

Above all, we configured P7.0 and P7.1 **pins** thanks to the I2C_initGPIO function. Indeed, P7.0 should work as I2C **SDA**, whereas P7.1 should work as I2C **SCL**. Such configurations were straightforward:

```
/* Select I2C function for I2C_SCL */
GPIO_setAsPeripheralModuleFunctionOutputPin(I2C_SCL_PORT, I2C_SCL_PIN,
    I2C_SELECT_FUNCTION);

/* Select I2C function for I2C_SDA */
GPIO_setAsPeripheralModuleFunctionOutputPin(I2C_SDA_PORT, I2C_SDA_PIN,
    I2C_SELECT_FUNCTION);
```

We defined the macros into the I2C.h file.

The next step was to define the I2C interface **parameters**. We easily set the parameters thanks to the EUSCI_B_I2C_initMasterParam structure. Let us give an example taken from the code:

```
/*
 * I2C Master Configuration Parameter
 * SMCLK frequency: 1MHz
 */
EUSCI_B_I2C_initMasterParam i2cParam_1M =
{
    EUSCI_B_I2C_CLOCKSOURCE_SMCLK,          // SMCLK Clock Source
    1000000,                                // SMCLK = 1MHz
    EUSCI_B_I2C_SET_DATA_RATE_100KBPS,      // Desired I2C Clock of 100khz
    0,                                      // No byte counter threshold
    EUSCI_B_I2C_NO_AUTO_STOP                // No Autostop
};
```

Listing 5.5: I2C: master configuration parameters.

The user must select a clock source. We selected **SMCLK** as clock source, but another possible solution could be ACLK. The second parameter defines the rate of the clock supplied to the I2C module (the frequency in Hz of the clock source specified in the selected clock source), whereas the third one sets the desired I2C data transfer rate. The last two parameters set threshold for automatic STOP and the STOP condition generation respectively. We effectively set parameters thanks to the I2C_initMasterParam function. The idea

was to initialize the interface by using different frequencies, that's why we prepared some configurations among which to choose. The implementation was straightforward:

```
void I2C_initMasterParam(int frequency)
{
    switch (frequency) {
    case SYSTEM_FREQUENCY_16MHZ:
        initMasterParam = i2cParam_16M;
        break;
    case SYSTEM_FREQUENCY_8MHZ:
        initMasterParam = i2cParam_8M;
        break;
    case SYSTEM_FREQUENCY_4MHZ:
        initMasterParam = i2cParam_4M;
        break;
    case SYSTEM_FREQUENCY_1MHZ:
        initMasterParam = i2cParam_1M;
        break;
    default:
        initMasterParam = i2cParam_1M;
        break;
    }
}
```

The I2C_init() function has a crucial role. It must be called anytime the interface has to be used:

```
void I2C_init()
{
    /* Initialize USCI_B2 and I2C Master to communicate with slaves */
    EUSCI_B_I2C_initMaster(I2C_BASE, &initMasterParam);

    /* Disable I2C module to make changes */
    EUSCI_B_I2C_disable(I2C_BASE);

    EUSCI_B_I2C_enable(I2C_BASE); /* Enable I2C Module */
}
```

The EUSCI_B_I2C_initMaster library function takes the *base address* of EUSCI_B2 interface and I2C parameters as parameters. Its goal is to initialize USCI_B2 and I2C master to communicate with slave devices. Then, the I2C module is disabled to make changes and then enabled to start operations. Now the reader should have understood the role of the System_initI2C function into the system.c file:

```
static void System_initI2C(int frequency)
{
    /* Set I2C SDA and SCL */
    I2C_initGPIO();

    /* Set parameters */
    I2C_initMasterParam(frequency);

    /* Initialize I2C */
    I2C_init();
}
```

Listing 5.6: I2C: initialization.

It implements exactly the set of steps needed to correctly initialize and use the selected I2C interface.

The I2C_setslave function has the role of specifying the slave address for I2C and enabling and clearing the interrupt flag:

```
void I2C_setslave(unsigned short slaveAddress)
{
    /* Specify slave address for I2C */
    EUSCI_B_I2C_setSlaveAddress(I2C_BASE, slaveAddress);

    /* Enable and clear the interrupt flag */
    EUSCI_B_I2C_clearInterrupt(I2C_BASE, EUSCI_B_I2C_TRANSMIT_INTERRUPT0
    + EUSCI_B_I2C_RECEIVE_INTERRUPT0);
}
```

This function is necessary to correctly perform an I2C communication, as part of the protocol as we will see later on.

The I2C.c file also implements further base functions whose role will be clearer in section 5.3.2. As an example, let use take into account the I2C_write8 function:

```
int I2C_write8(uint8_t reg, uint32_t timeout)
{
    /* Set master to transmit mode PL */
    EUSCI_B_I2C_setMode(I2C_BASE, EUSCI_B_I2C_TRANSMIT_MODE);

    /* Clear any existing interrupt flag PL */
    EUSCI_B_I2C_clearInterrupt(I2C_BASE, EUSCI_B_I2C_TRANSMIT_INTERRUPT0)
    ;

    /* Send a single byte with timeout */
    if (!EUSCI_B_I2C_masterSendSingleByteWithTimeout(I2C_BASE, reg,
    timeout)) {
        return 0;
    }

    return 1;
}
```

After setting the master to *transmit mode*, any existing interrupt flag is cleared. The EUSCI_B_I2C_masterSendSingleByteWithTimeout function performs single byte transmission from master to slave with timeout. In particular, behind the scenes, it sends a start, then transmits the byte to the slave and then sends a stop. The *timeout* is the amount of time to wait until giving up. The reader might remember the I2C_BASE parameter. This macro represents the base address of the target interface: the EUSCI_B2 module.

UART

The UART.c and UART.h files target an UART interface of the MCU. As we saw in section 4.4.1, the LoRaWAN® transceiver can be managed thanks to its UART compatible interface. The hardware layer here has the role of configuring a selected UART peripheral of the MCU, that is **eUSCI_A3**, for this purpose.

We implemented the UART_initGPIO function to configure P6.0 and P6.1 **pins**. In particular, P6.0 is configured to transmit data (**TX**), whereas P6.1 is configured to receive

data (**RX**). Here is the code:

```
/* Select UART functions for TXD */
GPIO_setAsPeripheralModuleFunctionInputPin(UART_TXD_PORT, UART_TXD_PIN,
    UART_SELECT_FUNCTION);

/* Select UART functions for RXD */
GPIO_setAsPeripheralModuleFunctionInputPin(UART_RXD_PORT, UART_RXD_PIN,
    UART_SELECT_FUNCTION);
```

The next step was to define the UART interface **parameters**. The configuration could be easily performed thanks to the EUSCI_A_UART_iniParam structure. Let us give an example taken from the code:

```
/**
 * @brief UART Configuration Parameter.
 * SMCLK frequency: 1MHz
 * Baud rate: 57600bps
 */
static const EUSCI_A_UART_initParam uartParam_1M =
{
    EUSCI_A_UART_CLOCKSOURCE_SMCLK,                // SMCLK Clock Source
    17,                                            // BRDIV = 17
    0,                                             // UCxBRF = 0
    74,                                            // UCxBRS = 74
    EUSCI_A_UART_NO_PARITY,                        // No Parity
    EUSCI_A_UART_LSB_FIRST,                        // LSB First
    EUSCI_A_UART_ONE_STOP_BIT,                     // One stop bit
    EUSCI_A_UART_MODE,                             // UART mode
    EUSCI_A_UART_LOW_FREQUENCY_BAUDRATE_GENERATION // Over sampling
};
```

Listing 5.7: UART: configuration parameters.

The user must select a clock source. We selected SMCLK as clock source, but another possible solution could be ACLK. The second parameter sets the clock prescalar. The third and forth parameters define the first and the second modulation stage register. All these values are pre-calculated and can be obtained from the datasheet of the MCU. However, we exploited an online calculator by TI to configure parameters to make the eUSCI_A_UART module to operate at **57600 bps**. The code reports how to reach the online calculator. The others are well-known usual parameter settings such as the desired parity, the direction of receive and transmit shift register, the number of stop bits and the mode of operation. The last parameter indicates low frequency or oversampling baud generation.

We effectively set the parameters thanks to the UART_initUartParam function. The idea was to initialize the interface by using different frequencies, that's why we prepared some configurations among which to choose. The implementation was straightforward:

```
void UART_initUartParam(int frequency)
{
    switch (frequency) {

    case SYSTEM_FREQUENCY_16MHZ:
        initParam = uartParam_16M;
        break;
    case SYSTEM_FREQUENCY_8MHZ:
```

```
        initParam = uartParam_8M;
        break;
    case SYSTEM_FREQUENCY_4MHZ:
        initParam = uartParam_4M;
        break;
    case SYSTEM_FREQUENCY_1MHZ:
        initParam = uartParam_1M;
        break;
    default:
        initParam = uartParam_16M;
        break;
  }
}
```

The UART_init() function has a crucial role. This function must be called to initialize the interface:

```
void UART_init()
{
    /* Configure UART Module */
    EUSCI_A_UART_init(UART_BASE, &initParam);

    /* Enable UART module */
    EUSCI_A_UART_enable(UART_BASE);

    /* Clear interrupt */
    EUSCI_A_UART_clearInterrupt(UART_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT
    );

    /* Enable USCI_A3 RX interrupt */
    EUSCI_A_UART_enableInterrupt(UART_BASE,
    EUSCI_A_UART_RECEIVE_INTERRUPT);
}
```

The EUSCI_A_UART_init function takes the base address of EUSCI_A3 interface and UART parameters as parameters. Its goal is to initialize USCI_A3 to communicate. Then, the UART module is enabled to make changes. Finally, the interrupt flag is cleared and enabled. Now the reader should have understood the role of the System_initUART function into the system.c file:

```
static void System_initUART(int frequency)
{
    /* Set UART TXD and RXD */
    UART_initGPIO();

    /* Set parameters */
    UART_initUartParam(frequency);

    /* Initialize UART */
    UART_init();
}
```

Listing 5.8: UART: initialization.

It implements exactly the set of steps needed to correctly initialize and use the selected UART interface.

To transmit a string over the UART interface, we implemented the UART_transmit function:

```
void UART_transmit(char *pStr)
{
    rxdata = 0;
    UART_RESPONSE_READY = FALSE;
    while (*pStr) {
        EUSCI_A_UART_transmitData(UART_BASE, *pStr);
        pStr++;
    }
}
```

After initializing the rxdata variable, the UART_RESPONSE_READY flag is set to FALSE. The flag has a relevant role as it controls the interrupt-base communication. The code shows that the function returns after sending all characters of the string to transmit. The EUSCI_A_UART_transmitData library function transmits a byte from the UART module.

The UART_receive function tests the UART_RESPONSE_READY flag:

```
char *UART_receive(void)
{
    if (UART_RESPONSE_READY) {
        return response;
    } else {
        return NULL;
    }
}
```

The role of this function will be clearer in section 5.3.2.

The last point is about the USCI_A3 **interrupt vector service routine (ISR)**. Here is its implementation:

```
#pragma vector=USCI_A3_VECTOR
__interrupt void USCI_A3_ISR(void)
{
    switch(__even_in_range(UCA3IV,USCI_UART_UCTXCPTIFG)) {

    /* USCI_NONE */
    case USCI_NONE:
        break;

    /* EUSCI_A_UART_RECEIVE_INTERRUPT_FLAG */
    case USCI_UART_UCRXIFG:
        if((response[rxdata]=EUSCI_A_UART_receiveData(UART_BASE))=='\n'){
            /* NULL terminate the string */
            response[rxdata++] = '\0';
            /* Set response flag */
            UART_RESPONSE_READY = TRUE;
        } else {
            rxdata++;
        }
        break;

    /* EUSCI_A_UART_TRANSMIT_INTERRUPT_FLAG */
    case USCI_UART_UCTXIFG:
        break;
```

```
25
26      /* EUSCI_A_UART_STARTBIT_INTERRUPT_FLAG */
27      case USCI_UART_UCSTTIFG:
28          break;
29
30      /* EUSCI_A_UART_TRANSMIT_COMPLETE_INTERRUPT_FLAG */
31      case USCI_UART_UCTXCPTIFG:
32          break;
33      }
34  }
```

Listing 5.9: UART: ISR.

We were interested in the USCI_UART_UCRXIFG receive interrupt flag. The response vector collects each character read from the UART interface after an interrupt. If the **\n** character is received, the response is ended with a **\0** character in order to have a well-formed string; the UART_RESPONSE_READY flag is set to TRUE. Otherwise, the rxdata variable is increased. Finally, the ISR returns.

We point out an important consideration before leaving the hardware layer section. The hardware layer was used to directly initialize, set up and control the hardware of the MCU. The *low level* functions were implemented by taking into account the target modules of the MCU that were relevant for our project. We were interested in calling these functions from higher layers according to application needs. This topic will be more clearer shortly.

5.3.2 Hardware abstraction layer

Now, we are going to discuss how we interfaced with the hardware layer through the **hardware abstraction layer (HAL)**. Its role is to create a separation between the hardware and application logic.

HDC1000

In section 4.3, we presented the module we used to implement the sensing sub-system. It is designed around the HDC1000 integrated circuit.

The HDC1000.c and HDC1000.h files deal with the module. They work as hardware abstraction layer units resorting to the I2C interface which is part of the hardware layer. We deeply investigate the HDC1000 datasheet with the aim of retrieving all the information we needed for our project.

The HDC1000.h header file has a key role. First of all, we created a **structure** capable of describing and storing all the available HDC1000 sensor data:

```
1  /* HDC1000 sensor data structure */
2  typedef struct HDC1000_ {
3      uint16_t temperature;
4      uint16_t humidity;
5      uint16_t configuration;
6      uint64_t serialID;
7      uint16_t manufacturerID;
8      uint16_t deviceID;
9  } HDC1000;
```

The header file also contains a macro describing the HDC1000 **slave address byte**:

```
#define HDC1000_SLAVE_ADDRESS          0x40      // ADR1=0, ADR0=0
```

Indeed, to communicate with the HDC1000, the master must first address the device via a slave address byte. The slave address byte consists of seven address bits and a direction bit that indicates the intent to execute a read or write operation. The HDC1000 features two address pins to allow up to 4 devices to be addressed on a single bus. The state of the ADR0 and ADR1 pins is sampled on every bus communication and should be set before any activity on the interface occurs. The address pin is read at the start of each communication event. It is not allowed to have on the I2C bus multiple devices with the same address. The HDC1000 contains **data registers** that hold configuration information, temperature and humidity measurement results, and status information. The datasheet reports that registers from 0x03 to 0xFA are reserved and should not be written [17]. We defined the macros describing the **register map**:

```
/* Registers */
#define HDC1000_TEMPERATURE     0x00 // Temperature measurement
#define HDC1000_HUMIDITY        0x01 // Relative Humidity measurement
#define HDC1000_CONFIGURATION   0x02 // HDC1000 configuration and status
#define HDC1000_SERIAL_ID_F     0xFB // First 2 bytes of the serial ID
#define HDC1000_SERIAL_ID_M     0xFC // Mid 2 bytes of the serial ID
#define HDC1000_SERIAL_ID_L     0xFD // Last byte bit of the serial ID
#define HDC1000_MANUFACTURER_ID 0xFE // ID of Texas Instruments
#define HDC1000_DEVICE_ID       0xFF // ID of HDC1000 device
```

In practice, the HDC1000 has an 8-bit pointer used to address a given data register. The pointer identifies which of the data registers should respond to a read or write command on the two-wire bus. This register is set with every write command. A write command must be issued to set the proper value in the pointer before executing a read command. The power-on reset (POR) value of the pointer is 0x00, which selects a temperature measurement [17]. Other important macros related to the HDC1000 **measurement configuration**:

```
  /*
  * ==============================================================
  * HDC1000 measurement configuration
  * ==============================================================
  *
  * NAME        REGISTERS                 DESCRIPTION
  *
  * RST         [15] Software reset       0  Normal Operation,
  *                                       1  Software Reset
  *
  * Reserved    [14] Reserved             0  Reserved, must be 0
  *
  * HEAT        [13] Heater               0  Heater Disabled
  *                                       1  Heater Enabled
  *
  * MODE        [12] Mode of acquisition  0  T or H is acquired.
  *                                       1  T and H are acquired
  *
  * BTST        [11] Battery Status       0  Battery voltage > 2.8V
  *                                       1  Battery voltage < 2.8V
  *
```

```
  * TRES        [10] Temperature           0  14 bit
  *                  Measurement           1  11 bit
  *                  Resolution
  *
  * HRES       [9:8] Humidity              00 14 bit
  *                  Measurement           01 11 bit
  *                  Resolution            10  8 bit
  * Reserved   [7:0] Reserved              0  Reserved, must be 0
  * ===============================================================
  */
#define CONFIG_MODE0  0x0000   // T or H is acquired
#define CONFIG_MODE1  0x1000   // T and H are acquired
```

The **configuration word** involves 16 bits as you can see by analyzing the comments. We were simply interested in having 14 bits temperature and humidity measurement resolution. The difference is in the **mode of acquisition**. If bit[12] is set to 0, temperature or humidity are acquired, otherwise temperature and humidity are acquired in sequence (temperature first). We did not take into account the other features provided by the IC.

The HDC1000.c file implements the abstraction layer. After declaring an HDC1000 structure variable:

```
static HDC1000 *HDC1000data;
```

we declared a set of useful **flags**:

```
static int temperature_triggered = 0;
static int humidity_triggered = 0;
static int temperature_and_humidity_triggered = 0;
```

whose role will be described shortly.

Les us start by describing how we performed the **sensor initialization** thanks to the HDC1000_init function. Here is the code:

```
int HDC1000_init(uint16_t configuration)
{
    /* Create the data structure */
    HDC1000data = (HDC1000*) malloc(sizeof(HDC1000));

    /* Set the initial configuration */
    HDC1000_setConfiguration(configuration);

    /* Read the serial ID */
    HDC1000_readSerialID();

    /* Read the manufacturer ID */
    HDC1000_readManufacturerID();

    /* Read the device ID */
    HDC1000_readDeviceID();

    return 1;
}
```

The function takes the configuration word as input. After creating the data structure by allocating the necessary memory space, the initial configuration is performed. This activity is described in figure 5.4. The datasheet explains how to correctly execute a **write**

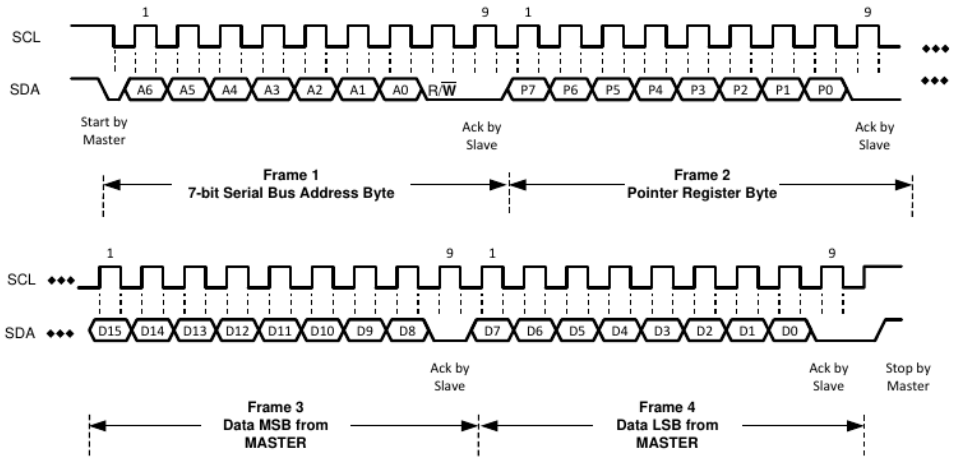


Figure 5.4: HDC1000 configuration [17].

operation to the sensor. In practice, the master has to select the target slave (frame 1), send the configuration register pointer 0x02 (frame 2), send MSB data (frame 3) and finally send LSB data (frame 4). From the code point of view the hardware abstraction layer performs the configuration thanks to the HDC1000_setConfiguration function:

```
int HDC1000_setConfiguration(uint16_t configuration)
{
    /* Write configuration */
    if (!HDC1000_writeConfiguration(configuration)) {
        return 0;
    }

    /* Save configuration */
    HDC1000data->configuration = configuration;

    return 1;
}
```

The HDC1000_writeConfiguration function performs the necessary steps:

```
static int HDC1000_writeConfiguration(uint16_t configuration)
{
    /* Configure I2C */
    I2C_init();

    /* Specify the slave address for HDC1000 */
    I2C_setslave(HDC1000_SLAVE_ADDRESS);

    /* Write the configuration */
    if (!I2C_write24(HDC1000_CONFIGURATION, swapMSB_LSB(configuration),
    TIMEOUT)) {
        return 0;
    }
    return 1;
}
```

After initializing the I2C interface, the slave address for HDC1000 is specified. At this point the hardware layer I2C_write24 function is called. The name of this function might confuse the reader. Actually, the written bytes are 4 and not 3, as the function also send the slave address under the scenes. The number 24 is the number of bits, but the actual name should be I2C_write32. We preferred to use this name to take into account the *invisible* operation performed by the library functions. More in detail, let us analyze the implementation of the I2C_write24 function:

```
int I2C_write24(uint8_t reg, uint16_t data, uint32_t timeout)
{
    /* Set master to transmit mode PL */
    EUSCI_B_I2C_setMode(I2C_BASE, EUSCI_B_I2C_TRANSMIT_MODE);

    /* Clear any existing interrupt flag PL */
    EUSCI_B_I2C_clearInterrupt(I2C_BASE, EUSCI_B_I2C_TRANSMIT_INTERRUPT0)
    ;

    /* Initiate start and send first byte with timeout */
    if (!EUSCI_B_I2C_masterSendMultiByteStartWithTimeout(I2C_BASE, reg,
    timeout)) {
        return 0;
    }

    /* Send the second byte (MSB of data) with timeout */
    if (!EUSCI_B_I2C_masterSendMultiByteNextWithTimeout(I2C_BASE, (
    unsigned char) (data & 0xFF), timeout)) {
        return 0;
    }

    /* Send the third byte (LSB of data) and stop with timeout */
    if (!EUSCI_B_I2C_masterSendMultiByteFinishWithTimeout(I2C_BASE, (
    unsigned char) (data >> 8), timeout)) {
        return 0;
    }

    return 1;
}
```

After setting the master to transmit and clearing any existing interrupt flag, the EUSCI_B_I2C_masterSendMultiByteStartWithTimeout is called. The role of this function is to start a multi-byte transmission from master to slave with timeout:

```
extern bool EUSCI_B_I2C_masterSendMultiByteStartWithTimeout(
uint16_t baseAddress,
uint8_t txData,
uint32_t timeout);
```

The function will send the slave address before the txData byte. This means that two bytes are sent by the function. Then, the second byte (MSB of data) and the third byte (LSB of data) with stop are sent to complete the transmission. We hope the reader got the mechanism. It is not so complicated, but we tried to respect the separation of concerns as much as possible. After configuring the sensor, we read the *device ID*, *manufacturer ID* and *device ID* to complete the initialization.

We can discuss how to **trigger a measure** by referring to figure 5.5. After selecting

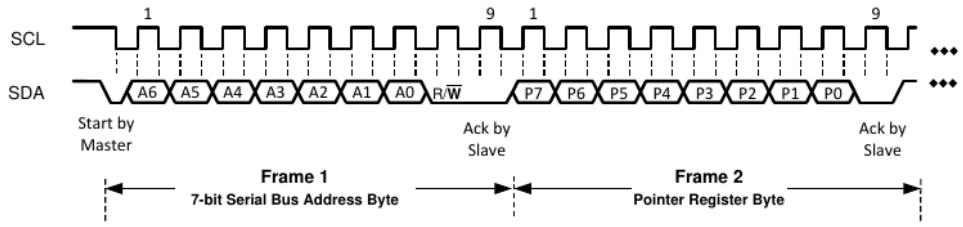


Figure 5.5: HDC1000 trigger humidity/temperature measurement [17].

the slave, the user has to trigger the measurement by executing a pointer write operation. In particular:

- set the address pointer to 0x00 for a temperature measurement;
- set the address pointer to 0x01 for a humidity measurement.

After triggering, the user must wait for the measurement to complete, based on the conversion time. Alternatively, wait for the assertion of DRDYn signal [17]. We followed the latter approach. From the code point of view, we wrote the HDC1000_trigger function:

```
int HDC1000_trigger(void)
{
    /* Check the acquisition mode of configuration register bit [12] */
    if (HDC1000data->configuration & 0x1000) {
        /* Read temperature and humidity in sequence */
        if (!HDC1000_readTemperatureAndHumidity()) {
            return 0;
        }
        /* Set the flag */
        temperature_and_humidity_triggered = 1;
    } else {
        /* Read temperature */
        if (!HDC1000_readTemperature()) {
            return 0;
        }
        /* Set the flag */
        temperature_triggered = 1;
        /* Read humidity */
        if (!HDC1000_readHumidity()) {
            return 0;
        }
        /* Set the flag */
        humidity_triggered = 1;
    }

    return 1;
}
```

Basically, we check the acquisition mode of the configuration register to understand how to trigger the measure. In case its value is 0x1000, temperature and humidity are read in sequence in a single operation (temperature first), otherwise we have to trigger the temperature first and humidity after. Finally, the relative flag is set. Let us analyze how

we triggered the temperature, as an example, thanks to the HDC1000_readTemperature function:

```
static int HDC1000_readTemperature()
{
    /* Configures I2C */
    I2C_init();

    /* Specify the slave address for HDC1000 */
    I2C_setslave(HDC1000_SLAVE_ADDRESS);

    /* Trigger the temperature register */
    if (!I2C_write8(HDC1000_TEMPERATURE, TIMEOUT)) {
        return 0;
    }

    return 1;
}
```

After initializing the I2C interface and specifying the slave address for HDC1000, the temperature register is triggered. The I2C_write8 function exactly performs the operation shown in figure 5.5. The HDC1000_TEMPERATURE macro represents the 0x00 temperature pointer register. As we mentioned previously, after triggering a measure, we had to wait for the assertion of DRDYn by the sensor. This interrupt is managed by the application as we will see in chapter 8.

We can now explain how we read output data to have an example of **read operation** from the sensor. We implemented the HDC1000_acquire function:

```
int HDC1000_acquire()
{
    /* Configures I2C */
    I2C_init();

    /* Specify the slave address for HDC1000 */
    I2C_setslave(HDC1000_SLAVE_ADDRESS);

    /* Check the flags */
    if (temperature_triggered) {
        /* Reset the flag */
        temperature_triggered = 0;
        /* Read the temperature */
        if (!I2C_read16(&HDC1000data->temperature, TIMEOUT)) {
            return 0;
        }
    } else if (humidity_triggered) {

        /* Reset the flag */
        humidity_triggered = 0;

        /* Read the humidity */
        if (!I2C_read16(&HDC1000data->humidity, TIMEOUT)) {
            return 0;
        }
    } else if (temperature_and_humidity_triggered) {

```

```
28          /* Reset the flag */
29          temperature_and_humidity_triggered = 0;
30          /* Read the temperature and humidity */
31          if (!I2C_read32(&HDC1000data->temperature, &HDC1000data->humidity
    , TIMEOUT)) {
32              return 0;
33          }
34      }
35
36      return 1;
37  }
```

After initializing the I2C interface and specifying the slave, as usual, we check which flag is set by the trigger function. If the temperature_triggered flag is set, as an example, the flag is reset and the sampled temperature is acquired thanks to the I2C_read16 function. For the sake of completeness, figure 5.6 reports how to read temperature and humidity in a single transaction. This can be obviously done after triggering the temperature measurement and setting the MODE bit to 1 in the configuration register. The figure starts with the third

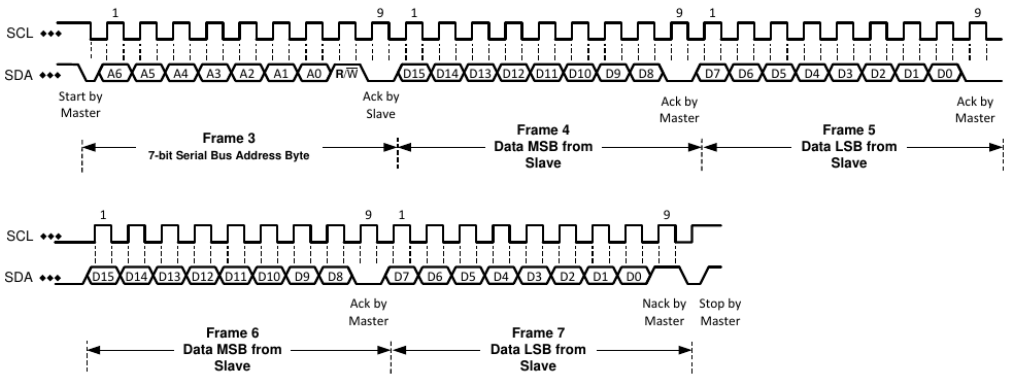


Figure 5.6: HDC1000 read humidity and temperature measure [17].

frame, because the first and second relate to the trigger operation.

We suggest the reader to inspect the HDC1000 datasheet to better understand how to configure the sensor as well as trigger and retrieve measures. However, our code is highly documented and can be used for this purpose. Moreover, we implemented other useful functions to deal with the target sensor, but the mechanism is similar to ones we have analyzed so far. By inspecting the code, the reader can easily compare the implementation with information taken from the datasheet.

LoRaE5

In section 4.4, we presented the module we used to implement the communication subsystem. Here, we are going to analyze the hardware abstraction layer, but the reader will know how the LoRaWAN® protocol works and *AT commands* in chapter 7. LoRaE5.c and LoRaE5.h files deal with the module.

From LoRa-E5 AT Command Specification datasheet we learned that the **command**

length never exceeds total 528 characters [41]. That's why we defined the following macro in the LoRaE5.h header file:

```
#define     COMMAND_LENGTH      528
```

Then, we defined the **operating modes**:

```
/* Operating modes */
#define     INIT_MODE           0
#define     TEST_MODE           1
#define     OTAA_MODE           2
```

Other useful macros related to the **logging** process:

```
/* Logging */
#define     LOG_LEVEL_DEBUG     0
#define     LOG_LEVEL_INFO      1
#define     LOG_LEVEL_WARN      2
#define     LOG_LEVEL_ERROR     3
#define     LOG_LEVEL_FATAL     4
#define     LOG_LEVEL_PANIC     5
#define     LOG_LEVEL_QUIET     2
```

and **error** handling (although useful for future work - §10.1):

```
/* Errors */
#define     ERROR_PARAMETER_INVALID             -1
#define     ERROR_COMMAND_UNKNOWN               -10
#define     ERROR_COMMAND_WRONG_FORMAT          -11
#define     ERROR_COMMAND_UNAVAILABLE           -12
#define     ERROR_PARAMETER_TOO_MANY            -20
#define     ERROR_COMMAND_LENGTH                -21
#define     ERROR_COMMAND_TIMEOUT               -22
#define     ERROR_INVALID_CHARACTER_RECEIVED    -23
#define     ERROR_24                            -24
```

The LoRaE5.c file implements the hardware abstraction layer. The first step was to introduce a useful variable to take trace of the current operating mode:

```
static int op_mode = 0;
```

as well as declare the isAlive and isReady flags.

```
static int isAlive = FALSE;
static int isReady = FALSE;
```

We also declared the command and response arrays to communicate with the transceiver module:

```
static char command[COMMAND_LENGTH];
static char response[COMMAND_LENGTH];
```

The LoRaE5_init function plays a crucial role. Here is its implementation:

```
int LoRaE5_init(int mode)
{
    isAlive = FALSE;
    isReady = FALSE;
    op_mode = INIT_MODE;

```

```
7     /* Test whether the connection of module is ok */
8     if (at_send_check_response("...")) {
9         isAlive = TRUE;
10        switch (mode) {
11
12            case TEST_MODE:
13
14            /* Enter TEST mode */
15            at_send_check_response("...");
16            /* Set RF Configuration */
17            at_send_check_response("...");
18
19            isReady = TRUE;
20            op_mode = TEST_MODE;
21            break;
22
23            case OTAA_MODE:
24
25            /* Enter OTAA mode */
26            at_send_check_response("...");
27            at_send_check_response("...");
28            at_send_check_response("...");
29            at_send_check_response("...");
30            at_send_check_response("...");
31            at_send_check_response("...");
32            at_send_check_response("...");
33
34            /* Join the LoRaWAN network */
35            if (at_send_check_response("...")) {
36                isReady = TRUE;
37            } else {
38                /* Failed joining the LoRaWAN network */
39                at_send_check_response("...");
40            }
41            op_mode = OTAA_MODE;
42            break;
43        }
44        /* Put the module in low-power mode **/
45        at_send_check_response("...");
46    } else {
47        return isAlive;
48    }
49    return isReady;
50 }
```

Listing 5.10: LoRaWAN® module initialization.

To make the code reading easier and focus on important aspects, we substituted real commands with three dots. The function performs a set of steps according to the *operating mode* passed as parameter. Actually, as a first step, the user must test whether the connection of module is OK. In case of success, the function sets the isAlive flag to TRUE and goes on with the initialization process, otherwise returns to the caller as the module is not available. After testing the availability, the function executes a sequence of commands according to the selected *operating mode*. The meaning of commands will be resumed in

section 7.3.3. At the end of the sequence, the isReady flag is set to TRUE as the initialization is successfully completed. The initialization process time execution depends on the selected *operating mode*, as we will see in chapter 7. Finally, before returning to the caller, the function puts the module in **low-power mode**. This setting significantly reduces the power consumption of the module as we will see in chapter 8.

The at_send_check_response function is responsible of sending a command as well as waiting for the response:

```
int at_send_check_response(char *p_ack, int timeout_ms, char *p_cmd)
{
    /* Check whether the expected response has been set */
    if (p_ack == NULL) {
      return 0;
    }

    /* Send command */
    if (strlen(p_cmd) <= COMMAND_LENGTH) {
      UART_transmit(p_cmd);
    }

    while (UART_receive() == NULL);

    /* Get the response */
    *response = strstr(UART_receive(), p_ack);

    /* Check the response */
    if (response != NULL) {
        return 1;
    }

    return 0;
}
```

The function takes the expected response, timeout and command as parameters. If the there is no expected response, the function returns as, according to the datasheet, each command has a response. After sending the command, the user must wait for its response. Here are two important aspects. First of all, you have to check whether the response matches the expected one to verify that the command is successfully executed. Furthermore, you must read the response before sending a new command. You will be certainly experiencing unexpected behaviors if such sequence is not respected. We might say this was a critical section of our implementation. In particular, a better solution should be taken into account with the aim of improving the response handling. The topic will be resumed in section 10.1.

We implemented a suitable function to send data over the LoRaWAN® network, that is, LoRaE5_sendData:

```
void LoRaE5_sendData(char *data)
{
    int ret = 0;

    if (isAlive && isReady) {

        /* Set command according to the current operating mode */
```

```
8         switch (op_mode) {
9
10            case TEST_MODE:
11            sprintf(command, "AT+TEST=TXLRPKT,\"%s\"\r\n", data);
12            break;
13
14            case OTAA_MODE:
15            sprintf(command, "AT+MSGHEX=\"%s\"\r\n", data);
16            break;
17        }
18
19        /* Wake up the module */
20        at_send_check_response("+LOWPOWER: WAKEUP", 1000, "A");
21
22        /* Wait */
23        __delay_cycles(2000);
24
25        /* Send command according to the current operating mode */
26        switch (op_mode) {
27
28            case TEST_MODE:
29            ret = at_send_check_response("TX DONE", 1000, command);
30            break;
31
32            case OTAA_MODE:
33            ret = at_send_check_response("Done", 1000, command);
34            break;
35        }
36    }
37
38    return ret;
39 }
```

Data are passed as parameter and the function creates the correct format according to the current *operating mode*. Before sending data, the module is waken up from SLEEP mode; that's why we introduced a suitable delay to let the module resuming correctly.

The LoRaWAN® protocol is large and thanks to *AT commands* the user might perform everything the application needs. In practice, the user should implement all the functions which can exploit the features of the target LoRaWAN® module. This was also our approach, but we preferred to implement and exploit a very low subset of all possible features of our module. The reader could have noticed that we did not exploit *logging* and *error* handling macros which would have obviously led to a more robust code. We decided to let such improvements for future work as we focused on a basic, but effective implementation strategy. Also this topic will be resumed in section 10.1.

5.3.3 Application layer

The **application layer** interfaces with the hardware abstraction layer and, sometimes, directly with the hardware layer. It means that, we could not always apply the separation of concerns correctly, although we did our best effort and achieved good results. However, the main goal of the application layer is to run *tasks* thanks to a *scheduler*. The application also applies some *policies* thanks to the *workload manager* and deal with the *intermittent*

behavior of the datalogger thanks to the CTPL library by TI. All these concepts will be discussed in chapter 8, whereas now we are going to present the application core.

Utilities

The utils.c and utils.h files implement general utility functions. The reader might inspect the files, but we report here the swapMSB_LSB function as an example:

```
int swapMSB_LSB(int data)
{
    int tmp = (unsigned char)data;
    tmp = tmp << 8;
    tmp += (unsigned char)(data >> 8);
    return tmp;
}
```

The function swaps the MSB and LSB of a word. It was used to write the configuration to HDC1000 sensor:

```
static int HDC1000_writeConfiguration(uint16_t configuration)
{
    /* Configures I2C */
    I2C_init();

    /* Specify the slave address for HDC1000 */
    I2C_setslave(HDC1000_SLAVE_ADDRESS);

    /* Write the configuration */
    if (!I2C_write24(HDC1000_CONFIGURATION, swapMSB_LSB(configuration),
    TIMEOUT)) {
        return 0;
    }

    return 1;
}
```

The action of swapping the bytes of the configuration register was required in order to perform the write operation according to the datasheet (see figure 5.4).

Scheduler

The scheduler.c and sheduler.h files implement a **custom scheduler**. The scheduler targets a *circular* behavior, but can be used to implement different *computing strategies* resorting to other solutions as we will see in chapter 8.

The scheduler.h file has an important role. First of all, we defined the scheduler **size**:

```
/* Scheduler size */
#define SCHED_SIZE  5
```

The key role is played by the **function pointer**:

```
/* Function pointer with no parameters */
typedef void (*fPtr)(void);
```

and the **scheduler structure** storing the actual scheduler status:

```
/* Scheduler structure */
typedef struct scheduler_ {
    int tail;       // Current tail
    int head;       // Current head
    int size;       // Current number of items
    int capacity;   // Capacity of scheduler
    fPtr *taskPtr; // Pointer to array of tasks
} scheduler;
```

The scheduler.c file implements the scheduler. The main point was to develop the Scheduler_create function able to **create** and **initialize** the scheduler:

```
scheduler* Scheduler_create(int capacity)
{
    /* Create the scheduler */
    scheduler *s = (scheduler*) malloc(sizeof(scheduler));

    /* Initialize the scheduler with a given capacity */
    if (s == NULL) {
        return NULL;
    } else {
        s->tail = -1;
        s->head = 0;
        s->size = 0;
        s->capacity = capacity;
        s->taskPtr = (fPtr*) malloc(capacity * sizeof(fPtr));
        return s;
    }
}
```

As you can see, after allocating the necessary space in memory, a correct initialization is performed to set the head, tail, size, capacity as well as the task function pointer.

The Scheduler_enqueue function **enqueues** a task:

```
int Scheduler_enqueue(scheduler* s, fPtr taskPtr)
{
    if (s == NULL) {
        return -1;
    } else if (Scheduler_isFull(s) == 1) {
        return 0;
    } else {
        s->tail = (s->tail + 1) % s->capacity;
        s->taskPtr[s->tail] = taskPtr;
        s->size++;
        return 1;
    }
}
```

The very first step is to check whether the scheduler passed as parameter is effective (not NULL) and check whether the queue is not full. In this case, the tail is moved upwards, the task pointer is added and the size increased.

The Scheduler_dequeue function **dequeues** a task:

```
fPtr Scheduler_dequeue(scheduler *s)
{
    fPtr taskPtr;
```

```
4
5       if (s == NULL) {
6           return -1;
7       } else if (Scheduler_isEmpty(s) == 1) {
8           return 0;
9       } else {
10          taskPtr = s->taskPtr[s->head];
11          s->head = (s->head + 1) % s->capacity;
12          s->size--;
13          return taskPtr;
14      }
15  }
```

Here again, the very first step is to check whether the scheduler passed as parameter is effective (not NULL) and check whether the queue is not empty. In this case, the task pointer is retrieved, the head is moved upwards and the size decreased.

The Scheduler_isEmpty function checks whether the queue is **empty**:

```
1  int Scheduler_isEmpty(scheduler* s)
2  {
3      if (s == NULL) {
4          return -1;
5      } else if (s->size == 0) {
6          return 1;
7      } else {
8          return 0;
9      }
10 }
```

whereas the Scheduler_isFull function checks whether the queue is **full**:

```
1  int Scheduler_isFull(scheduler *s)
2  {
3      if (s == NULL) {
4          return -1;
5      } else if (s->size == s->capacity) {
6          return 1;
7      } else {
8          return 0;
9      }
10 }
```

That's all. Our goal was to create a simple, but effective scheduler to run *tasks*. This is the topic of the next section.

Application

We implemented the application.c and application.h files to orchestrate the overall application.

The application.h file defines the macros setting the **application mode**:

```
1  /* Application mode */
2  #define APPLICATION_MODE_TEST          0
3  #define APPLICATION_MODE_NORMAL        1
```

and describing the current **application status**:

```
/* Application status */
#define APPLICATION_STATUS_INIT        0
#define APPLICATION_STATUS_RUNNING     1
#define APPLICATION_STATUS_SHUTDOWN    2
#define APPLICATION_STATUS_RESTART     3
#define APPLICATION_STATUS_NOTXRX      4
#define APPLICATION_STATUS_READY       5
```

The application.c file basically implements the task-based computing model we discussed in section 2.5.2: the division of the problem to solve in suitable **tasks**. From the project point of view, a task is the implementation of a functional requirement (§3.3.4). Tasks are run by the scheduler we have presented in the previous section.

The Application_init() function performs the initialization of the application:

```
void Application_init(int mode)
{
    /* Initialize the workload manager */
    WorkloadManager_init();

    /* Select mode */
    switch (mode) {

        case APPLICATION_MODE_TEST:
        Application_test();
        break;

        case APPLICATION_MODE_NORMAL:
        Application_setup();
        break;

        default:
        break;
    }

}
```

After initializing the *workload manager*, the application is run according to the selected *mode*. The role of the *workload manager* will be discussed in chapter 8, but now we can simply anticipate that it is responsible of managing the hardware and applying *policies*.

5.3.4 Entry point to the application

We implemented the **entry point** to the application thanks to main.c and main.h files. It was implemented with the aim of being simple and straightforward. Its goal is to initialize the batteryless environmental datalogger.

The main.h file simply defines a macro:

```
#define APPLICATION_FAILURE 1
```

The reader might be confused as this is an embedded system and code is usually never ending. The idea was to point out that the control of the code is given to the application and never returns.

The main.c file implements the main function which uses the macro:

```c
int main(void)
{
    /* Initialize the application */
    Application_init(APPLICATION_MODE_NORMAL);

    /* Should not reach this point */
    return APPLICATION_FAILURE;
}
```

Actually, if the code returns from the main function, a very unexpected behavior is the cause. This behavior usually never happens, but electronics sometimes might be challenging. We preferred to add that code although unreachable.

There is much more to describe about the implementation of the firmware, but the discussion is postponed to chapter 8. This choice might be strange, but there are good reasons. The mission of the overall application was to capture and send measures over the LoRaWAN® network. The key point is the datalogger had to be *batteryless* and *intermittent*; these features introduced new challenges. To cope with these challenges, we created a *worload manager*, able to apply different *policies*. Furthermore, we resorted to a library by TI to deal with the *intermittent* behavior. Since the discussion is relevant and central for this thesis work, we preferred to dedicate a new chapter. The reader may initially skip chapter 6 and 7 and directly jump to chapter 8, if the goal is to have a linear reading experience about the firmware implementation of the datalogger. The rest of this chapter simply describes how we documented and tested the code.

5.4 Documentation

Developers usually document their projects. Indeed, documenting a project is a good practice for at least two reasons. Above all, developers can think about their choices while they are documenting a peace of code. Comments usually help developers to better focus on a concept, on an algorithm step or a calculation. The other important reason is about memory. Documentation allows to understand how the code was built and can be used by developers to enter the topic and take the control of the code even after months or years. This is true both for whom wrote the code and whom will inherit the project.

Code Composer Studio (§A.1) enabled us to using *Doxygen*. According to creators,

> "**Doxygen** is a widely-used documentation generator tool in software development. It automates the generation of documentation from source code comments, parsing information about classes, functions, and variables to produce output in formats like HTML and PDF. By simplifying and standardizing the documentation process, Doxygen enhances collaboration and maintenance across diverse programming languages and project scales. It is free, open source and cross-platform"[10].

The Node root folder of the project contains the **Node.doxyfile** file. By double-clicking the file, a new window opened in Code Composer Studio allowing us to set up the documentation generator. Furthermore, there are several output formats available among which you can choose: HTML, LaTeX, Man Pages, rich text format (RTF) and XML. After saving, you have simply to right-click on the file and choose Build Documentation. The process creates a folder containing the generated documentation. We chose the HTML format, therefore the process generated a **html** folder into the root one.

To browse the generated documentation, you simply need to open the index.html file by choosing your preferred Browser. Figure 5.7 shows its output. As you can see, the generator

Figure 5.7: Code documentation of the datalogger.

works fine and pages are clean and easy to inspect. Browsing is straightforward and the level of detail depends on your settings which are performed thanks to the Node.doxyfile file.

The last point concerns how we documented the code. Here we report the code we used to document the main function into the main.c file as an example:

```
/**
 * @brief Entry point to the application.
 *
 * @return int
 */
int main (void)
{
    ...
}
```

It is not complicated, but interested readers may browse the Doxigen website to understand how to document the code with the aim of:

- putting comments in the code such that Doxygen incorporates them in the documentation it generates;
- structuring the contents of a comment block such that the output looks good.

The overall result is a professional documentation that will be properly consulted and appreciated by other developers.

5.5 Testing

Testing a firmware is not straightforward. Developers usually create further design units with the aim of verifying the correctness of their code when targeting software components

running on desktop or laptop computers. Instead, the firmware is usually written thanks to a development tool whose main components are the *cross compiler* and the *debugger*. The main goal is to *load* the firmware into the target MCU. Code Composer Studio integrates everything you need to achieve the goal.

Among all, developers should extensively use **comments** in order to document their code as we have seen in the previous section. Comments are a powerful way to document the relevant statements, think what you are doing and remember what you have done. In practice, comments are useful to test the code in the meanwhile you are writing it to focus on what you are implementing. This is important as developers do not write other code to test the firmware. When the code is ready, you cross compile and then load the firmware into the target MCU.

At this point, developers check the MCU behavior in the field. Usually, **electronic test instruments** are needed to measure and display signals. Having a multimeter, an oscilloscope, a signal generator and other instruments, is a good practice. We might state their are necessary during the firmware development. We obviously followed this approach. As an example, during the development, we needed to display on a digital oscilloscope the I2C communication between the MCU and the sensor. Some issues prevented SDA and SCL signals to be clean and well-formed. By displaying their waveforms, we understood the cause and fix the problem; the fixing would have been a nightmare without an oscilloscope.

Another important testing tool is the *debugger*. A **debugger** or **debugging tool** is a computer program used to test and debug other programs. The main problem is how to debug a firmware. We were lucky, as the MSP-EXP430FR5994 LaunchPad™ Development Kit integrates an onboard eZ-FET debug probe as we saw in section 4.2.3. It allowed us to debug and program the MCU as well as communicate to the PC. This unit also provided power to target MCU during the development. Code Composer Studio has a powerful debugger we extensively used to test our code step by step. We might say that debugging is a normal activity during the development and helps solving *any* kind of issue thanks to *breakpoints*, that is, statements for which developers want to break the code execution to test or check the MCU current state.

Before leaving this chapter, we would like to mention a powerful tool we used during the development: the **EnergyTrace++ technology™**. This tool allows a real-time power consumption reading and state updates from the MSP430FR5994 MCU, including CPU and peripheral state to be viewed through the EnergyTrace GUI [22]. The tool might be considered both a debugging and testing tool as allows developers to perform energy and power consumption measures with the aim of *estimating* how much the target system is power-hungry. It played a key role as ours was a batteryless embedded system. We will provide details in chapter 9.

In this chapter, we have explained how we implemented the firmware running on the target microcontroller. We have seen that our solution is based on a custom scheduler whose role is to run relevant tasks. As already mentioned, the reader might directly jump to chapter 8 to understand how we managed the *context* and tried different *computing strategies* to maximize and control the intermittent behavior of the datalogger. Furthermore, the same chapter explains how we performed some optimizations aiming of reducing the power consumption as much as possible.

Chapter 6

Server

In this chapter, we will explain how we implemented a simple and reliable REST Web service capable of managing and storing data provided by the datalogger and gateway. The design and implementation of a Web service should be faced before creating any client. This is required, because developers must decide how the service can be consumed by a client.

6.1 Web services

To understand the role of Web services, it might be better to introduce some concepts and motivations.

The **Internet** is a computer network that interconnects billions of *computing devices* throughout the world [27]. It is also known as *network of networks*. It is not the only one, but in the following we are going to focus on the *public* Internet. A **computing device** may be:

- a traditional *desktop* or *laptop* PC;
- a *server* that stores and transmits information such as Web pages;
- a *thing* such as a smartphone, a tablet, a TV, a gaming console, a thermostat, a home security system, a home appliance, a watch, a car, a traffic control system and much more. Of course, it may be a datalogger.

In Internet jargon, all of these devices are called **hosts** or **end systems** [27].

For our discussion, we are not interested in describing the Internet as a set of hardware and software components, but rather as a networking infrastructure that provides services to *distributed applications*. A **distributed application** involves multiple end systems that exchange data with each other [27]. As an example, in our project a *gateway* and a *server* are end systems exchanging data. Actually, data are exchanged by applications running on end systems. These applications are implemented thanks to common programming languages such as C, Java or Python. Here, we do not want to review the basics of networking, but focus on how data are exchanged among distributed applications.

The problem of exchanging data among distributed applications is so relevant. One possible solution is based on **remote procedure calls (RPC)** systems. A RPC is when a

computer program causes a procedure to execute on another computer on a shared network, which is written as if it were a local procedure call. The programmer do not explicitly write the details for the remote interaction [56]. The main drawbacks are complexity and low interoperability. Another solution concerns **distributed object** systems. In this case, a *middleware* layer transfers the object model into a distributed environment. The objects of a single application program can *live* in multiple hosts connected by a network, but each object can be used as if it were local. Such strategy is simpler and more interoperable. Nowadays, the most advisable solution takes into account the **Web services**. They are Web applications exchanging data thanks to open Internet protocols and standards such as HTTP, SOAP, XML and JSON. The main advantages are high simplicity and high interoperability. Figure 6.1 shows these solutions as well as their main architectures.

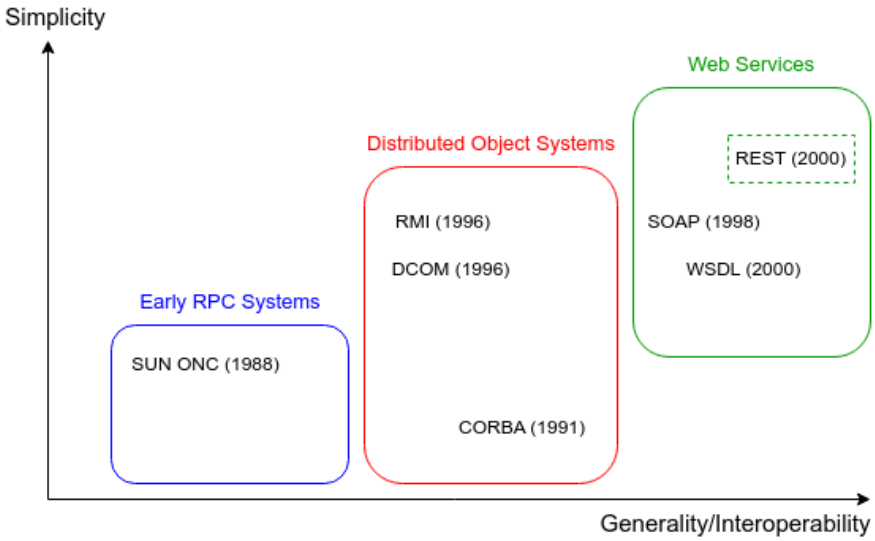


Figure 6.1: Solutions for the interaction among distributed applications.

Web services are also known as *WEB application programming interfaces* or **Web API**. Figure 6.1 shows that Web APIs can be built thanks to different architectures. In particular, we were interested in **REST Web services** that we are going to present now.

6.2 REST architecture

REST stands for **REpresentational State Transfer**. It is a way for building distributed applications that was introduced by Roy Fielding, one of the main developers of HTTP. What is amazing is that the topic represents the core of his PhD Thesis. REST is implemented by the HTTP protocol and derived starting with the Null Style describing a system in which there are no distinguished boundaries between components [13].

6.2.1 Constraints

REST was developed starting with the system needs as a whole. After defining the Null Style as the empty set of *constraints*, specific **constraints** were incrementally added in order to describe the software engineering principles guiding REST. They are:

- **client-server**: the *separation of concerns* between client and server components allow them to evolve independently;
- **stateless**: the session *state* is kept entirely on the client;
- **cache**: data within a response to a request are implicitly or explicitly labeled as *cacheable* or *non-cacheable*;
- **uniform interface**: it is the central feature that distinguishes the REST architectural style from other network-based styles. The server exposes a uniform interface for accessing *resources* without defining their representation;
- **layered system**: the architecture have to be composed of *hierarchical layers* to promote substrate independence;
- **code on demand**: client functionality can be extended by downloading from the server and executing code in the form of applets or scripts. This is an optional constraint.

A REST Web service is any service that adheres to the constraints we have reported here [13].

6.2.2 Resources and resource identifiers

A **resource** is the key abstraction of information in REST. It is any information such as a document, an image, a temporal service or a collection of other resources. Each resource can be identified through a public **uniform resource identifier (URI)**. Each resource may have one or more **representations** which capture the current or intended state of it. From the technical point of view, a representation is a sequence of bytes, plus representation metadata to describe those bytes . Example of representations are HTML documents and JPEG images [13].

When a client requests an operation on a resource to a Web API, a representation of the resource may be transferred. Resources themselves are never transferred. They stay always in the server side where their URI points to. As an example, a URI may be:

```
https://127.0.0.1:8080/api/gateways
```

In Web API jargon, a URI is called **endpoint**. We already introduced this concept in section 3.4.2. In practice, a REST Web API exposes a set of public URIs which are used by clients to perform operations on resources.

6.2.3 Operations

The REST style is implemented with the HTTP protocol that offers a fixed and uniform interface to act on resources. Indeed, the semantic of each **operation** is defined by HTTP unambiguously and depends on:

- the requested method;
- the request body;
- the request headers;
- the target URI.

Each *request* has a *response* containing the result of the operation. In particular, the response contains the headers, body and *status code* we are going to introduce shortly.

Basically, HTTP provides the create, read, update, delete (CRUD) operations defined by REST. Table 6.1 reports the operations you may perform on a resources (*res*). In general, a **REST API client** application can use these HTTP methods to access resources of a REST Web service.

CRUD operation	**HTTP request**	**HTTP response (if no error)**
Create a new resource under *res* (or send data to *res* for resource-specific processing)	POST *res* (resource or data representation in the body)	URI of new resource created and/or result of processing (in body)
Read resource *res*	GET *res* (no body)	representation of current state of *res* (in the body)
Update resource *res* by replacing its state with a new one (*res* can be created)	PUT *res* (new resource representation in the body)	description of executed operation (200 or 201 or 204 status code)
Delete resource *res*	DELETE *res*	-
Read information about resource *res* without transferring its representation	HEAD *res* (no body)	same as for GET but without body (only headers sent)
Read supported methods for a resource	OPTIONS *res* (no body)	list of supported methods (in the allow header and possibly in the body)
Test the connection to resource *res* (loopback testing similar to echo)	TRACE *res*	the received request (in the body)
Partially update the resource res by applying a patch	PATCH *res* (patch to be applied to *res* in the body)	description of executed operation (200 or 201 or 204 status code)

Table 6.1: HTTP operations on resources.

6.2.4 Status codes

A key role is played by HTTP **status codes**. As we have briefly mentioned in the previous section, REST clients perform HTTP requests to REST APIs. Each request has a response containing, among other data, a status code that provides information about results. If there are no errors, the HTTP response has a status code such as the ones reported in table 6.1.

More in detail, status codes are numbered based on the category of the result. Table 6.2 reports the status code categories, whereas table 6.3 reports the most common status codes you may use for REST applications.

Range	Category
1xx	Request received
2xx	Successful operation
3xx	Redirection
4xx	Client error
5xx	Server error

Table 6.2: HTTP status code categories [38].

Code	Meaning	Description
200	OK	The requested action was successful
201	Created	A new resource was created
202	Accepted	The request was received, but no modification has been made yet
204	No Content	The request was successful, but the response has no content
400	Bad Request	The request was malformed
401	Unauthorized	The client is not authorized to perform the requested action
404	Not Found	The requested resource was not found
415	Unsupported Media Type	The request data format is not supported by the server
422	Unprocessable Entity	The request data was properly formatted but contained invalid or missing data
500	Internal Server Error	The server threw an error when processing the request

Table 6.3: HTTP most common status codes [38].

6.3 Technologies

In section 3.4.2, we modeled the target REST API with a set of endpoints. To implement the service, we chose three basic technologies:

- **CherryPy** for building the service itself;

- **SQLAlchemy** for storing data;
- **JSON** for exchanging data.

We had to meet FR3 class functional requirements and NFR7, NFR8, NFR9 non functional requirements. Let us now briefly present these technologies.

6.3.1 CherryPy

According to creators,

> "**CherryPy** is a *pythonic*, object-oriented web framework. It allows developers to build Web applications in much the same way they would build any other object-oriented Python program. This results in smaller source code developed in less time"[4].

Basically, our choice was based on a simple, reliable and agile technology to target a Web application for experimental purposes. Among many, we might have chosen a Java-based technology such as Spring Boot or a Microsoft-based technology such as ASP.NET, but they usually require more effort as they target production-grade applications.

At the end of the day, CherryPy is a **Web application server** as it aggregates a *Web server* and an *application server* into a single component. A **Web server** is the interface dealing with the HTTP protocol because it transforms incoming HTTP requests into entities that are then passed to the *application server* and also transforms information from the *application server* back into HTTP responses. An **application server** is the component hosting one or more *applications*. From the *server side* point of view, an **application** is a piece of software that takes a unit of information, applies business logic to it and returns a processed unit of information [46].

CherryPy's creators had in mind a *simple*, but powerful framework. Let us take into account the following code which demonstrates the most basic application you might write with CherryPy:

```
1 import cherrypy
2
3 class HelloWorld(object):
4   @cherrypy.expose
5   def index(self):
6     return "Hello world!"
7
8 if __name__ == '__main__':
9   cherrypy.quickstart(HelloWorld())
```

Listing 6.1: CherryPy: basic application.

The role of this code snippet is to start a server and host an application that will be served at request reaching http://127.0.0.1:8080/ [4]. The code can also be used to test your environment setup. We saved this code into a file called ServerHW.py; the file is located into the root Server folder. By executing the code, thanks to the following command:

```
python3 ServerHW.py
```

you should get the following outcome:

```
[24/Apr/2025:10:12:31] ENGINE Listening for SIGTERM.
[24/Apr/2025:10:12:31] ENGINE Listening for SIGHUP.
[24/Apr/2025:10:12:31] ENGINE Listening for SIGUSR1.
[24/Apr/2025:10:12:31] ENGINE Bus STARTING
CherryPy Checker:
The Application mounted at '' has an empty config.

[24/Apr/2025:10:12:31] ENGINE Started monitor thread 'Autoreloader'.
[24/Apr/2025:10:12:31] ENGINE Serving on http://127.0.0.1:8080
[24/Apr/2025:10:12:31] ENGINE Bus STARTED
```

> "The first three lines indicate the server will handle *signals* for you. The next line tells you the *current state* of the server, as that point it is in STARTING stage. Then, you are notified your application has no specific *configuration* set to it. Next, the server starts a couple of internal utilities that we will explain later. Finally, the server indicates it is now ready to accept incoming communications as it listens on the address 127.0.0.1:8080 implying that the application is ready to be used"[4].

You may open your favorite Browser at the previous address and see the result.

That's all. With this example we wanted to prove how easy is to create and run a new application with CherryPy. Of course, ours is a bit more complex. We are going to explain everything about our code and its configuration in the next sections.

6.3.2 SQLAlchemy

According to creators,

> "**SQLAlchemy** is the Python SQL toolkit and *Object Relational Mapper* that gives application developers the full power and flexibility of SQL. It provides a full suite of well-known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and *pythonic* domain language"[43].

One of the most significant front-facing portion of SQLAlchemy is the **object relational mapper (ORM)** using which classes can be mapped to the database, thereby allowing the object model and database schema to develop in a cleanly *decoupled* way from the beginning.

We will discuss about our database implementation later, but here we would like to give an example to prove how easy is to create and start working with a database:

```
1 from sqlalchemy import create_engine
2
3 engine = create_engine('sqlite:///Database.sqlite', echo=True)
```

Listing 6.2: SQLAlchemy: simple example.

This code creates a database file called Database.sqlite, whereas engine is an object acting as a central source of connections to a database. SQLAlchemy uses a third party driver, that is a DBAPI implementation, in order to interact with a particular database. In practice, it resorts to **dialects**. All dialects require that an appropriate DBAPI driver is installed [43].

We decided to use the **SQLite** dialect, but we could have chosen another one such as Firebird, Microsoft SQL Server, MySQL, Oracle, PostgreSQL or Sybase.

6.3.3 JSON

According to creators,

> "**JavaScript object notation (JSON)** is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. Its main feature is to be a text format that is completely **language independent**"[26].

We chose JSON for data-interchange, although a possible alternative would have been the **extensible markup language (XML)**. However, compared to XML, JSON allows faster data transfers as the size of files is smaller.

6.4 Developed code

This section describes the code we implemented by construction as we have done in the previous chapter. We refer to files located into the Server folder. Furthermore, please refer to section A.2.2 to check whether all the required modules have been successfully installed.

6.4.1 Entry point to the application

The Server.py file, whose content is reported in the following listing, represents the entry point to the application:

```
import cherrypy
from Repository.DatabaseManager import DatabaseManager
from Controller.Controllers import GatewayController
import os.path

class Application(object):

  @cherrypy.expose
  def index(self):
    return open('Static/views/index.html')

  @cherrypy.expose
  def user(self):
    return open('Static/views/user.html')

class API(object):

  @cherrypy.expose
  def index(self):
    return open('Static/views/api.html')

# Entry point to the application
def main():

  # Update the global CherryPy configuration
  cherrypy.config.update('Config/Server.conf')

  # Check the DB file existence
  checkDBFile()

```

```
31    # Create the database
32    DatabaseManager.create_database()
33
34    # Create an instance of the application
35    application = Application()
36
37    # Attach an instance of the API class to the main application
38    application.api = API()
39
40    # Attach an instance of the GatewayController class to the api instance
41    application.api.gateways = GatewayController()
42
43    # Mount the application on the '/' base path
44    cherrypy.tree.mount(application, '/', config='Config/Application.conf')
45
46    # Start the builtin server and engine
47    cherrypy.engine.start()
48
49    # Wait for the EXITING state, KeyboardInterrupt or SystemExit (block)
50    cherrypy.engine.block()
51
52 # Check the DB file existence. If there is no file,
53 # the contents of Measures.txt file will be erased.
54 def checkDBFile():
55     if ((os.path.exists("Repository/Database.sqlite")) is False):
56         open("Repository/Measures.txt", "w").close()
57
58 # Standard Python check on whether the file is used directly or as module
59 if __name__ == '__main__':
60    main()
```

Listing 6.3: Web service: entry point.

Above all, the cherrypy module must be imported to work with CherryPy. We did not need anything else to deal with this technology. We also needed to import DatabaseManager and GatewayController classes that will be presented later on.

By inspecting the code, the reader may notice we wrote a main method. We preferred to use this approach with the aim of clarifying exactly the set of steps to be performed to create and run the application.

As a first step, we updated the global CherryPy **configuration**. This topic will be discussed in the next section. Right now, we can simply anticipate that thanks to a file, we defined the *host* and *port* on which the server would have listened for incoming connections as well as other useful global configurations.

Then, we checked about the existence of the Database.sqlite file. If the file was not present at sturtup, the contents of Measure.txt file would have been erased. The role of Measure.txt file will be discussed in section 9.3. Both files are located into the Repository folder.

As a next step, we created the **database**. At the beginning, the create_database method of the DatabaseManager class, was a placeholder to take into account this crucial activity. We will return to this topic later on as there are a lot of concepts to be discussed.

Here is the key point. CherryPy uses its own internal lookup algorithm to retrieve the *handler* referred to the Request-URI. A **handler** is a Python-callable object attached to a

tree of published objects. Each Request-URI maps to a Python object. CherryPy resorts to two important concepts:

- **published**: *"a Python object is said to be published when it is attached to a tree of objects and the root of this tree is mounted on the CherryPy engine server via a call to cherrypy.tree.mount or cherrypy.quickstart methods"* [46];
- **exposed**: *"a published object is said to be exposed when it has an attribute named exposed set to True. An exposed object must be Python callable"* [46].

It is important to point out that being published is not sufficient for an object to be treated as a potential handler for a URI by CherryPy. A published object must be exposed so that it becomes visible to the CherryPy engine. An exposed object is usually referred to as a **page handler** by the CherryPy community [46].

To define the **tree of exposed objects**, we wrote the Application and API classes; we also imported the GatewayController class. All the classes have exposed methods. We created the objects with the following instructions:

```
1 application = Application()
2 application.api = API()
3 application.api.gateways = GatewayController()
```

Our goal was to create the relationship between a Request-URI and the page handler shown in table 6.4. In practice, we were interested in exploiting the default mechanism of matching URLs we have previously discussed for managing requests which where not related to the API, such as /user. On the other hand, we wanted to switch to the mechanism that was aware of HTTP operations, such as /api/gateways. To achieve the goal, we also needed to set the request.dispatch property into the application configuration file as we will see in the next section.

Request-URI Path	Published Object	Page Handler
/	application	index
/user	application	user
/api	application.api	index
/api/gateways	application.api.gateways	HTTP method dispatcher

Table 6.4: Relationship between a Request-URI and the page handler.

Then, we had to **mount the application** on a selected base path. We performed this action thanks to the following instruction:

```
1 cherrypy.tree.mount(application, '/', config='Config/Application.conf')
```

The application object works as a *root* mounted on the '/' base path. The cherrypy.tree.mount method has three parameters:

- **root**: an instance of a *controller class* (a collection of page handler methods) which represents the root of the application. This may also be an *Application* instance, or *None* if using a dispatcher other than the default;

- **script_name**: a string containing the *mount point* of the application. This should start with a slash, and be the path portion of the URL at which to mount the given root. For example, if root.index() will handle requests to:
  http://www.example.com:8080/dept/app1/
  then the script_name argument would be /dept/app1. It must not end in a slash. If the script_name refers to the root of the URI, it must be an empty string (not "/");
- **config**: a file or dictionary containing the application configuration.

Finally, we started and blocked the server:

```
cherrypy.engine.start()
cherrypy.engine.block()
```

Actually, developers may mount more than one application on a CherryPy server by calling the cherrypy.tree.mount method several times. These applications would be unaware of each other; furthermore, each application should have its own configuration file [46]. In our case, we were interested in mounting and running a **single application**. This means that the following lines of code:

```
cherrypy.tree.mount(application, '/', config='Config/Application.conf')
cherrypy.engine.start()
cherrypy.engine.block()
```

could be totally substituted by the following one:

```
cherrypy.quickstart(application, '/', config='Config/Application.conf')
```

The result would have been the same as the method mounts the given root, starts the builtin server and engine, then blocks. Hence the name quickstart.

6.4.2 Entities, attributes and relations

Now, we will briefly describe the *entities* that the application can manipulate as well as their *attributes* and *relations* with other *entities*.

In computer science, an **entity** is an object that has an identity, which is independent of the changes of its *attributes*. It represents long-lived information relevant for the users and is usually stored in a database [52]. An **attribute** is a specification that defines a property of the entity [49]. A **relation** represents the way in which two or more entities are connected.

According to our specifications, we defined three main **entities**: gateway, datalogger and measure. In particular:

- gateway: concentrates one or more dataloggers:
  - id: [primary key] uniquely identifies a gateway in the application;
  - ip4: [attribute] Internet Protocol version 4 address (mandatory);
  - ip6: [attribute] Internet Protocol version 6 address.
- datalogger: [entity] performs measures:
  - id: [primary key] uniquely identifies a datalogger in the application;

  - serial: [attribute] serial number of the sensor mounted on the datalogger (mandatory);
  - gateway_id: [foreign key] gateway that concentrates the datalogger.

- measure: [entity] a measure performed by a datalogger:
  - id: [primary key] uniquely identifies a measure in the application;
  - temperature: [attribute] temperature in Celsius (mandatory);
  - humidity: [attribute] relative humidity in percentage (mandatory);
  - timestamp: [attribute] timestamp of the measure (mandatory);
  - datalogger_id: [foreign key] datalogger that performed the measure.

Concerning the **relations** we defined that:

- a gateway has zero to many dataloggers;
- a datalogger has zero to many measures.

6.4.3 Repositories

After defining entities, attributes and relations, we started working on the **database**. Our goal was to have an application to manipulate entities via the traditional CRUD interface. We were interested in creating the database structure shown in figure 6.2. The database structure is quite simple and reflects the nature of our experimental application.

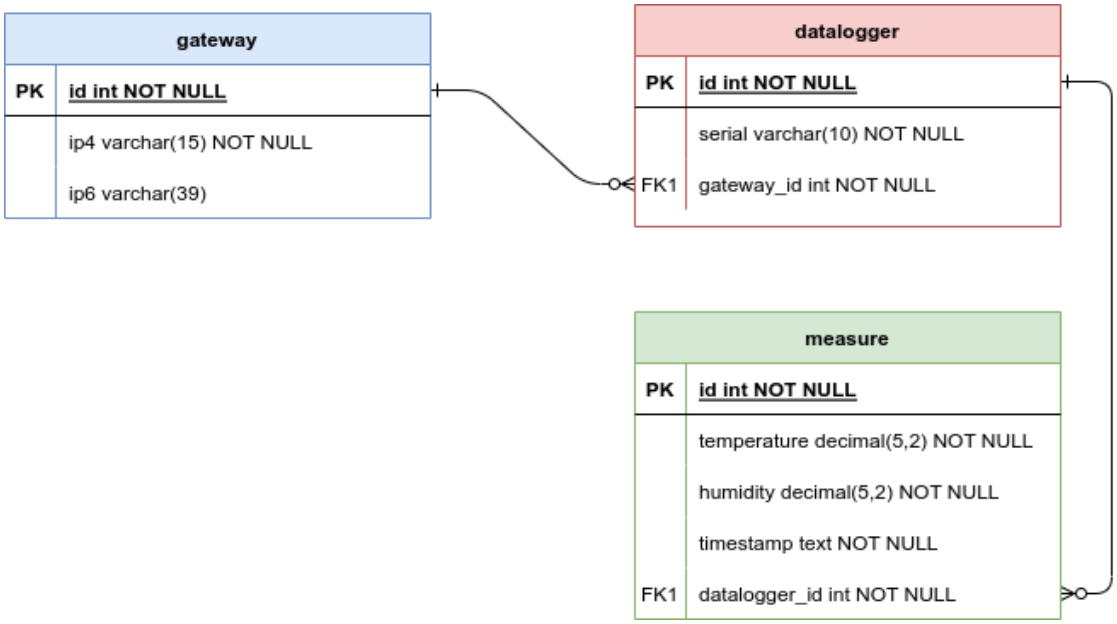


Figure 6.2: Database structure of the application.

As we have already discussed, we selected the SQLAlchemy technology to achieve the goal. In particular, we highly exploited the object relational mapper (ORM). This **extra**

layer maps or translates an object into a database row and viceversa. It reduces the developer's effort considerably as a manual approach would be a nightmare, especially if you introduce a modification. This kind of layers may also have negative impacts from the performance point of view, especially if you deal with large and complex projects. However, for us benefits overcame drawbacks.

In the following, we are going to refer to files located into the Repository folder. As a first step, we wrote the Repositories.py file to declare **models**. More in detail, we defined the *module-level constructs* that would have formed the structures we would have queried from the database. This activity, known as **declarative mapping**, defines at once both a **Python object model**, as well as **database metadata** that describes real SQL tables that exist, or will exist, in a particular database.

Mapping

The mapping started with a Base class created by making a simple subclass against the DeclarativeBase class to be imported from sqlalchemy.orm.

```
class Base(DeclarativeBase):
    pass
```

> "Individual mapped classes are then created by making subclasses of Base. A mapped class typically refers to a single particular database table, the name of which is indicated by using the __tablename__ class-level attribute. Then, columns that are part of the table are declared, by adding attributes that include a special typing annotation called Mapped. The name of each attribute corresponds to the column that is to be part of the database table. More specific typing information may be indicated using SQLAlchemy type objects in the right side mapped_column directive. This directive is used for all column-based attributes that require more specific customization. Besides typing information, this directive accepts a wide variety of arguments that indicate specific details about a database column, including server defaults and constraint information, such as membership within the primary key and foreign keys. All ORM mapped classes require at least one column be declared as part of the primary key. The introduced mapping approach is known as **annotated declarative table configuration**"[43].

We stared by writing the GatewayRepository mapped class:

```
class GatewayRepository(Base):
    __tablename__ = 'gateway'

    id: Mapped[int] = mapped_column(primary_key=True)
    ip4: Mapped[VARCHAR]= mapped_column(VARCHAR(15), nullable=False)
    ip6: Mapped[VARCHAR]= mapped_column(VARCHAR(39), nullable=True)

    dataloggers: Mapped[List["DataloggerRepository"]] = relationship(
    back_populates="gateway", cascade="all, delete-orphan")

    def __repr__(self) -> str:
        return "Gateaway(id={self.id!r}, name={self.ip4!r}, fullname={
    self.ip6!r})"

```

```
13      def as_dict(self):
14          return {c.name: getattr(self, c.name) for c in self.__table__.
        columns}
15
16      def to_json(self):
17        return dict(id=self.id, ip4=self.ip4, ip6=self.ip6)
```

Listing 6.4: Gateway: repository mapped class.

As you can see, we included the table name and attributes. Each attribute would have become a column in the table capable of storing a particular data type according to our specifications. We also used the relationship construct, that is, another variant of Mapped, in order to denote a linkage between two ORM classes. This topic will be discussed after introducing the other mapped classes. Finally, we also included the ___repr___, as_dict_ and to_json_ methods to convert data to a particular format. We mainly exploited the to_json method.

Let us now introduce the other mapped classes without reporting methods to convert data for the sake of simplicity:

```
1  class DataloggerRepository(Base):
2      __tablename__ = 'datalogger'
3
4      id: Mapped[int] = mapped_column(primary_key=True)
5      serial: Mapped[VARCHAR] = mapped_column(VARCHAR(10), nullable=False)
6      gateway_id: Mapped[int] = mapped_column(ForeignKey("gateway.id"))
7
8      gateway: Mapped["GatewayRepository"] = relationship(back_populates="
       dataloggers")
9
10     measures: Mapped[List["MeasureRepository"]] = relationship(
       back_populates="datalogger", cascade="all, delete-orphan")
```

Listing 6.5: Datalogger: repository mapped class.

```
1  class MeasureRepository(Base):
2      __tablename__ = 'measure'
3
4      id: Mapped[int] = mapped_column(primary_key=True)
5      temperature: Mapped[TEXT]  = mapped_column(TEXT, nullable=False)
6      humidity: Mapped[TEXT]  = mapped_column(TEXT, nullable=False)
7      timestamp: Mapped[TEXT] = mapped_column(TEXT, nullable=True)
8      datalogger_id: Mapped[int] = mapped_column(ForeignKey("datalogger.id"
       ))
9
10     datalogger: Mapped["DataloggerRepository"] = relationship(
       back_populates="measures")
```

Listing 6.6: Measure: repository mapped class.

Everything should be clear, but relations between classes have to be clarified:

- one gateway should have zero to many dataloggers. GatewayRepository.dataloggers links GatewayRepository to DataloggerRepository; DataloggerRepository.gateway links DataloggerRepository to GatewayRepository;

- one datalogger should have zero to many measures. DataloggerRepository.measures links DataloggerRepository to MeasureRepository; MeasureRepository.datalogger links MeasureRepository to DataloggerRepository.

This kind of mappings might seem tricky, but SQLAlchemy has a large documentation that developers can easily consult to go more in depth and understand each construct [43].

Serialization

A key point was about **serialization** and **de-serialization** of SQLAlchemy objects. We used the marshmallow-sqlalchemy library to perform these activities.

After declaring the models, that is, the mapped classes we have discussed before, we had to generate the marshmallow **schemas**. Developers must be sure to declare models before instantiating schemas. We automatically generated fields for a model's columns using SQLAlchemyAutoSchema:

```
class MeasureRepositorySchema(SQLAlchemyAutoSchema):
    class Meta:
        model = MeasureRepository
        load_instance = True
```

```
class DataloggerRepositorySchema(SQLAlchemyAutoSchema):
    class Meta:
        model = DataloggerRepository
        load_instance = True

    measures = fields.Nested(MeasureRepositorySchema, many=True)
```

```
class GatewayRepositorySchema(SQLAlchemyAutoSchema):
    class Meta:
        model = GatewayRepository
        load_instance = True

    dataloggers = fields.Nested(DataloggerRepositorySchema, many=True)
```

That's all. We wish to thank the authors' library as its role was fundamental in our project. We will see that serializing and de-serializing data is very easy and straightforward. Furthermore, we suggest the reader to **put models and schemas in the same file** with this order: models before and schemas after. The contents of Repositories.py file follow this approach.

Database manager

We created the DatabaseManager.py file into the Repository folder. Its role is to create the database, the sessions and perform CRUD operations.

As a first step, we declared the engine variable:

```
engine = create_engine('sqlite:///Repository/Database.sqlite', echo=True)
```

> "The start of any SQLAlchemy application is an object called the **Engine**. This object can be created thanks to the create_engine method which creates a new _engine.Engine instance. This object acts as a central source of connections to a particular database,

> providing both a *factory* as well as a holding space called a *connection pool* for these database connections. The engine is typically a global object created just once for a particular database server, and is configured using a URL string which will describe how it should connect to the database host or backend"[43].

As you can see, we selected the **SQLite** database to be created in the Repository folder. The selected database name is Database.sqlite. If the echo param is set to True, the engine logs all statements.

Then, we declared the **sessionmaker** to interact with the database as well as obtain its handle:

```
Session = sessionmaker(bind = engine)
```

> "A session object is the handle to database. Session class is defined using sessionmaker, a configurable session *factory method* which is bound to the engine object created earlier"[43].

Here the key point. After creating the DatabaseManager class, we implemented the create_database method:

```
class DatabaseManager(object):

    def create_database():
        Base.metadata.create_all(engine)
```

The single instruction performed by the method is simple, but powerful. It creates all tables stored in the Base class metadata. Conditional by default, will not attempt to recreate tables already present in the target database. The engine param is used to access the database [43]. Obviously, we had to import models and schemas we have previously introduced.

We called this method from the main method of Server.py file:

```
DatabaseManager.create_database()
```

After running the application, if not present, a new Database.sqlite file is created into the Repository folder. This allowed us to start easily storing and retrieving our data. Anytime we dropped the database during the development, it was created from scratch at the first application run.

Finally, we would like to present two example of operations performed on the database. We wrote a set of methods into the DatabaseManager class which were called by the *services* that we will introduce later on. The createGateway method was used to perform a *create* operation into the database:

```
def createGateway(gateway):
    session = Session()
    gateway = GatewayRepository(ip4 = gateway.ip4, ip6 = gateway.ip6)
    session.add(gateway)
    session.commit()
    session.refresh(gateway)
    return gateway.id
```

As you can see, after creating a session object, we created a GatewayRepository object by using the data coming from the *service*. The session was used to add and commit. To

retrieve the identifier of the created record, we refreshed and returned the identifier to the service. The readAllGateways method was used to *read* all the available data:

```
def readAllGateways():
    session = Session()
    gateways = session.query(GatewayRepository).all()
    return [GatewayRepositorySchema().dump(gateway) for gateway in
    gateways]
```

In practice, we queried the database to read a list of GatewayRepository objects. Then, the list were **serialized** and **dumped** thanks to GatewayRepositorySchema. Indeed, our goal was to send to the client a *response* by exploiting the JSON representation.

6.4.4 Models

Model–view–controller (MVC) [53] is a software *design pattern* commonly used for developing user interfaces that divides the related program logic into three interconnected elements:

- the model: the internal representations of information;
- the view: the interface that presents information to and accepts it from the user;
- the controller: the software linking the two.

We will take into account the view in chapter 7, but we can start describing the model.

First of all, we should not confuse the set of classes we wrote in the Models.py file located into the Model folder, with the repository classes we have already discussed. The idea is to describe the internal representations of information, or **model**, from the application perspective and not from the database perspective. That's why we wrote the following classes:

```
class Gateway(object):

    def __init__(self, ip4, ip6):
        self.id = None
        self.ip4 = ip4
        self.ip6 = ip6

    def __str__(self):
        pass
```

```
class Datalogger(object):

    def __init__(self, serial, gateway_id):
        self.id = None
        self.serial = serial
        self.gateway_id = gateway_id

    def __str__(self):
        pass
```

```
class Measure(object):

    def __init__(self, temperature, humidity, timestamp, datalogger_id):
        self.id = None
        self.temperature = temperature
        self.humidity = humidity
        self.timestamp = timestamp
        self.datalogger_id = datalogger_id

    def __str__(self):
        pass
```

Their meaning will be better clarified briefly, after introducing *controllers*.

6.4.5 Controllers

In general, a **controller**, is an object that accepts input and converts it to commands for the model or view. Since we were taking into account a Web application, a controller object for us, was a CherryPy exposed object or **page handler** capable of handling a URI as we have discussed in section 6.4.1.

The key point was how to perform a **RESTful-style dispatching**, or in other words, how to handle endpoints like the following:

```
http://127.0.0.1:8080/api/gateways/
http://127.0.0.1:8080/api/gateways/1
http://127.0.0.1:8080/api/gateways/1/dataloggers
http://127.0.0.1:8080/api/gateways/1/dataloggers/1
http://127.0.0.1:8080/api/gateways/1/dataloggers/1/measures
```

We created the Controllers.py file located into the Controller folder. Here, we report part of controllers definitions to discuss the strategy we used:

```
@cherrypy.expose
@cherrypy.popargs('gateway_id')
class GatewayController(object):

    def __init__(self):
        self.dataloggers = DataloggerController()
        self.service = GatewayService()
```

```
@cherrypy.expose
@cherrypy.popargs('datalogger_id')
class DataloggerController(object):

    def __init__(self):
        self.measures = MeasureController()
        self.service = DataloggerService()
```

```
@cherrypy.expose
class MeasureController(object):

    def __init__(self):
        self.service = MeasureService()
```

First of all, you may notice the @cherrypy.expose *decorator*. If applied to a class, it exposes all the methods of the class.

Then, in the GatewayController constructor we have an instance of DataloggerController, whereas in the DataloggerController constructor we have an instance of MeasureController. We will discuss about *services* later on.

Let us focus on the @cherrypy.popargs decorator that played a crucial role. Its mechanism was a bit tricky, but worked fine. The decorator provides a _cp_dispatch function that *pops off* path segments into cherrypy.request.params under the names specified. The dispatch is then *forwarded* on to the next vpath element [4]. It is not so simple to describe the mechanism, but we can try by describing peaces of code. The REST service page handler is:

```
http://127.0.0.1:8080/api/gateways
```

and we created its page handler in the main method of Server.py file thanks to the following exposed object:

```
application.api.gateways = GatewayController()
```

Actually, we needed to set a further configuration in Application.conf file located into the Config folder as we will see in the next section. The GatewayController class declares HTTP methods to deal with REST. The GET method has the following implementation:

```
@cherrypy.tools.json_out()
def GET(self, gateway_id=None):
    if gateway_id is None:
        return self.service.getAllGateways()
    else:
        gateway = self.service.getGatewayByID(gateway_id)

    if gateway is None:
        raise cherrypy.NotFound()
    else:
        return gateway
```

If gateway_id is not present, the user performed the following GET request:

```
http://127.0.0.1:8080/api/gateways
```

In this case, the controller calls the service to retrieve all the available gateways. Otherwise the presence of gateway_id means the user performed a GET request like:

```
http://127.0.0.1:8080/api/gateways/1
```

meaning, the controller calls the service to retrieve the gateway whose identifier matches gateway_id. You may notice the presence of @cherrypy.tools.json_out() decorator in order to send JSON data as a response. Now, let us take into account the POST method:

```
@cherrypy.tools.json_in()
def POST(self):
    current = cherrypy.request.json

    gateway = Gateway(current['ip4'], current['ip6'])

    gateway_id = self.service.addGateway(gateway)

```

```
9       if gateway_id is not None:
10          cherrypy.response.status = '201 Created'
11          cherrypy.response.headers['Location'] = 'http://%s:%s/%s/%d' %(
    host, port, api, gateway_id)
```

This method is called with a POST request like:

```
http://127.0.0.1:8080/api/gateways
```

In this case, after retrieving data from the request, we call the service to add a new gateway. If the operation is successfully executed, the service returns the new resource identifier and the controller can send the user the URI of created resource. As you can see, the controller refers to the model to create a gateway object to pass to the service. We will return on this topic later on. The controller also implements PUT and DELETE requests the reader can easily inspect. The next example should clarify the role of @cherrypy.popargs decorator. Let us refer to a GET request such as the following:

```
http://127.0.0.1:8080/api/gateways/1/dataloggers
```

This endpoint is managed by the DataloggerController class. It is possible thanks to the dataloggers field of GatewayController object. In practice, cherrypy.popargs gives a name to any segment that CherryPy would not be able to interpret otherwise. This makes the matching of segments with page handler signatures easier and helps CherryPy understand the structure of our URL [4].

The reader may wonder why using this kind of URI Dispatching instead of using a popular third-party package like Routes or another mechanism like Virtual Host. The idea was to exploit the features provided by CherryPy technology as much as possible. However, CherryPy offers another way to deal with dispatching, that is, the special ___cp___ dispatch method. Please refer to the official documentation to go more in depth [4].

6.4.6 Services

We created a file named Services.py located into the Service folder. A **service** represents an extra layer capable of promoting the **separation of concerns**. Other technologies, such as Spring Boot, introduce other layers and suggest how to structure a project to face the development in the right way.

In our project, we simply wanted to take into account the service layer to stress the role of controllers and repositories. Controllers are not aware of repositories. They have to dispatch the request and perform a response. In a *monolithic* code, a controller should dispatch the request, perform checks, convert data, connect to a database, store or retrieve data, check whether the operation was successfully executed, convert back data and perform a response. Such approach is not scalable and may easily lead to the project failure.

The service layer played a simple, but relevant role in the development of our Web application. A controller can ask for a service; data are passed to the service thanks to the model. The service interfaces with repositories performing several checks possibly. Finally, the service returns results to the controller. The controller creates a response by using data returned by the service. Let us give an example. A POST request to:

```
http://127.0.0.1:8080/api/gateways
```

is handled by the POST method of GatewayController. The page handler retrieves the request data as first step:

```
current = cherrypy.request.json
```

After retrieving data, the page handler creates a Gateway object, that is, the model:

```
gateway = Gateway(current['ip4'], current['ip6'])
```

Right now, we are in the page handler. Now, the service is involved:

```
gateway_id = self.service.addGateway(gateway)
```

The controller calls the GatewayService object to perform the addGateway service. We should clarify the role of the service. It depends. In our case, we could have simply checked whether a gateway was already existing, that is, a gateway with the same IP4 or IP6 address. Furthermore, we should have been sure that the resource was created if everything was fine. Since our application had experimental purposes, we decided not to implement these features. However, we created **pass-through** methods to promote the separation of concerns. In the following, we report the implementation of the GatewayService class:

```
class GatewayService(object):

    def addGateway(self, gateway):
        return DatabaseManager.createGateway(gateway)

    def getAllGateways(self):
        return DatabaseManager.readAllGateways()

    def getGatewayByID(self,id):
        return DatabaseManager.readGatewayByID(id)

    def updateGateway(self, gateway):
        return DatabaseManager.updateGateway(gateway)

    def deleteGateway(self, id):
        return DatabaseManager.deleteGateway(id)
```

6.5 Configuration

CherryPy comes with its own **configuration system** allowing you to parameterize the HTTP server as well as the behavior of the CherryPy engine when processing a Request-URI. The settings can be stored either in a text file with syntax close to the INI format or in a pure Python *dictionary* [46]. We decided to use **configuration files**.

We are now ready to describe the two lines of code we have skipped in section 6.4.1 about the Server.py file.

6.5.1 Global

To configure the CherryPy server instance, we created the Server.conf file whose content is reported in the following listing. The file is located into the Config folder. The global

section allows the user to perform global configurations.

```
[global]
server.socket_host="127.0.0.1"
server.socket_port=8080
serevr.api="api/gateways"
engine.autoreload.on=True
```

Listing 6.7: CherryPy: global configuration file.

We started defining the **host** and the **port** on which the server would have listened for incoming connections. It is usually a best practice to place these properties in a global configuration file. We also defined the API's base path. Then, we exploited the **autoreloader**, a *monitor* which restarts the process as soon as it detects a modification to a Python module imported by the application. Thanks to this feature, we could avoid stopping and restarting the server manually any time the code was changed. By default the monitor waits one second before checking for new changes; this value can be easily set by modifying the engine.autoreload_frequency option. For more details, please look at the documentation [4]. Actually, the server was stopped and restarted anytime we saved the code modifications.

After creating the file, we could update the global CherryPy configuration thanks to the following line of code:

```
1 cherrypy.config.update('Config/Server.conf')
```

6.5.2 Per application

To configure the CherryPy application instance, we created the Application.conf file whose content is reported in the following listing. Here as well, the file is located into the Config folder:

```
[/]
tools.sessions.on=False
tools.staticdir.root=os.path.abspath(os.getcwd())

[/static]
tools.staticdir.on=True
tools.staticdir.dir="./Static"

[/api/gateways]
request.dispatch= cherrypy.dispatch.MethodDispatcher()
tools.response_headers.on=True
tools.response_headers.headers=[('Content-Type', 'application/json')]
```

Listing 6.8: CherryPy: per application configuration file.

The root (/) section disabled the **session** support in our CherryPy application. We were not interested in the usual mechanism to use a session identifier that is carried during the conversation between the user and the application. Then, we indicated the **root directory** of all of our static contents. This must be an **absolute path** for security reason. CherryPy will complain if you provide only relative paths when looking for a match to your URLs.

In the static section, we indicated that all URLs which path segment started with /static would have been served as **static content**. We mapped that URL to the Static directory, as direct child of the root directory. The entire sub-tree of the Static directory would have

been served as static content. CherryPy mapped URLs to path within that directory. We will return to this topic in the next chapter.

Finally, in the /api/gateways section, we switched from the default mechanism of matching URLs to method for one that is aware of HTTP operations. This configuration was possible thanks to the MethodDispatcher instance. The dispatching was treated with the **REST spirit** starting from /api/gateways URI. Then, we forced the responses content-type to be application/json and we ensured that GET requests was responded to clients that accepted that content-type by having a Accept: application/json header set in their request.

As we have seen in section 6.4.1, we used the file to mount the application:

```
cherrypy.tree.mount(application, '/', config='Config/Application.conf')
```

6.6 Testing

We created the ServerTest.py file to test the Web application. In particular, we resorted to the **pytest** framework. This framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries [37].

Let us examine a couple of test examples taken from the file:

```
import requests

# Get all gateways
def test_getAllGateways():
    endpoint = "http://127.0.0.1:8080/api/gateways"
    response = requests.get(endpoint)
    assert response.status_code == 200

# Add a gateway 1
def test_addGateway1():
    endpoint = "http://127.0.0.1:8080/api/gateways"
    gateway = {"gateway_id": None, "ip4": "192.168.1.101", "ip6": "fe80::
    e48e:a027:12ce:df3a"}
    response = requests.post(endpoint, json=gateway)
    assert response.status_code == 201
```

We used the **requests** library to perform requests and get responses. Anytime you want to create a test to be run by pytest, you have to create a new function with a name that starts with test_ for pytest to understand that this is a test case. After defining the endpoint, you create a request and use **assertions** to evaluate the results. If the assertion holds, the test passes, otherwise it fails.

The first test performs a GET request to retrieve all the gateways. The expected status code is 200. The second adds a new gateway. You may notice that data are created by using the JSON format and passed to the request. The expected status code is 201 as the resource is created.

We simply had to run the following command to perform a complete test:

```
pytest -v ServerTest.py
```

The output showed a list of test rows each one displaying the test name, its status and the partial coverage percentage. The status could be PASSED or FAILED. In case of failures, a report was displayed to target the problem.

We would like to conclude this chapter with a final remark. During the implementation of our Web service, we constantly use a very popular tool called *Postman*. **Postman** is an API platform for building and using APIs. It simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs faster [33]. You can easily download and start using the tool in a bunch of seconds. As a try, after creating several resources, by selecting the GET method and targeting 127.0.0.1:8080/api/gateways, you should get something like the following outcome:


```
[ {
    "dataloggers": [
      {
        "measures": [
          {
            "id": 1,
            "temperature": "21.5",
            "humidity": "50.1",
            "timestamp": null
          }
        ],
        "id": 1,
        "serial": "1_1"
      },
      {
        "measures": [
          {
            "id": 2,
            "temperature": "21.5",
            "humidity": "50.1",
            "timestamp": null
          }
        ],
        "id": 2,
        "serial": "1_2"
      }
    ],
    "id": 1,
    "ip4": "192.168.1.101",
    "ip6": "fe80::e48e:a027:12ce:df3a"
  },
  {
    "dataloggers": [
      {
        "measures": [],
        "id": 3,
        "serial": "2_1"
```

```
      }
    ],
    "id": 2,
    "ip4": "192.168.1.102",
    "ip6": null
  },
  {
    "dataloggers": [
      {
        "measures": [],
        "id": 4,
        "serial": "3_1"
      }
    ],
    "id": 3,
    "ip4": "192.168.1.103",
    "ip6": null
} ]
```

Data are represented in JSON format with the expected hierarchy:

```
-- gateway_1
  -- datalogger_1
    -- measure_1_1
    ..
    -- measure_1_j
-- gateway_2
  -- datalogger_2_1
  ..
  -- datalogger_2_k
    -- measure_2_k_1
    ..
-- gateway_n
```

Listing 6.9: Application data in JSON format.

In this chapter we have explained how we implemented a simple REST Web service capable of managing and storing data provided by the datalogger and gateway. Our goal was to create a reliable software component that met our specifications.

Chapter 7

Gateway

In this chapter, we will cover several topics. After introducing the LoRaWAN® protocol and the hardware module we selected from the market, we will explain how we implemented a software component capable of dealing with datalogger's incoming messages and performing HTTP requests, as a client, to the Web service. Finally, a section will be devoted to how to reach the Web service thanks to a Browser.

7.1 LoRa® and LoRaWAN®

As we widely discussed in the previous chapters, a datalogger should be able to send measures to a *gateway*. We also introduced the main features of the datalogger: batteryless and intermittent. Furthermore, the distance between devices had to be taken into account. We mainly focused on solutions that were:

- low power;
- long range.

Finally, we anticipated that the LoRaWAN® technology was selected. In the following, we will present differences and relationships between LoRa® and LoRaWAN® technologies as well as the role of a *gateway* in this scenario.

7.1.1 LoRa®

Long Range (LoRa®) by Semtech Corporation, is a wireless proprietary long-range and low-power wide area network (LPWAN) technology. It represents the **physical layer** of a LoRaWAN® network. Its role is to manage the modulation, power, receiver and transmission radios, and signal conditioning.

LoRa® uses frequency bands in industrial, scientific, and medical (ISM) **license-free** space which depend on the physical location [32]:

- 915 MHz - in the USA with power limits, but no duty cycle limit;
- 868 MHz - in Europe with a 1 % and 10 % duty cycle;

- 433 MHz - in Asia.

This technology can also work at 2.4 GHz to achieve higher data rates compared to sub-gigahertz bands, at the cost of range [31].

The modulation technique is derived from **chirp spread spectrum (CSS)** technology with the aim of balancing data rate with sensitivity in a fixed channel bandwidth. Basically, this technology encodes the information on radio waves using **chirp pulses** or **symbols**, which are sinusoidal waves that increase or decrease over time [32]. The modulated transmission is robust against disturbances and can be received across great distances. Under optimal line-of-sight conditions, a device adopting the LoRa® technology can communicate up to **10 km** with low-power consumption.

The chirp rate and the symbol rate affect the **bitrate** R_b. The bitrate ranges from 0.25 kbit s^{-1} to 22 kbit s^{-1} [41] and can be calculated as [32]:

$$R_b = SF \cdot \frac{B}{2^{SF}} \ [bps] \tag{7.1}$$

where SF is the **spreading factor** and B is the **bandwidth**.

In general, LoRa® is ideal for applications that transmit small chunks of **data** with low bit rates [31]. Data are encoded using the increasing or decreasing frequency rate and multiple transmissions can be sent with different data rates on the same frequency. Multiple sub-bands subdivide the band. In practice, LoRa® uses 125 kHz **channels** and dedicates six 125 kHz channels and pseudorandom channel hopping. Each **frame** will be transmitted with a specific spreading factor whose increment will slow the transmission, but will lead to a longer transmission range. The frames in LoRa® are orthogonal, meaning multiple frames can be sent simultaneously as long as each is sent with a different spreading factor. In total, there are six different spreding factors ranging from 7 to 12. The typical LoRa® **packet** contains a preamble, a header and 51 to 222 bytes payload [32].

A powerful feature of LoRa® networks, called **adaptive data rate (ADR)**, enables dynamically scalable capacity based on the density of nodes and infrastructure. The idea is to set nodes that are close to the base station to a higher data rate because of signal confidence. Nodes in close proximity can transmit data, release their bandwidth, and enter a sleep state quickly versus distant nodes which transmit at slower rates. ADR is controlled by the network management in the cloud [32].

Finally, table 7.1 reports further information about the **uplink** and **downlink** features.

Feature	**Uplink**	**Downlink**
Modulation	CSS	CSS
Link budget	156 dB	164 dB
Bitrate (adaptive)	0.3 kbit s^{-1} to 5 kbit s^{-1}	0.3 kbit s^{-1} to 5 kbit s^{-1}
Message size per payload	0 to 250 bytes	0 to 250 bytes
Message duration	40 ms to 1.2 s	20 ms to 160 ms
Energy spent per message	11 µA h (at full sensitivity)	0.5 µA h

Table 7.1: LoRa® uplink and downlink features [32].

Obviously, we cannot cover an exhaustive and detailed description of LoRa® technology. We simply introduced the main features and relevant information that the reader might take into account for a more in-depth study.

7.1.2 LoRaWAN®

LoRaWAN® is a media access control (MAC) layer protocol built on top of LoRa® technology [31]. It adds proper network protocols to manage traffic for data collection and endpoint device management.

Unlike LoRa®, LoRaWAN® is an open, cloud-based protocol designed and maintained by the LoRa Alliance®, whose goal is to enable devices to communicate wirelessly. Node authentication and data encryption for security are also incorporated.

There are three **MAC protocols** that are part of the data link layer. They balance latency with energy usage [32]:

- class A - is the best trade-off between power consumption and latency;
- class B - balances power and latency;
- class C - has the minimum latency but the highest power consumption.

Data encryption is performed resorting to the AES128 model to ensure **security**. In particular, LoRaWAN® separates *authentication* and *encryption* by using *keys* [32]:

- a network session key NwkSKey - for authentication;
- an application session key AppSKey - for user data.

Devices must send a **JOIN request** to join a LoRaWAN® network. In particular, the following sequence of steps describes the **over-the-air-activation (OTAA)** process:

1. an end device sends a JOIN request;
2. a gateway will respond and assign a dynamic device address;
3. application and network session keys will be negotiated during the process.

Alternatively, an end device can use the **activation by personalization (ABP)**. In this case, a LoRaWAN® carrier/operator pre-allocates 32 bits of network and session keys. A customer will purchase a connectivity plan and obtain a set of keys from an endpoint manufacturer with the keys burned into the device (hardcoding) [32]. In other words, any end device must perform the **activation** to be registered with a network before sending and receiving messages. OTAA is recommended as more secure than ABP. Furthermore, ABP has the downside that devices can not switch network providers without manually changing keys in the device [31].

From a technical point of view, LoRaWAN® is an asynchronous, **ALOHA-based** protocol. The ALOHA protocol allows clients to transmit messages with no arbitration. There are no reservations or multiplexing techniques. A *hub* immediately retransmits packets it has received. If an *endpoint* notices one of its packets was not acknowledged, it will wait and then retransmit the packet. In a LoRaWAN® network, gateways play the role of hubs

working as data concentrators, whereas end devices are endpoints. Collisions only occur if transmissions use the same channels and spreading factor [32].

LoRaWAN® networks are deployed in a **star-of-stars topology** as shown in figure 7.1. Main elements are [31]:

- **end devices** - autonomous electronic systems hosting sensors and actuators, able to send LoRa® modulated wireless messages to *gateways* or receive messages wirelessly back from *gateways*;
- **gateways** - devices working as data concentrators able to receive messages from end devices and forward them to the *network server*. Each gateway must be registered using configuration settings to a LoRaWAN® *network server*. Moreover, a gateway can be **indoor** or **outdoor**;
- **network server** - a software component running on a server that manages the entire network;
- **application server** - a software component running on a server that securely processes application data;
- **join server** - a software component running on a server that processes join-request messages sent by end devices.

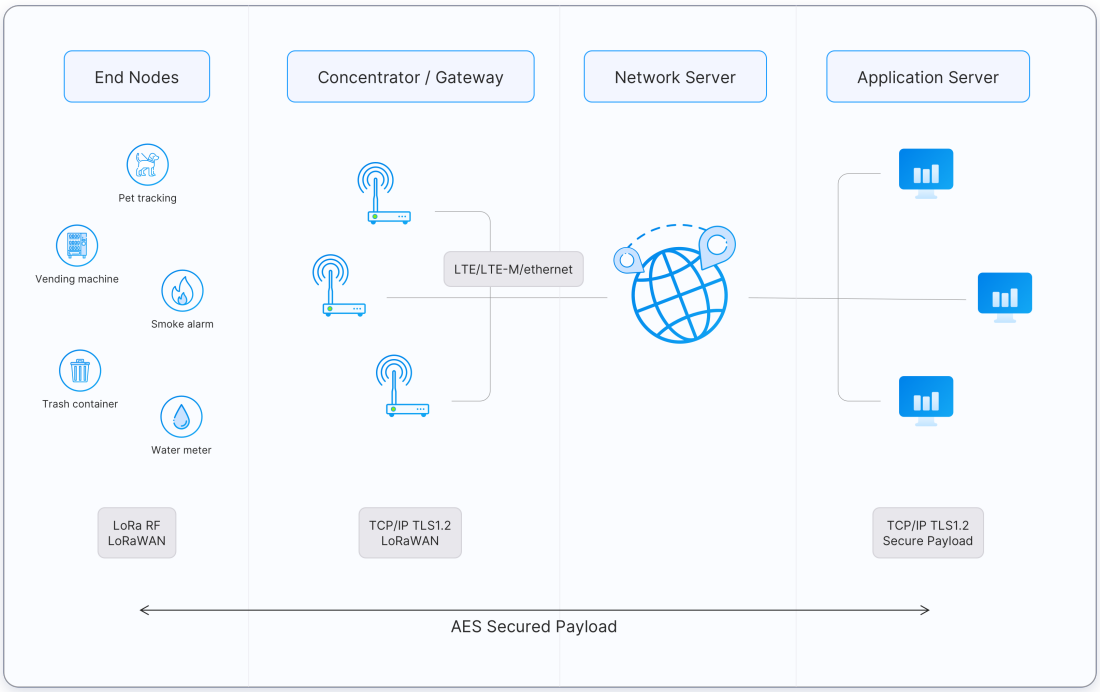


Figure 7.1: A typical LoRaWAN® network architecture [31].

Essentially,

> "end devices communicate with nearby gateways and each gateway is connected to the network server. Thanks to the ALOHA based protocol, end devices do not need to peer with specific gateways. When a end device sends a message, nearby gateways within range forward the message to the network server. If the latter receives multiple copies of the same message, it keeps a single copy of the message and discards others (de-duplication). The network server also performs end devices authenticity validation, integrity of messages validation, device address checking and selection of the best gateway for routing downlink messages. The application server processes application-specific data messages received from end devices. It also generates all the application-layer downlink payloads and sends them to the connected end devices through the network server. In general, a LoRaWAN® network can have more than one application server. The collected data can be interpreted by applying techniques like *machine learning* and *artificial intelligence* to solve business problems. Finally, the join server assists in secure device activation, root key storage, and session key generation. If an end device sends a JOIN request message, the network server delegates the join server to process the request. If successfully processed, it generates session keys, and transfers NwkSKey and AppSKey to the network server and the application server respectively"[31].

To sum up, a LoRaWAN® network involves several elements enabling a scalable architecture. Such scalability can be exploited by companies providing their services with the aim of reaching the required capillarity. In general, a company should specify, design, implement and deploy LoRaWAN® compliant end devices. The number of units depends on the target application. Then, the company should deploy indoor or outdoor gateways, working as data concentrators, which receive messages from end devices and forward them to the network server. The number of gateways is a critical choice. The goal is to cover the physical areas of interest by taking into account many parameters such as distance and transmission power. A network server manages the entire network, although complex applications may require more than one to perform load balancing.

7.2 A crucial choice

A relevant aspect concerns the **implementation** of a gateway. As we have seen in the previous section, a gateway works as a data concentrator and can serve thousands of end devices. Such feature suggests that the implementation is not so straightforward.

More in detail, a gateway [31] is a *router* equipped with a hardawre unit implementing a LoRa® concentrator, allowing it to receive LoRa® packets. Furthermore, two solutions are basically adopted:

- running a **minimal firmware** to target a low-cost and easy to use software implementation. The firmware mainly focuses on packet forwarding;
- running an **operating system** that performs packet forwarding as background program, but gives more liberty to the gateway administrator to manage it and to install their own software.

Another relevant aspect is about **channels**. Experts recommend not to use single-channel gateways. Such solution is low-cost, but they can receive LoRa® packets on a

specific spreading factor and channel, and exchange them back and forth with the network. Moreover, single-channel gateways are not LoRaWAN® compliant and only offer poor coverage. In practice, a multi-channel gateway is preferable; you can buy a ready-to-use solution or build your own one which are beneficial to the whole community. The reader should also be aware that LoRaWAN® operates on unlicensed bands, so in most countries, running your own gateway is completely legal. However, some countries may apply restrictions such as where to install antennas. If you have doubts, it is better to reach out your local communities [31].

Commercial LoRaWAN® gateways are expensive. In addition, implementing a gateway from scratch would have required a huge amount of work and results would have been poor compared to a commercial solution. That's why we decided to implement a **pseudo** gateway capable of receiving messages from dataloggers and forward them to the REST Web service. We mainly focused on the batteryless datalogger to understand whether it had been able to communicate over the LoRaWAN® network. In particular, as we will see in the next sections, we selected another transceiver from the market with the goal of using the **TEST** operating mode. Basically, the idea was to create a **point-to-point** connection between the batteryless datalogger and the pseudo gateway as TEST operating mode allows users to perform radio frequency tests quickly without any knowledge of LoRa® chip.

This crucial choice does not affect the experimental work, but represents a starting point to achieve the main goal: joining a LoRaWAN® network via **OTAA** operating mode. We point out that the datalogger is LoRaWAN® compliant as it already implements the JOIN procedure, but we could not test it. We will return on this topic in section 10.1.2.

7.3 The transceiver

To implement the hardware component of the gateway, we bought the **Wio-E5 mini** development board by **seeed studio** shown in figure 7.2. According to the manufacturer,

> "Wio-E5 mini is a compacted-sized development board suitable for the rapid testing and building of small-size prototyping and helps you design your ideal LoRaWAN® wireless IoT device with a long-distance transmission range"[42].

Wio-E5 mini is embedded with the same chip we presented in chapter 4.4. It leads out all GPIOs of Wio-E5 STM32WLE5JC, including UART, ADC, SPI and I^2C. It also mounts the *reset* and *boot* buttons.

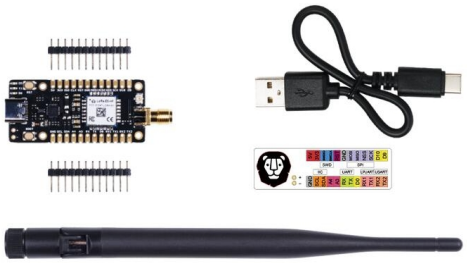


Figure 7.2: Wio-E5 mini development board [42].

The module can be easily connected to a desktop or laptop computer via USB cable. Such feature allowed us to immediately start developing the software component capable of interfacing and controlling the module.

7.3.1 AT commands

Both the datalogger's communication module and the gateway's transceiver can be easily controlled through **AT Commands**. This topic was briefly mentioned in section 5.3.2, but now we will go more in depth. In particular, the datalogger's module can be controlled via UART connection, whereas the gateway's transceiver can be controlled via USB. We will report here data that were useful in the project by referring to AT Command Specification [41].

A **command** is *case insensitive* and its length never exceeds total **528** characters. One valid command must end with \n; \r\n is also valid. All commands have **response**. Default UART configuration is: 9600, 8, n, 1, that is, 8 bits data, no parity, 1 stop bit. The very basic command tests whether connection of module is OK:

```
AT\r\n
```

The response should be:

```
+AT: OK\r\n
```

Table 7.2 reports the list of basic AT Commands:

Command	Comment
AT+ID	Read all, DevAddr(ABP), DevEui(OTAA), AppEui(OTAA)
AT+ID=DevAddr	Read DevAddr
AT+ID=DevEui	Read DevEui
AT+ID=AppEui	Read AppEui
AT+ID=DevAddr,"devaddr"	Set new DevAddr
AT+ID=DevEui,"deveui"	Set new DevEui
AT+ID=AppEui,"appeui"	Set new AppEui
AT+KEY=APPKEY,"16 bytes length key"	Change application session key
AT+DR=band	Change the Band Plans
AT+DR=SCHEME	Check current band
AT+CH=NUM, 0-7	Enable channel 0 7
AT+MODE="mode"	Select work mode: OTAA, ABP or TEST
AT+JOIN	Send JOIN request
AT+MSG="Data to send"	Send string format frame (no need server confirmation)
AT+CMSG="Data to send"	Send string format frame (must be confirmed by the server)
AT+MSGHEX="xx xx xx xx"	Send hex format frame (no need server confirmation)
AT+CMSGHEX="xx xx xx xx"	Send hex format frame (must be confirmed by the server)

Table 7.2: Basic AT Commands [41].

7.3.2 Errors

Table 7.3 shows the **error code** list applying to all LoRaWAN® supported commands. The user should refer to this list to know what is happening to LoRaWAN® module, when gets errors.

Code	Comment
-1	Parameters is invalid
-10	Command unknown
-11	Command is in wrong format
-12	Command is unavailable in current mode
-20	Too many parameters (max 15)
-21	Length of command is too long (exceed 528 bytes)
-22	Receive end symbol timeout, command must end with <LF>
-23	Invalid character received
-24	Either -21, -22 or -23

Table 7.3: Error code list [41].

7.3.3 Operating modes

There are three operating modes: **ABP**, **OTAA** and **TEST**. A LoRaWAN® module can only work with one mode at a time. By default, ABP is enabled, all test commands are unavailable, LoRaWAN® will return error -12 if it receives test command in non-TEST mode.

AT+MODE command will reset the LoRaWAN® stack when first enter ABP/OTAA mode and reset LoRa® chip when first enter TEST mode.

ABP/OTAA mode status is remembered by LoRaWAN® module; each time it starts, it will enter previous working mode before reset or re-power.

7.3.4 LoRaWAN® over-the-air-activation (OTAA)

This mode is used to **join** a LoRaWAN® network. First of all, the user should perform this command sequence to get the **device EUI**, the **App EUI** and set the **App Key**:

```
AT+ID=DevEui\r\n
AT+ID=AppEui\r\n
AT+KEY=APPKEY,"2B7E151628AED2A6ABF7158809CF4F3C"\r\n
```

In practice, the first two commands check the **ID** of the LoRaWAN® module.

The next step is to set up the **datarate** which depends on the band plans. For Europe:

```
AT+DR=EU868\r\n
```

Then, you have to enable **channels**. For Europe, you should enable channels 0, 1 and 2; others are disabled:

```
AT+CH=NUM,0-2\r\n
```

Then, you need to set up the operating **mode**:

```
AT+MODE=LWOTAA\r\n
```

Finally, you can **join**:

```
AT+JOIN\r\n
```

If the end device successfully joined the network, you should have a response like the following:

```
+JOIN: Start
+JOIN: NORMAL
+JOIN: Network joined
+JOIN: NetID 000013 DevAddr 26:01:5F:66
+JOIN: Done
```

After joining a network, you are allowed to **send** TEXT:

```
AT+MSG=HELLO\r\n
```

or HEX messages:

```
AT+MSGHEX="00 11 22 33 44"\r\n
```

7.3.5 Test mode (TEST)

TEST command is not like other commands, as it includes several sub-commands as you can see in table 7.4. In TEST mode, users can perform a radio frequency (RF) test quickly without any knowledge of LoRa® chip. Commands which are related to RF configuration are disabled in TEST mode.

Sub-Command	Comment
STOP	Set LoRaWAN® Modem to TEST stop mode
TXCW	Transmit continuous wave
TXCLORA	Transmit continuous LoRa® signal
RFCFG	Set RF configuration in TEST mode
RXLRPKT	Continuous receive pure LoRa® packet
TXLRPKT	Send one HEX format packet out
TXLRSTR	Send one string format packet
RSSI	Get RSSI value of specified channel
LWDL	Send LoRaWAN® downlink packet, useful tool to test CLASS C device

Table 7.4: TEST mode sub-command list [41].

Before using any TEST command, the LoRaWAN® module must enter the TEST mode, or error code -12 will be reported. To enter TEST **mode**:

```
AT+MODE=TEST\r\n
```

The response should be:

```
+MODE: TEST\r\n
```

After entering TEST mode, you might **query** TEST mode with the goal of checking the RF configuration:

```
AT+TEST=?\r\n
```

Then, you should set up the **RF configuration**. As an example:

```
AT+TEST=RFCFG,866,SF12,125,12,15,14,ON,OFF,OFF\r\n
```

Parameters are: frequency, spreading factor, bandwidth, TX preamble, RX preamble, power, CRC, inverted IQ and public LoRaWAN®.

Now, it depends whether you want to send or receive LoRa® Packets. To **send** TX LoRa® Packet, you can use the HEX format:

```
AT+TEST=TXLRPKT, "HEX String"\r\n
```

or the TEXT format:

```
AT+TEST=TXLRSTR, "TEXT"\r\n
```

To **receive** RX LoRa® Packet in continuous RX mode, simply:

```
AT+TEST=RXLRPKT\r\n
```

Finally, you might want to stop the LoRaWAN® module operating in TEST mode:

```
AT+TEST=STOP\r\n
```

7.4 Developed code

In this section, we will describe the code with the usual by construction approach. We will refer to files located into the Gateway folder. Please refer to section A.2.3 to check whether all the required modules have been successfully installed.

7.4.1 Entry point to the application

We started the software implementation by creating the Gateway.py file. Let us have a look at main method:

```
1  # Entry point to the application
2  def main():
3
4    # Create a RESTClient object
5    restClient = RESTClient()
6
7    # Create a SerialCommunicationHandler object
8    serialCommunicationHandler = SerialCommunicationHandler()
9
10   # Create the Gateway object
11   gateway = Gateway(serialCommunicationHandler, restClient)
12
13   # Run the gateway
14   gateway.start()
```

Listing 7.1: Gateway: entry point.

The method represents the entry point to the application and starts thanks to the standard Python checking on whether the file is used directly or as a module:

```
if __name__ == '__main__':
  main()
```

Its implementation is quite simple as basically deals with three classes:

- the **Gateway** class defined in the same module;
- the imported **SerialCommunicationHandler** class;
- the imported **RESTClient** class.

After instantiating a RESTClient and a SerialCommunicationHandler object, the method instantiates a Gateway object capable of taking the control and running the previous ones. Such activity is performed by calling the start method.

Here is the Gateway class:

```
class Gateway(object):

  def __init__(self, serialCommunicationHandler, restClient):
    self.serialCommunicationHandler = serialCommunicationHandler
    self.restClient = restClient

  def start(self):
    # Check whether the Web servise is reachable
    if (self.restClient.testWS() == False):
      logging.error('Could not reach Web Service: ' + server_host + ':' +
     server_port + '/' + server_api)
      return
    else:
      logging.info('Web Service successfully reached: '+ server_host + ':
    ' + server_port + '/' + server_api)

    # Check whether the selected serial port can be opened
    if (self.serialCommunicationHandler.openPort() == False):
      logging.error('Could not open serial port: ' + serial_port)
      return
    else:
      logging.info('Serial port successfully opened: ' + serial_port)
      self.run()

  def run(self):

    # Register the RESTClient object as observer
    self.serialCommunicationHandler.register(self.restClient)

    # Run the REST client
    self.restClient.run()

    # Run the serial communication handler
    self.serialCommunicationHandler.run()un()
```

Instance variables are set by the constructor, but the key role is played by the start method. Indeed, the start method **checks** whether the Web service is reachable and the selected serial port can be opened. If checks are successfully verified, it calls the run method whose

goal is to register the RESTClient object as an *observer*, run the REST client and run the serial communication handler in this order.

Another point is about **logging**. Logging is an useful and precious activity helping developers anytime. A logger was set in each class thanks to the following piece of code:

```
logging.basicConfig (
  level=logging.INFO,
  format="[%(levelname)s %(asctime)s] %(message)s",
  datefmt="%Y/%m/%d %I:%M:%S %p",
  handlers=[logging.FileHandler("Gateway.log"), logging.StreamHandler()]
)
```

The logger defines the level, message and date format. We wanted to save log messages into the Gateway.log file by creating a handler as well as logging on the active terminal. As an example, here we report log messages that were created after running the application. The reader may notice that the application was waiting for incoming messages from dataloggers:

```
[INFO 2025/04/03 08:34:43 AM] Web Service successfully reached: ...
[INFO 2025/04/03 08:34:43 AM] Serial port successfully opened: ...
[INFO 2025/04/03 08:34:43 AM] Sent command: AT
[INFO 2025/04/03 08:34:43 AM] Response: +AT: OK
[INFO 2025/04/03 08:34:43 AM] Sent command: AT+UART=BR
[INFO 2025/04/03 08:34:43 AM] Response: +UART: BR, 57600
[INFO 2025/04/03 08:34:43 AM] Sent command: AT+MODE=TEST
[INFO 2025/04/03 08:34:43 AM] Response: +MODE: TEST
[INFO 2025/04/03 08:34:43 AM] Sent command: AT+TEST= ...
[INFO 2025/04/03 08:34:43 AM] Response: +TEST: ...
[INFO 2025/04/03 08:34:43 AM] Sent command: AT+TEST=RXLRPKT
[INFO 2025/04/03 08:34:43 AM] Response: +TEST: RXLRPKT
```

7.4.2 REST client

We created a RESTClient.py file to perform HTTP requests to the Web service. Its main role was to *consume* the service as a **REST client**.

The constructor defines instance variables whose purpose will be explained later on:

```
class RESTClient(object):

  def __init__(self):
    self.gateway_endpoint = None
    self.datalogger_endpoint = None
    self.gateways = None
    self.dataloggers = None
```

The testWS method tests the Web service reachability and retrieves the available data:

```
def testWS(self):
  endpoint = 'http://%s:%s/%s' %(server_host, server_port, server_api)
  response = requests.get(endpoint)
  if response.status_code == 200:
    # Retrieve the data
    self.gateways = response.json()
    return True
  else:
    return False
```

As we have seen in the previous section, this method is called by Gateways.start method with the aim of checking whether the Web service is reachable. A failure will prevent the application to run. Moreover, it retrieves all the available data from the Web service which will be saved into the gateways variable.

The run method checks whether the gateway is already registered to the service:

```
def run(self):

  # Check whether this gateway is already registered
  registered = False
  for gateway in self.gateways:
    if (gateway['ip4'] == gateway_host):
      # Save the gateway endpoint
      self.gateway_endpoint = 'http://%s:%s/%s/%s' %(server_host,
   server_port, server_api, gateway['id'])
      # Save the list of dataloggers managed by this gateway
      self.dataloggers = gateway['dataloggers']
      registered = True
      break

   # Register this gateway
   if not registered:
      endpoint = 'http://%s:%s/%s' %(server_host, server_port, server_api
   )
      gateway = {"gateway_id": None, "ip4": gateway_host, "ip6": None}
      response = requests.post(endpoint, json=gateway)
      if response.status_code == 201:
        self.gateway_endpoint = response.headers['Location']
        logging.info('Gateway successfully created. Endpoint is: ' + self
   .gateway_endpoint)
```

By scanning the list of registered gateways, the method checks the presence of the candidate. In case of matching, it saves its endpoint as well as the list of its dataloggers thanks to dataloggers variable. If the gateway is not present in the list, it creates a new one in order to be registered.

The update method is notified by the SerialCommunicationHandler object as we will see later on:

```
def update(self, message):

  # Parse message
  measure = self.parse(message)

  if measure is not None:
    # Format the measure
    measure = self.formatMeasure(measure)
    # Send measure to server
    self.sendMeasure(measure)
```

Anytime a message is notified, it is *parsed*. We were interested in incoming measures; all the other messages were discarded. If a measure is recognized, it is initially formatted and successively sent to the server.

The parser method searches for RX LoRaWAN® packets such as the following, which represents a measure sent by a datalogger:

```
+TEST: RX "00894205702864243083404 0"
```

Here is its implementation:

```
def parse(self, message):
  message = message.decode('utf-8')
  message = re.split("\s", message)
  if message[1] == "RX":
    temp = message[2].replace("\"", "")
    return temp
  else:
    return None
```

The parser returned:

```
00894205702864243083404 0
```

After paring, we had to format the measure. This is the role of formatMeasure method:

```
def formatMeasure(self, measure):
  n = [14, 5, 5]
  measure = [measure[sum(n[:i]):sum(n[:i+1])] for i in range(len(n))]
  measure[1] = "{:.2f}".format(int(measure[1]) / pow(2,16) * 165 - 40)
  measure[2] = "{:.2f}".format(int(measure[2]) / pow(2,16) * 100)
  return measure
```

Since the format was decided a priori, the function starts creating a list of items:

```
00894205702864 24320 34164
```

Then, it applies formulas we took from the datasheet of the HDC1000 sensor [17] in order to convert raw data. As an example, the previous raw data are converted as:

```
00894205702864 21.23 51.94
```

that is, the serial number, temperature and humidity. A list with converted data is returned to the caller.

The last step was to send the measure to the service. This activity is performed by the sendMeasure method:

```
def sendMeasure(self, measure):

  # Get the gateway
  endpoint = '%s' %(self.gateway_endpoint)
  response = requests.get(endpoint)
  if response.status_code == 200:
    # Retrieve the data
    gateway = response.json()
    self.dataloggers = gateway['dataloggers']
  else:
    return False

  # Check wheter this datalogger is already registered
  registered = False
  for datalogger in self.dataloggers:
    if (datalogger['serial'] == measure[0]):
      # Save the datalogger endpoint
      self.datalogger_endpoint = '%s/dataloggers/%s' %(self.
   gateway_endpoint, datalogger['id'])
```

```
      registered = True
      break

  # Register this datalogger
  if not registered:
    endpoint = '%s/dataloggers' %(self.gateway_endpoint)
    datalogger = {"datalogger_id": None, "serial": measure[0]}
    response = requests.post(endpoint, json=datalogger)
    if response.status_code == 201:
      self.datalogger_endpoint = response.headers['Location']
      logging.info('Datalogger successfully created. Endpoint is: ' +
   self.datalogger_endpoint)

  # Send the measure
  endpoint = '%s/measures' %(self.datalogger_endpoint)
     current_datetime = datetime.now()
  timestamp = current_datetime.timestamp()
  timestamp = datetime.fromtimestamp(timestamp).strftime("%Y-%m-%d %H:%M
    :%S")
  measure = {"measure_id": None, "temperature": measure[1], "humidity":
    measure[2], "timestamp": timestamp}
  response = requests.post(endpoint, json=measure)
  if response.status_code == 201:
    logging.info('Sent measure to: ' + endpoint)
```

After retrieving the gateway to be updated, we checks whether the current datalogger is already registered. If so, we are done, otherwise the datalogger is registered to the service. Finally, the measure is sent to the service. This method correctly sets up endpoints with the goal of performing the right operation. As an example, after sending a measure, we got the following log:

```
Sent measure to: http://127.0.0.1:8080/api/gateways/1/dataloggers/1/
    measures
```

7.4.3 Serial communication handler

We created a SerialCommunicationHandler.py file to implement a piece of code to handle incoming messages from a selected serial port and *dispatch* them to objects of interest. First of all, let us focus on two main aspects:

```
class SerialCommunicationHandler(object):

  serialPort = serial.Serial (
    port=None,
    baudrate=57600,
    bytesize=serial.EIGHTBITS,
    parity=serial.PARITY_NONE,
    stopbits=serial.STOPBITS_ONE,
    timeout=None)

  def __init__(self):

    # Creaate a Serial object
    self.serialPort.port = port
```

```
15
16      # Create an empty observer list
17      self.observers = set()
```

The constructor sets up a Serial object and an empty list of **observers**. The Serial class is imported from the serial library. Section A.2.3 clarifies the user should install the **pyserial** library thanks to the pip tool. The Serial object sets up the *baudrate*, *bytesize*, *parity*, *stop bit* and *timeout*. Actually, the port to be used is read from the configuration file and set in the constructor. The list of observers is used to implement the **observer** pattern. The SerialCommunicationHandler object is **observable** and can notify its observers.

We needed three methods in the class to implement the pattern:

```
1 def register(self, observer):
2   self.observers.add(observer)
3
4 def unregister(self, observer):
5   self.observers.discard(observer)
6
7 def dispatch(self, responce):
8   for observer in self.observers:
9     observer.update(responce)
```

The register method allows to add a new observer in the list. As we have seen before, it is called by Gateway.run method to register the REST client. The unregister method is its counterpart; we did not use it, but it was implemented for the sake of completeness. The dispatch method has the main role. It notifies the LoRaWAN® response or the incoming message to each registered observer. The crucial point is that observers apply their own business to the dispatched message. The observable simply notifies its observers.

A brief parenthesis about the serial port connection. We wrote the openPort method to perform this activity. It calls the open method of the Serial object and returns the result:

```
1 def openPort(self):
2   try:
3     self.serialPort.open()
4   except:
5     return False
6   else:
7     return True
```

The method is called by Gateway.start method when the application starts.

The run method takes the control of the SerialCommunicationHandler object:

```
1  def run(self):
2
3   command = 'AT'
4   self.sendCommand(command)
5   response = self.serialPort.readline().rstrip()
6   logging.info('Response: ' + response.decode('utf-8'))
7
8   command = 'AT+UART=BR'
9   self.sendCommand(command)
10  response = self.serialPort.readline().rstrip()
11  logging.info('Response: ' + response.decode('utf-8'))
12
13  command = 'AT+MODE=TEST'
```

```
14    self.sendCommand(command)
15    response = self.serialPort.readline().rstrip()
16    logging.info('Response: ' + response.decode('utf-8'))
17
18    command = 'AT+TEST=RFCFG,%s,%s,%s,%s,%s,%s,%s,%s' %(frequency,
        spreadingFactor, bandwidth, txPreamble, rxPreamble, power, crc,
        invertedIQ, public)
19    self.sendCommand(command)
20    response = self.serialPort.readline().rstrip()
21    logging.info('Response: ' + response.decode('utf-8'))
22
23    command = 'AT+TEST=RXLRPKT'
24    self.sendCommand(command)
25    response = self.serialPort.readline().rstrip()
26    logging.info('Response: ' + response.decode('utf-8'))
27
28    while 1:
29      response = self.serialPort.readline().rstrip()
30      logging.info('Response: ' + response.decode('utf-8'))
31      self.dispatch(response)
```

After performing the set of commands to work in TEST mode and starting receiving LoRaWAN® packets, it enters a while loop waiting for responses and messages. As we have learned in section 7.3.1, each command has a response, whereas a message is asynchronous, such as an incoming measure by a datalogger. Anyhow, either responses and messages are dispatched to observers.

The last point concerns commands. A command can be easily sent thanks to the sendCommand method:

```
1 def sendCommand(self, command):
2   self.serialPort.write(bytes(command, 'utf-8') + bytes('\r\n', 'utf-8'))
3   logging.info('Sent command: ' + command)
```

The serial port is written by converting the string to utf-8 format and adding the required \r\n characters with the goal of creating a valid command.

7.5 Configuration

We created the **Gateway.conf** file to configure the software component. Here is its content:

```
[serial_port]
serial_port.port="/dev/ttyUSB0"

[LoRaWAN]
LoRaWAN.frequency=866
LoRaWAN.spreding_factor=SF7
LoRaWAN.bandwidth=125
LoRaWAN.tx_preamble=12
LoRaWAN.rx_preamble=15
LoRaWAN.power=14
LoRaWAN.crc=ON
LoRaWAN.inverted_iq=OFF
LoRaWAN.public=OFF
```

```
[gateway]
gateway.socket_host="127.0.0.1"

[server]
server.socket_host="127.0.0.1"
server.socket_port=8080
server.api="api/gateways"
```

Listing 7.2: Gateway: configuration file.

The serial_port section allowed us to select a **serial port**. The implementation was performed on a Linux machine, but the port name can be easily configured for other operating systems.

The LoRaWAN section allowed us to set the **radio frequency** configuration for the LoRaWAN® transceiver working in TEST mode. Some parameters must match with those of a datalogger.

The gateway section allowed us to decide the **IP address** to use for the gateway. Such configuration could be done at run time, but we preferred to apply this approach to bypass all the tricky operations to scan the available interfaces and find actual IP4 or IP6 addresses.

The server section allowed us to set host and port of the **server** running the Web service as well as its base path.

Similar configurations were also described in section 6.5 with the aim of providing a bit of flexibility to the project. You can run both the service and gateway on different machines, thus, you are not constrained on running the entire application on localhost.

7.6 Testing

We created the GatewayTest.py file to test the gateway. Actually, we were simply interested in ensuring that the Web service was reachable and the serial port was opened. We resorted to the **pytest** library as we did for testing the Web service. Here is the code:

```
1  import pytest
2
3  from RESTClient import RESTClient
4  from SerialCommunicationHandler import SerialCommunicationHandler
5
6  client = RESTClient()
7  serialCommunicationHandler = SerialCommunicationHandler()
8
9  # Test Web service - run the service before running the test
10 def test_WebService():
11   reachable = client.testWS()
12   assert reachable == True
13
14 # Test serial communication port
15 def test_serialCommunication():
16   opened = serialCommunicationHandler.openPort()
17   assert opened == True
```

Such tests might seem weird, but the gateway cannot work if any of these resources are not available. It is like when you power a device and nothing works, but then you realize the power cable is not plugged.

7.7 Browser

Before concluding this chapter, we would like to present how we met FR4 class functional requirements. Indeed, we wanted to allow a user to access data collected by the application thanks to a *Web Browser*.

A **Web Browser (browser)** is an application for accessing websites and the Internet. When a user requests a Web page from a particular website, the browser retrieves its files from a Web server and then displays the page on the user's screen. There are many browsers out there capable of dealing with on a range of devices, including desktops, laptops, tablets and smartphones. The user have simply to choose the preferred one.

Let us examine a piece of code taken from Server.py file located into the Server folder and focus on page handlers that manage **static** contents:

```
class Application(object):

  @cherrypy.expose
  def index(self):
    return open('Static/views/index.html')

  @cherrypy.expose
  def user(self):
    return open('Static/views/user.html')

class API(object):

  @cherrypy.expose
  def index(self):
    return open('Static/views/api.html')
```

Our goal was to create a very simple website composed by three different Web pages:

- index.html - website landing. The request is managed by **/ (root)** page handler;
- user.html - for showing collected data. The request is managed by **/user** page handler.
- api.html - for API documentation. The request is managed by **/api** page handler.

The reader might remember how we set up the static contents for the server (§6.5.2):

```
[/static]
tools.staticdir.on=True
tools.staticdir.dir="./Static"
```

The Static folder, located into the Server folder, contains other sub-folders:

- **views** - stores index, user and API HTML files defining the Web pages mark-up. We exploited the Bootstrap 5 technology to create a simple, but responsive website;
- **css** - stores a simple cascading style sheet, called style.css, describing the presentation of documents;

- **js** - stores a JavaScript (JS) file, called app.js, used as scripting language for Web pages;
- **images** - stores some images.

We report here an example of method taken from the JavaScript file to point out how was *ease* to consume the API:

```
async function getGateways() {
  let url = 'http://127.0.0.1:8080/api/gateways';
  try {
    let res = await fetch(url);
      return await res.json();
  } catch (error) {
    console.log(error);
  }
}
```

After defining the URL of interest, we perform a request thanks to the fetch function. The **Fetch API** accesses resources across the network. You can make HTTP requests using GET, POST and other methods, download, and upload files:

```
const response = await fetch(resource[, options]);
```

It accepts two arguments:

- resource - the URL string, or a Request object;
- options - the configuration object with properties like method, headers, body, credentials, and more.

The fetch function starts a request and returns a **promise**. When the request completes, the promise is resolved with the Response object. If the request fails due to some network problems, the promise is rejected. Because the await keyword is present, the asynchronous function is paused until the request completes. When the request completes, response is assigned with the Response object of the request. The response object, returned by the await fetch, is a generic *placeholder* for multiple data formats. We were interested in JSON format. res.json is a method of the Response object that allows a JSON object to be extracted from the response. The method returns a promise, so we waits for the JSON: await res.json. The result returns to the caller whose goal is to render it.

We did not target a complex implementation. We just needed a simple website to show data collected by the application. The reader might open files and see they are clean and clear.

In this chapter, we have introduced the hardware managing the LoRaWAN® protocol. We have also explained how we implemented a software component for dealing with the incoming messages from a datalogger and performing HTTP requests, as a client, to the Web service. Finally, we have seen how to reach the service thanks to a Browser. In the next chapter, we will discus about *computing strategies* and *optimizations* of the datalogger.

Chapter 8

Datalogger: computing strategies and optimizations

In this chapter, we will explain how we managed the *context* and tried different *computing strategies* to maximize and control the intermittent behavior of the datalogger. We will also explain how we performed some optimizations with the aim of reducing the power consumption as much as possible.

8.1 FRAM utilities

In section 5.2, we presented the MSP430 Peripheral Driver Library thanks to which we reduced the effort of describing complex tasks and targeted code portability. In particular, we wrote most of the code we discussed in chapter 5 by resorting to this library.

To deal with the FRAM available into the target MCU, we followed a similar approach by resorting to the **MSP MCU FRAM Utilities** by Texas Instruments, a collection of embedded software utilities that leverage the ultra-low power and virtually unlimited write endurance of FRAM. The collection includes [19]:

- ***Compute Through Power Loss (CTPL)***. *A utility API that enables ease of use with LPMx.5 low-power modes and a powerful shutdown mode that allows an application to save and restore critical system components when a power loss is detected;*
- ***LZ4 Compression (LZ4)***. *A lightweight compression utility based on the open source LZ4 compression standard and algorithm. The utility provides APIs for both compression and decompression on MSP FRAM microcontrollers and has been optimized for ultra-low power to enable datalogging, over the air updates and more;*
- ***Random Number Generator (RNG)***. *Implementation of a counter mode deterministic random byte generator (CTR-DRBG) according to the NIST SP 800-90A Rev 1 specification. Random numbers are generated using seed information stored in the TLV tables and are unique to each device;*
- ***Non-Volatile Storage (NVS)***. *Library that make handling of non-volatile data*

easy and robust against intermittent power loss or asynchronous device resets. Includes three different storage containers for a wide range of applications.

Although our project contains the entire collection into the fram-utilities folder, we only used the CTPL API. However, the other utilities might be exploited as future work as we will discuss in chapter 10.

8.2 Application start-up

In section 5.3.4, we introduced the entry point to the application whose goal is to *initialize* the batteryless environmental datalogger. As we saw, the code was implemented with the aim of being simple and straightforward. We report it here again:

```
1 int main(void)
2 {
3     /* Initialize the application */
4     Application_init(APPLICATION_MODE_NORMAL);
5
6     /* Should not reach this point */
7     return APPLICATION_FAILURE;
8 }
```

Let us go more in depth about the **initialization** of the datalogger. We coped with two main challenges as the system had to be:

- batteryless - *powered* only by a small photovoltaic panel;
- intermittent - *suspend* the computation in a safe way when a power failure occurred.

These features had important implications not only from the hardware point of view, as we saw in chapter 4, but also from the firmware point of view.

Above all, we programmed the system in the usual way, that is, thanks to a Micro-USB cable connected to the host PC we were using during the development. We could modify the firmware as many times as we needed in order to be built and loaded into the target MCU. The MCU module and Code Composer Studio IDE allowed us to perform such activities in a very simple way. In particular, the datalogger was powered by the circuitry mounted on the MCU module by exploiting the voltage coming from the Micro-USB cable. The point is, during the development, the datalogger worked as a *normal system* powered by a stable and always available power source.

The other important aspect concerned intermittency. We needed to understand how to suspend the computation in a safe way when a power failure occurred, that is, saving the **context**, as we already discussed in section 2.5. We will return to the context in the next section, but the reader might have guessed that, when the system was deployed as batteryless, it should have started, or better, it should have restored the context which was saved after removing the USB cable the last time. At that point the system would have been batteryless and intermittent. Anytime the system resumed, it restored the context and started working; as soon as a power failure occurred, it saved the context and suspended the computation in a safe way. This *life-cycle* would have been repeated *forever*.

Basically, there were activities to perform **only once** to initialize the datalogger. Let us refer to figure 8.1. The main function calls the Application_init function to initialize the hardware and perform the application **start-up**.

The very first step was to call the WorkloadManager_init function whose goal was to call the System_init function, we introduced in section 5.3.1, to initialize the hardware. In particular, the latter receives the system_frequency as parameter. The frequency can be selected by modifying the system_frequency variable we declared into the workload_manager.c file:

```
1 static uint32_t system_frequency = SYSTEM_FREQUENCY_1MHZ;
```

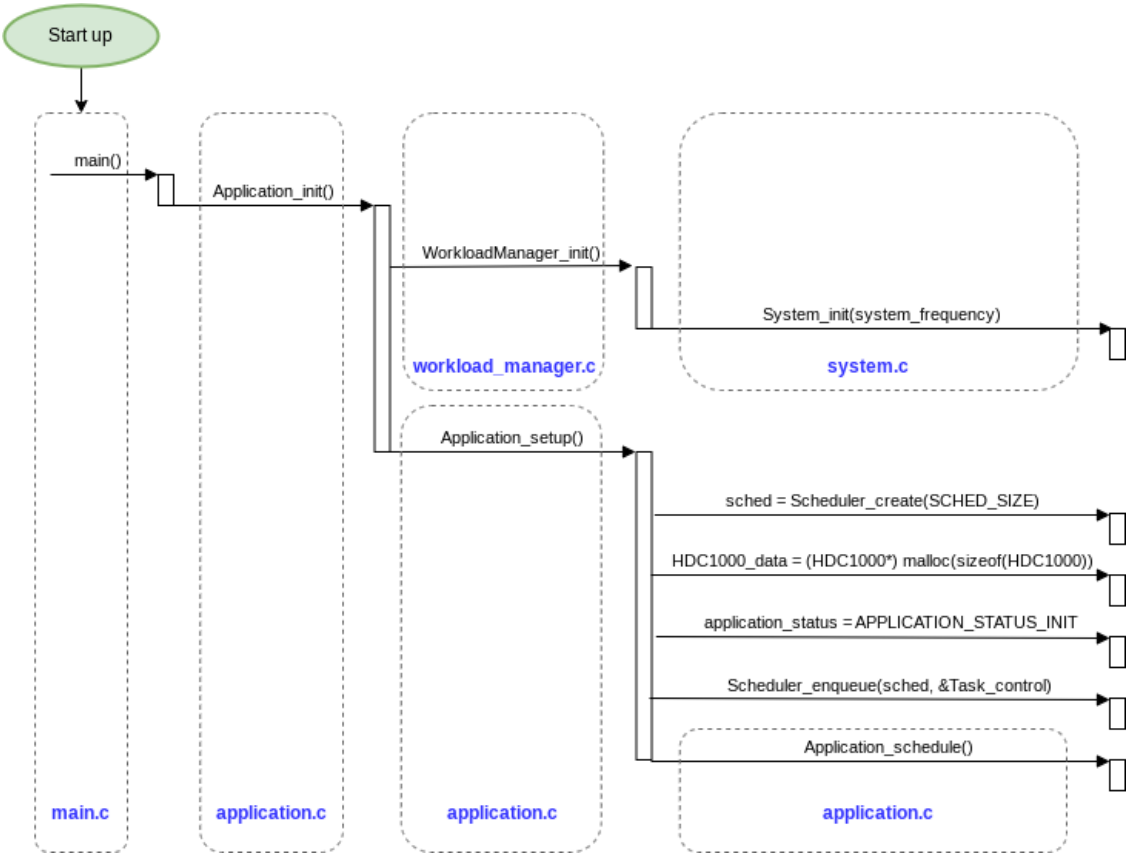


Figure 8.1: Datalogger start-up.

The **workload manager** represents an *abstraction* of our code. The idea was to apply different **policies** and **runtime settings** such as changing the system frequency. Indeed, we also implemented the WorkloadManager_setFrequency function to change the system frequency at run-time. This is an advanced topic we will resume in chapter 10. All the settings performed by the System_init function were retained anytime the system resumed thanks to the CTPL library we will introduce later on.

The next step was to call the Application_setup function which has a crucial role. After creating the **scheduler** and the structure capable of handling the temperature and humidity **sensor data**, the **status** of the application is set up. Then, the Task_control is

enqueued and the scheduler is run. From this point forward, the control is given to the scheduler. We will return to the scheduler and tasks in section 8.4.

Here is the key point. The data structures were created only once and retained between sessions. We obviously wanted to achieve this result as part of the context saving. Such aspect will be more clearer later on when we will introduce the tasks.

Before moving to the next section, we would like to make a final remark. As you can see in figure 8.1, each function belongs to a particular design unit depicted with dashed lines in the figure. The modularity was key in our project as we implemented the separation of concerns, the best practice we have already mentioned in the previous chapters.

8.3 The context

In section 2.5, we introduced theoretical aspects about the context concerning batteryless systems. In particular, we saw that the context involves physical and logical aspects. Indeed, developers must be sure that the hardware will resume correctly as well as the code will start running from a coherent and precise state. Our main goal was to manage the intermittent behavior of the batteryless environmental datalogger, which implied saving and restoring the physical and logical context.

8.3.1 Low-power modes

Although we did not take into account the MCU available **low-power modes** explicitly, we would like to briefly introduce this topic here, to better understand the contents of the next section.

The MSP430FR5994 MCU is designed for ultralow-power applications and uses different **operating modes** [21] which are shown in figure 8.2 The MCU provides one *active* mode and seven software selectable *low-power* modes. When the CPU is started up, the MCU will be in *active* mode. If required, the MCU can enter a particular *low-power* mode. The MCU can be waken up by an event; as an example, an *interrupt* might be one of such events [3].

The low-power modes LPM0 through LPM4 are configured with the CPUOFF, OSCOFF, SCG0, and SCG1 bits in the **status register (SR)** [21]. Instead of describing each low-power mode, we prefer to report settings, modes and CPU/Clock status in figure 8.3. In this way the reader can easily understand the bit settings an their consequences on clock sources and signals we introduced in section 5.3.1. Moreover,

> "the advantage of including the CPUOFF, OSCOFF, SCG0, and SCG1 mode-control bits in the SR is that the present operating mode is saved onto the stack during an interrupt service routine. Program flow returns to the previous operating mode if the saved SR value is not altered during the interrupt service routine. When setting any of the mode-control bits, the selected operating mode takes effect immediately. Peripherals operating with any disabled clock are disabled until the clock becomes active. Peripherals may also be disabled with their individual control register settings. All I/O port pins, RAM, and registers are unchanged. Wakeup from LPM0 through LPM4 is possible through all enabled interrupts"[21].

Here is the key point. When LPMx.5 (LPM3.5 or LPM4.5) is entered, the voltage regulator of the power management module (PMM) is disabled. **All RAM and register**

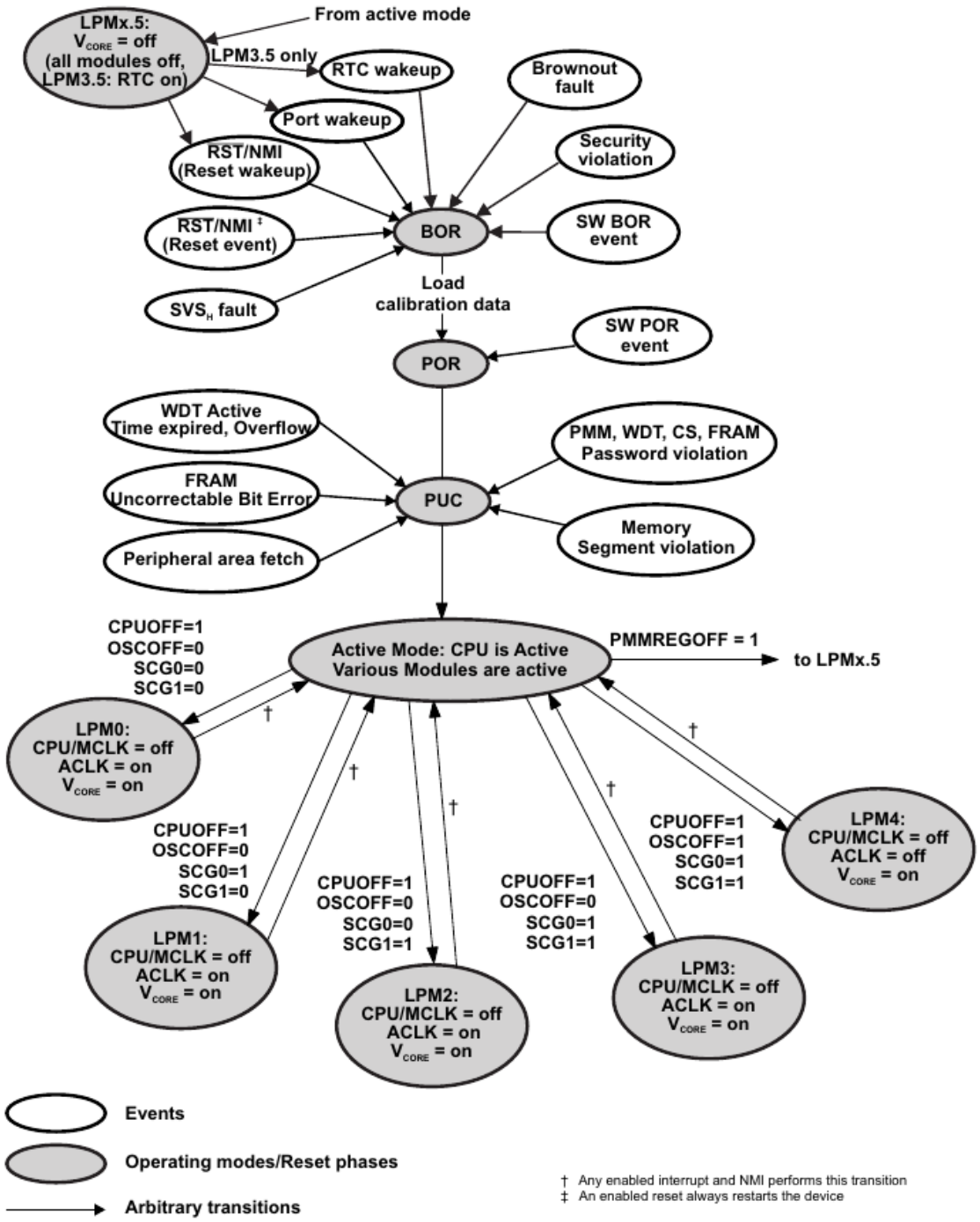


Figure 8.2: MSP430FR5994 MCU operating modes [21].

contents are lost. Although the I/O register contents are lost, the I/O pin states are locked upon LPMx.5 entry. Wakeup from LPM4.5 is possible through a power sequence, a

SCG1[1]	SCG0	OSCOFF[1]	CPUOFF[1]	Mode	CPU and Clocks Status[2]
0	0	0	0	Active	CPU, MCLK are active. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK, MCLK, or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0).
0	0	0	1	LPM0	CPU, MCLK are disabled. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0).
0	1	0	1	LPM1	CPU, MCLK are disabled. ACLK is active. SMCLK optionally active (SMCLKOFF = 0). DCO is enabled if sources ACLK or SMCLK (SMCLKOFF = 0). DCO bias is enabled if DCO is enabled or DCO sources MCLK or SMCLK (SMCLKOFF = 0).
1	0	0	1	LPM2	CPU, MCLK are disabled. ACLK is active. SMCLK is disabled.
1	1	0	1	LPM3	CPU, MCLK are disabled. ACLK is active. SMCLK is disabled.
1	1	1	1	LPM4	CPU and all clocks are disabled.
1	1	1	1	LPM3.5	When PMMREGOFF = 1, regulator is disabled. No memory retention. In this mode, RTC operation is possible when configured properly. See the *RTC* module for further details.
1	1	1	1	LPM4.5	When PMMREGOFF = 1, regulator is disabled. No memory retention. In this mode, all clock sources are disabled; that is, no RTC operation is possible.

Figure 8.3: MSP430FR5994 MCU operating modes settings [21].

reset (RST) event, or from specific I/O. Wakeup from LPM3.5 is possible through a power sequence, a RST event, a real-time clock (RTC) event, or from specific I/O [21].

To sum up, LPM0 can halt the CPU execution and the CPU can be in sleep mode. To reduce more power, there are other four lower power modes such as LPM3 and LPM4. To further reduce the power consumption, LPM3.5 and LPM4.5 are available; in these modes there is no memory retention [3].

8.3.2 Physical context

To manage the physical context, we resorted to the **compute through power loss (CTPL)** software utility by Texas Instruments we have mentioned in section 8.1.

The library helped us achieving exactly one of our main goals: saving and restoring the physical context, as according to software utility developers,

> "the CTPL enables an application to easily save the CPU and peripheral states into nonvolatile FRAM before powering down or entering a deep-sleep mode like LPMx.5. When a CPU wakes up, the CTPL restores an application exactly where it last executed with **context-save** and **restore**.
>
> The CTPL saves the state of a microcontroller by storing the stack from RAM, key peripherals, and CPU states to FRAM. Upon wakeup from low-power mode or a complete power-cycle, the CTPL checks for a signature in FRAM to determine whether an application should execute the C start-up routine (part of the C compiler runtime library that runs before entering the application main function) or return where the application left off. From a valid signature, the CTPL library restores the state of the CPU, key peripherals, and stack. The CTPL emulates the ease of use of a low-power mode LPM0-LPM3 function call.

> Using CTPL to skip the C start-up routine, the application restarts quicker from a cold start. The CTPL can help avoid C start-up routine reinitialization and save significant energy for applications and large arrays. In an MSP application, waking up from deep-sleep mode requires the software to restart. The application and its peripherals then need to reinitialize. The CTPL uses LPMx.5 to simplify the process of entering and exiting low-power mode and restoring the application"[18].

The software example

To use the CTPL utility, we started from a software demonstration by Texas Instruments taken from the *MSPWare*. **MSPWare** contains hundreds of software examples which are a great starting point when starting a new project or adding a new peripheral. Actually, we already highly resorted to MSPWare during the development to learn using the MSP Driver Library.

The example, called ctpl_ex4_adc12_b_monitor_msp-exp430fr5994 was perfect for us, as it allowed us to start creating the code for managing the intermittent behavior of the datalogger. By asynchronously removing Micro-USB power, the application automatically entered CTPL shutdown when Vcc reached a level of below 2.6 V. By reapplying Micro-USB power the application resumed with a counter value that did not start from zero. This goal was possible by exploiting the ADC12_B peripheral working as **power loss monitor**.

More in detail, the example demonstrated how to use the ADC12_B battery monitor and window comparator to actively monitor supply voltage and detect when power was lost. The ADC12_B peripheral was configured with a 2.0 V reference voltage and the internal battery monitor channel provided Vcc/2. The ADC12_B low side window comparator was configured to trigger the interrupt when Vcc reached ADC_MONITOR_THRESHOLD, 3.0 V by default. The high side window comparator was set to ADC_MONITOR_THRESHOLD + 0.1 V to ensure the device had reached a stable voltage before enabling the monitor. When the high side interrupt was triggered, it was disabled and the low side interrupt was enabled to begin actively monitoring Vcc. When a power loss was detected, the device will have invoked the **ctpl_enterShutdown** API which saved the application and peripheral state and waited for the device to enter BOR with a **64 ms** timeout. The device will have restored application and peripheral state when power was resumed and continued execution from the next line of code.

The main application blinked LED1 with incremental counts, resetting after four blinks. The power supply could be removed, by disconnecting the Micro-USB cable or unplugging the jumpers connecting the on-board emulator to the device, after a specific count of blink and then reapplied to verify that the context was saved.

Let us refer to figure 8.4 to better understand how the power loss monitor works. Basically, the power loss monitor is able to monitor the MCU power supply voltage: the output voltage of the regulator powering the system. At power-up, the CPU starts running as soon as the voltage reaches the **supply voltage supervisor shreshold (SVS TH)**. When the voltage reaches the ADC_MONITOR_THRESHOLD + 0.1 V, set to 3.1 V in the project, the ADC12_B high side window comparator triggers the interrupt to enable the monitor. More in detail, when the high side interrupt is triggered, it is disabled, and the low side interrupt is enabled to begin actively monitoring Vcc. The interrupt is represented by the green point in the figure. When a power loss is detected, the application invokes the **ctpl_enterShutdown** API which saves the application and peripheral state and waits for

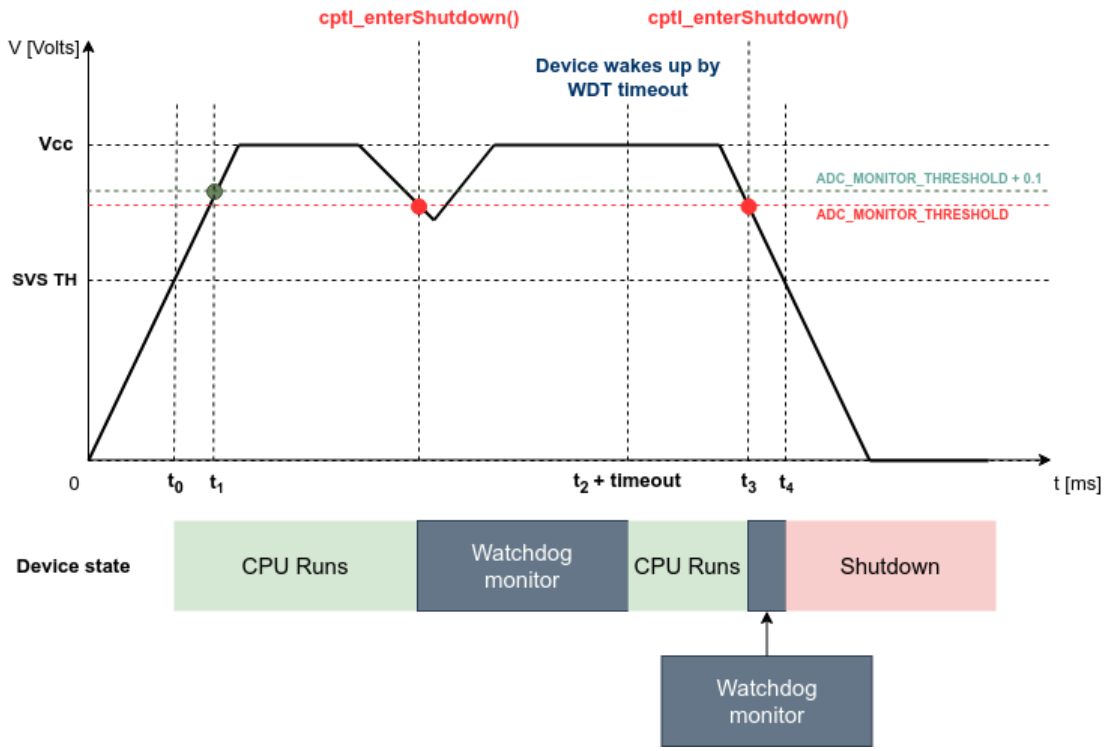


Figure 8.4: Datalogger: power loss monitor [18].

the device to enter BOR with a **64 ms** timeout. This event is represented by the red points in the figure. The example shows that if the voltage under monitoring returns higher than ADC_MONITOR_THRESHOLD, 3.0 V in the project, before the timeout, the API returns to the application without saving the context. On the contrary, if the voltage gets lower than SVS TH, before the timeout expiration, the context is saved and the system shutdown is performed. At a new start-up, the context is restored: the device restores application and peripheral state when power is resumed and continues the execution from the next line of code, the line following the ctpl_enterShutdown function call.

An important note. We had to modify the ctpl_low_level_macros.asm file located into the ctpl folder in order to compile the project. In particular, line **35** and line **54**, that originally were:

```
clr.b   &DMA0CTL_L      ; sw trigger, channel 0
clr.b   &DMA0CTL_L      ; sw trigger, channel 0
```

had to be changed as follows:

```
clr.w   &DMA0CTL        ; sw trigger, channel 0
clr.w   &DMA0CTL        ; sw trigger, channel 0
```

This modifications were crucial as they were part of the code performing copy and fill with either CPU or DMA to handle the context. Furthermore, we also had to perform a setting in Code Composer Studio to suppress the diagnostic warning **10367**. This could be easily done by reaching the *Diagnostic Options*: Build/MSP430 Compiler/Advanced Options/Diagnostic

Options. Finally, the value was added by selecting the Suppress diagnostic <id> section. The warning prevented a successfully compilation of the project.

Another important note. We had to update the hardware to save the context correctly. Indeed, the device supply voltage and ramp conditions were considered to avoid scenarios where voltage ramped down too slowly or too quickly. In our case, it rump down too quickly, thus we decided to add a **220 µF/16 V electrolytic capacitor** close to the LoRaWAN® module power supply which solved the problem. We will return to the topic in section 10.1.

CTPL API

The example used two out of four CTPL API functions.

The **ctpl_init** function initializes the utility and has to be called at the start of the _system_pre_init function for CCS or the **___low_level_init** function for IAR IDEs.

> "By default these functions are defined in ctpl_pre_init.c, but some applications might have their own version of the function. In this case the ctpl_pre_init.c file can be omitted and the function called at the start of the application's low level function"[19].

The **ctpl_enterShutdown** function saves the state of all the peripherals defined in the *include device file*, the context of the CPU and the active stack to non-volatile FRAM storage.

> "After saving the state it is marked as valid so that it can be restored after a reset or powering the device back on. All interrupt and wakeup sources are disabled and the device waits in active mode for the SVS to put the device into BOR. MCLK is configured to 4 MHz and the SMCLK and WDT_A dividers are set based on the timeout parameter. In this state the only source of a wakeup is a device reset, power up or a shutdown timeout. In all three wakeup scenarios the state is restored and the application resumed. The saved peripheral states, CPU state and stack are restored from the FRAM storage and the function returns back to the application from where it was called"[19].

The function takes the **timeout** as parameter ranging from 1 ms to 1024 ms. In the example, the value was set to 64 ms. If the device did not enter BOR after *timeout* milliseconds, the watchdog timer would have reset the device and caused a restore of the saved state.

Our implementation

First of all, we added the library as discussed in section 8.1.

Then, we added the ctpl_pre_init.c file to the project. It contains the functions that are called before the main function and initialization of variables (if required):

```
#include <msp430.h>
#include <ctpl.h>

#if defined(__TI_COMPILER_VERSION__)
int _system_pre_init(void)
#elif defined(__IAR_SYSTEMS_ICC__)
int __low_level_init(void)
#endif
```

```
{
    /* Initialize ctpl library */
    ctpl_init();
    /* Insert application pre-init code here. */
    return 1;
}
```

Listing 8.1: CTPL library: initialization.

In practice, as we have seen before, the ctpl_init function has to be called at the start to enable the CTPL library. If your application already declares this function, you can remove this file and add the call to ctpl_init at the start of your function. We preferred using the file ctpl_pre_init.c as a project unit instead of using the second possibility.

The ctpl_lnk_msp430fr5994.c file specifies the **system memory map**. The user should not modify its content. The file is simply added to the project and used.

The key role is played by the ctpl_msp430fr5994.c file. We report here only the section that can be modified by the user as the rest must be untouched:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>
#include <msp430.h>
#include <peripherals/ctpl_peripherals.h>

#define CTPL_SAVE_MPU
#define CTPL_SAVE_PORTA
#define CTPL_SAVE_PORTB
#define CTPL_SAVE_PORTC
#define CTPL_SAVE_PORTD
#define CTPL_SAVE_PORTJ
#define CTPL_SAVE_FRAM
#define CTPL_SAVE_CS
#define CTPL_SAVE_SYS
#define CTPL_SAVE_SFR
#define CTPL_SAVE_RTC_C
#define CTPL_SAVE_MPY32
//#define CTPL_SAVE_CRC16
//#define CTPL_SAVE_CRC32
//#define CTPL_SAVE_TA2
//#define CTPL_SAVE_TA3
//#define CTPL_SAVE_TA0
//#define CTPL_SAVE_TA1
//#define CTPL_SAVE_TA4
//#define CTPL_SAVE_TB0
//#define CTPL_SAVE_EUSCIA0
//#define CTPL_SAVE_EUSCIA1
//#define CTPL_SAVE_EUSCIA2
#define CTPL_SAVE_EUSCIA3
//#define CTPL_SAVE_EUSCIB0
//#define CTPL_SAVE_EUSCIB1
#define CTPL_SAVE_EUSCIB2
//#define CTPL_SAVE_EUSCIB3
//#define CTPL_SAVE_REF_A
//#define CTPL_SAVE_COMP_E
//#define CTPL_SAVE_ADC12_B
```

```
//#define CTPL_SAVE_DMA
#define CTPL_SAVE_WDT_A
```

Listing 8.2: The *include device file* used to save the state of peripherals.

Here is the magic. You have to comment out a line if you would not like to save the peripheral. As you can see, we were interested in saving all the peripherals whose macro is not commented. This file represents the **include device file** we mentioned before which is taken into account by the **ctpl_enterShutdown** function when saving the context.

A further step was to initialize the **ADC monitor**. As the reader might remember from section 5.3.1, the System_initADCMonitor is called by the System_init function (system.c file). Its implementation follows:

```
static void System_initADCMonitor(int frequency)
{
    /* Initialize timer for ADC trigger */
    switch (frequency) {
       case SYSTEM_FREQUENCY_16MHZ:
       TA0CCR0 = (frequency/ADC_MONITOR_FREQUENCY*16);
       break;

       case SYSTEM_FREQUENCY_8MHZ:
       TA0CCR0 = (frequency/ADC_MONITOR_FREQUENCY*8);
       break;

       case SYSTEM_FREQUENCY_4MHZ:
       TA0CCR0 = (frequency/ADC_MONITOR_FREQUENCY*4);
       break;

       case SYSTEM_FREQUENCY_1MHZ:
       TA0CCR0 = (frequency/ADC_MONITOR_FREQUENCY);
       break;

       default:
       TA0CCR0 = (frequency/ADC_MONITOR_FREQUENCY);
       break;
    }
    TA0CCR1 = TA0CCR0 / 2;
    TA0CCTL1 = OUTMOD_3;
    TA0CTL = TASSEL__SMCLK | MC__UP;

    /* Configure internal 2.0V reference */
    while (REFCTL0 & REFGENBUSY);

    REFCTL0 |= REFVSEL_1 | REFON;
    while (!(REFCTL0 & REFGENRDY));

    /* Initialize ADC12_B window comparator */
    ADC12CTL0 = ADC12SHT0_2 | ADC12ON;
    ADC12CTL1 = ADC12SHS_1 | ADC12SSEL_0 | ADC12CONSEQ_2 | ADC12SHP;
    ADC12CTL3 = ADC12BATMAP;
    ADC12MCTL0 = ADC12INCH_31 | ADC12VRSEL_1 | ADC12WINC;
    ADC12HI = (uint16_t) (4096 * ((ADC_MONITOR_THRESHOLD + 0.1) / 2) /
    (2.0));
    ADC12LO = (uint16_t) (4096 * (ADC_MONITOR_THRESHOLD / 2) / (2.0));
```

```
42      ADC12IFGR2 &= ~(ADC12HIIFG | ADC12LOIFG);
43      ADC12IER2 = ADC12HIIE;
44      ADC12CTL0 |= ADC12ENC;
45  }
```

Listing 8.3: ADC monitor initialization.

After initializing the timer for ADC trigger, the code configures an internal 2.0 V reference. Then, it initializes the ADC12_B window comparator. The monitor enables the high side to the monitor voltage plus 0.1 V to make sure the voltage is sufficiently above the threshold. When the high side is triggered, the interrupt service routine, which will be introduced later on, switches to the low side and waits for the voltage to drop below the threshold. When the voltage drops below the threshold, the device invokes the **ctpl_enterShutdown** function to save the application state and enter complete device shutdown. As you can see, part of the code is written at register level. We preferred not to touch this implementation as it was written by TI's developers.

The reader may wonder where to call exactly the library function to enter the device shutdown. This topic will be discussed shortly.

8.3.3 Logical context

We saved the logical context thanks to *tasks* managed by the *scheduler* implementing a *computing strategy*. Actually, we partially implemented the techniques we discussed in section 2.5. We are now ready to face and discuss all these topics.

8.4 Tasks

In section 8.2, we have recalled the entry point to the application whose goal is to initialize the batteryless environmental datalogger. In particular, we have seen that the Application_setup function is also responsible of creating and running the scheduler. We introduced the scheduler implementation in section 5.3.3. From now on, we will mostly take into account the contents of application.c file.

We created the **scheduler** thanks to the following statement:

```
1 /* Create the scheduler */
2 sched = Scheduler_create(SCHED_SIZE);
```

The function call creates the scheduler and initializes it with a given capacity. The function returns a pointer to a scheduler structure:

```
1 /* Scheduler structure */
2 typedef struct scheduler_ {
3     int tail;       // Current tail
4     int head;       // Current head
5     int size;       // Current number of items
6     int capacity;   // Capacity of scheduler
7     fPtr *taskPtr; // Pointer to array of tasks
8 } scheduler;
```

The capacity can be easily set by modifying the following macro into the scheduler.h file:

```
/* Scheduler size */
#define SCHED_SIZE  5
```

The crucial point concerns the fPtr field, a pointer to the array of *tasks*. Indeed, the scheduler has the main role of dequeuing function pointers. Obviously, we also had to declare variables into the application.c file to correctly handle the scheduler:

```
/* Task pointer with no parameters */
fPtr taskPtr;

/* Scheduler */
scheduler *sched;
```

To run the scheduler, we implemented the Application_schedule function by exploiting the example taken from [3]. Here is the code :

```
void Application_schedule()
{
    while (1) {
        while (!Scheduler_isEmpty(sched)) {
            taskPtr = Scheduler_dequeue(sched);
            (*taskPtr)();
        }
        __delay_cycles(100000);
    }
}
```

The Application_setup function runs the scheduler with the following simple function call:

```
/* Run the scheduler */
Application_schedule();
```

From this point forward the function starts dequeue tasks. In general, a task can be enqueued thanks to the Scheduler_enqueue function whose parameters are the scheduler pointer and the target task pointer:

```
Scheduler_enqueue(sched, &Task_control);
```

A **task** represents an abstraction of an *atomic operation* from the application point of view, as we saw in section 2.5.2. The idea was to create a task for each important operation which should not be interrupted. In other words, a task represents an application step for a task-based implementation. The main problem was to understand how many tasks we needed for the implementation. The solution was given, as usual, by the functional requirements of our project (3.3.4). Above all, each task was implemented as a function. As an example, the functional requirement FR1 was implemented thanks to the Task_control function whose main goal is to handle the datalogger:

```
void Task_control(void)
{
    ...
}
```

As we have seen before, the Application_setup function enqueues the Task_control task before running the scheduler. In this way, this task is the first one to be dequeued to take the control of datalogger.

In this section we simply wanted to introduce the scheduler and task implementation from an application point of view. In the next section, we will present different *computing strategies* we evaluated with the goal of ensuring a proper and correct execution even in the intermittent regime. Furthermore, we will discuss the implementation of a *feasible* computing strategy that allowed us to achieve the goal.

8.5 Computing strategies

A **computing strategy** is a *finite set of steps* able to solve a computing problem. This definition *might* coincide with the one of *algorithm*. An **algorithm** is a finite set of steps able to solve a problem. The algorithm will then be *represented* by a program written thanks to a selected programming language. To be run, the program must be loaded into the main memory as a *process* with the goal of being executed by a CPU. Developers usually resort to *pseudo-codes*, *flow-charts* and other techniques to describe algorithms. When you are at runtime, decisions are taken at computing level. You are running an algorithm, but computing decisions depend on the *current application status* and resources. You may need to add further variables or flags to control the overall application.

To control the **application status**, we defined a suitable set of macros, into the application.h file:

```
#define APPLICATION_STATUS_INIT         0
#define APPLICATION_STATUS_RUNNING      1
#define APPLICATION_STATUS_SHUTDOWN     2
#define APPLICATION_STATUS_RESTART      3
#define APPLICATION_STATUS_NOTXRX       4
#define APPLICATION_STATUS_READY        5
```

At runtime, we used a variable to control the *current* application status:

```
/* Application status */
int application_status;
```

this variable is initially set by the Application_setup function:

```
/* Set the application status */
application_status = APPLICATION_STATUS_INIT;
```

We also needed to declare suitable variables to control **data**:

```
int DATA_TRIGGER;
int DATA_READY;
int DATA_SEND;
int DATA_SENT;
int TIMER_EXPIRED;
```

These variables have the role of describing the current status of data concerning a measure. We will return to the topic later on.

8.5.1 Interrupts

Before moving to computing strategies, we would like to introduce the key role of *interrupts* and their **interrupt service routines (ISRs)**. Our implementation was totally *interrupt-based* to avoid *polling* and time wasting. In section 5.3.1, we already presented the ISR to handle UART responses. Now, we will briefly discuss about the others.

First of all, we were highly interested in being notified by the temperature and humidity sensor when the triggered measure was available. The DRDYn signal of the HDC1000 sensor module was connected to port P6.3 and worked as active-low signal. The ISR dealing with this interrupt has the following implementation:

```
/*
 * ======================================================
 * This is the P6.3 interrupt vector service routine.
 * HDC1000 sensor module - DRDYn
 * ======================================================
 */
#pragma vector=PORT6_VECTOR
__interrupt void Port6_ISR_handler(void)
{
    uint32_t status;

    /* Get P6.3 interrupt status */
    status = GPIO_getInterruptStatus(GPIO_PORT_P6, GPIO_PIN3);

    /* Clear P6.3 interrupt flag */
    GPIO_clearInterrupt(GPIO_PORT_P6, GPIO_PIN3);

    /* Check whether the pin is set */
    if (status) {
        /* The measure can be acquired */
        // do something
    }
}
```

Nothing complicated. After retrieving the port interrupt status, the interrupt is cleared. In case the pin is set, we can acquire the measure that is previously triggered.

Another important ISR regarded the harvesting module. Our goal was to be notified about the incoming shutdown of the module. The Status[1] pin of the energy harvesting module has this purpose and was connected to P4.2 port. Here is the implementation:

```
/*
 * ======================================================
 * This is the P4.2 interrupt vector service routine.
 * Energy harvester module - Status[1]
 * ======================================================
 */
#pragma vector=PORT4_VECTOR
__interrupt void Port4_ISR_handler(void)
{
    uint32_t status;

    /* Get P4.2 interrupt status */
    status = GPIO_getInterruptStatus(GPIO_PORT_P4, GPIO_PIN2);

    /* Clear P4.2 interrupt flag */
    GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN2);

    /* Check whether the pin is set */
    if (status) {
        /* Incoming shutdown */ // do something
```

```
    }
}
```

We just remember that the Status[1] signal is active-high.

Then, we moved to the timer. We were interested in being notified about the TA1 expiration. As we discussed in section 5.3.1, the timer is working in CONTINUOUS mode although it can be started and stopped when necessary. The goal was to start a new measure after the timer expiration. Here is its ISR:

```
/*
* ======================================================
* This is the TIMER1_A1 interrupt vector service routine.
* ======================================================
*/
#pragma vector=TIMER1_A1_VECTOR
__interrupt void Timer1_A1_ISR(void)
{
   /*
    * Any access, read or write, of the TAIV register automatically
    resets the
    * highest "pending" interrupt flag
    */
    switch ( __even_in_range(TA1IV,14) ){
        case  0: break; //No interrupt
        case  2: break; //CCR1 not used
        case  4: break; //CCR2 not used
        case  6: break; //CCR3 not used
        case  8: break; //CCR4 not used
        case 10: break; //CCR5 not used
        case 12: break; //CCR6 not used
        case 14:
            /* Enable the trigger for a new measure*/
            // do something
        break;
    default: break;
    }
}
```

Finally, we copied and paste the ADC12 ISR from the software example code which was key in our project as we discussed in section 8.3.2. Let us report the ISR to inspect it:

```
/*
 * ======================================================
 * This is the ADC12 interrupt vector service routine.
 * ======================================================
 */
#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
    switch(__even_in_range(ADC12IV, ADC12IV_ADC12LOIFG)) {
        // Vector  0: No interrupt
        case ADC12IV_NONE:        break;
        // Vector  2: ADC12MEMx Overflow
        case ADC12IV_ADC12OVIFG:  break;
        // Vector  4: Conversion time overflow
        case ADC12IV_ADC12TOVIFG: break;
```

```
16          // Vector  6: Window comparator high side
17          case ADC12IV_ADC12HIIFG:
18          /* Disable the high side and enable the low side interrupt. */
19          ADC12IER2 &= ~ADC12HIIE;
20          ADC12IER2 |= ADC12LOIE;
21          ADC12IFGR2 &= ~ADC12LOIFG;
22          break;
23          // Vector  8: Window comparator low side
24          case ADC12IV_ADC12LOIFG:
25          // do something
26          /* Disable the low side and enable the high side interrupt. */
27          ADC12IER2 &= ~ADC12LOIE;
28          ADC12IER2 |= ADC12HIIE;
29          ADC12IFGR2 &= ~ADC12HIIFG;
30          break;
31          default: break;
32      }
33  }
34
```

Two interrupts are taken into account. The case ADC12IV_ADC12HIIFG concerns the window comparator high side whose role is to start monitoring the Vcc voltage. After receiving this interrupt, the high side is disabled, whereas the low side interrupt is enabled to wait for a power loss. On the contrary, the ADC12IV_ADC12LOIFG flag concerns the window comparator low side. In this case a power failure occurred and the guide example exploited this interrupt to call the ctpl_enterShutdown function with a selected timeout. Since the latter returns exactly in that point, the low side is disabled and the high side interrupt is enabled to start monitoring again the Vcc voltage.

Just an important remark. ISRs are usually implemented to execute as fast as possible. The idea is to set some flags or variables to be used in the main program. ISRs react to interrupts and represent one of the best ways of implementing a reactive system. As we will see later on, we also exploited ISRs to enqueue tasks. This choice did not affect the ISRs execution time and allowed us to improve the overall performance.

The reader may have noticed that we used the comment *do something* in the crucial part of ISRs. What to put in place of comments depends on the selected computing strategy we are going to discuss now.

8.5.2 Control-based

We initially implemented a computing strategy we called **control-based**. The idea was to return the control to Task_control task after each atomic operation. The result is shown in figure 8.5.

Each task and ISR enqueues the Task_control task after being executed. The main advantage of this solution is to improve the *brain* of the application in order to take decisions quickly according to the current application status and data-flows. In practice, the Task_control task implements a **controller**. As an example, the controller would schedule the Task_manageWorkload task when needed to give the workload manager the possibility of performing some optimizations such as reducing or increasing the system frequency or modifying settings of some resources. Furthermore, the workload manager would apply a different *policy*. As an example, instead of performing just one measure, the

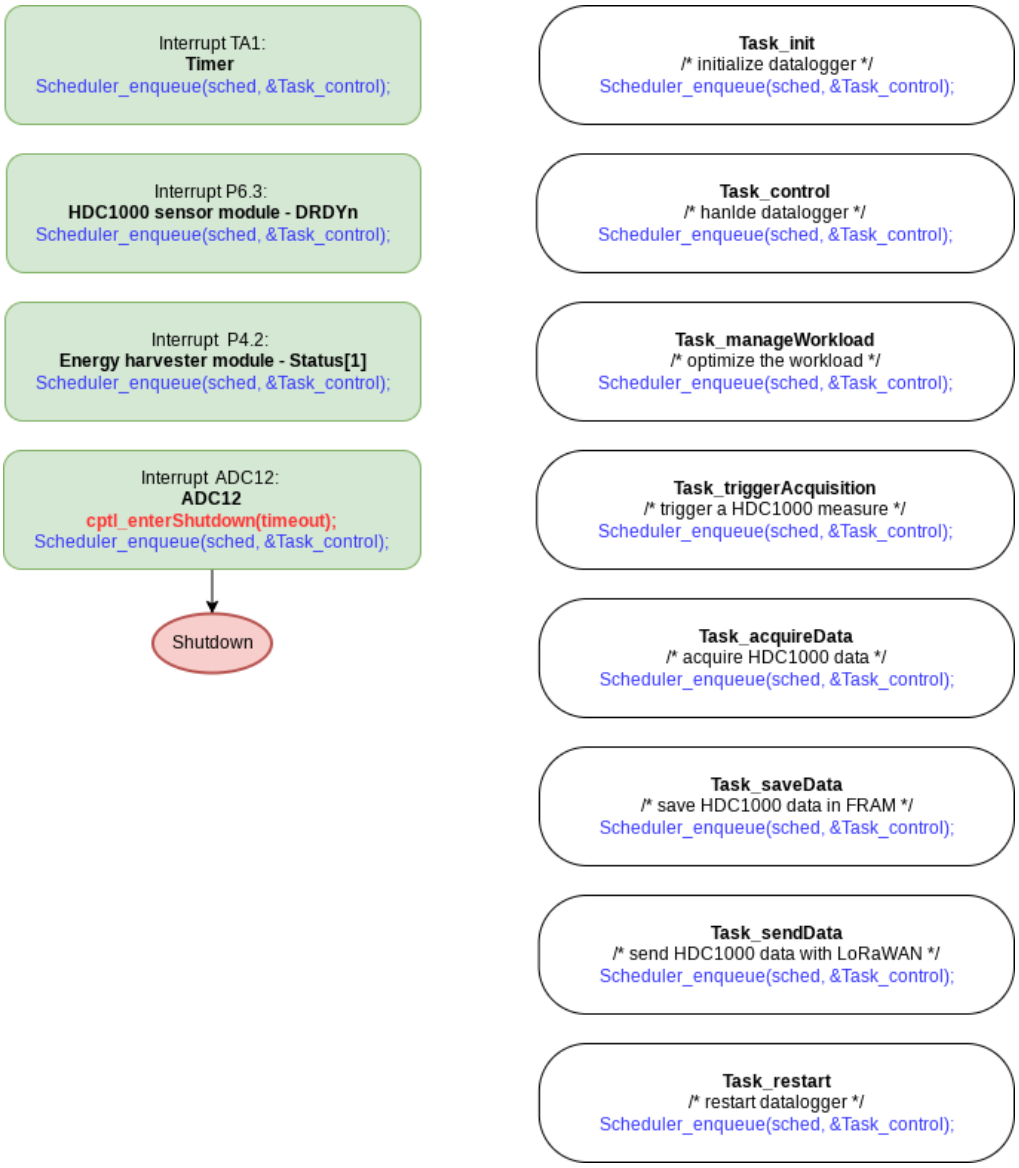


Figure 8.5: Datalogger: *control-based* computing strategy.

datalogger could be set to perform more than one measure in sequence. However, this is an advanced topic that will faced in chapter 10.

The main disadvantage concerns **overheads**. The controller is always scheduled, preventing other tasks to be executed *as soon as possible*. Moreover, the controller has to be well-implemented to control the applications according to the current status.

8.5.3 Control-based with deferring

To reduce overheads and the complexity of the controller implementation, we tried another computing strategy called **control-based with deferring** shown in figure 8.6.

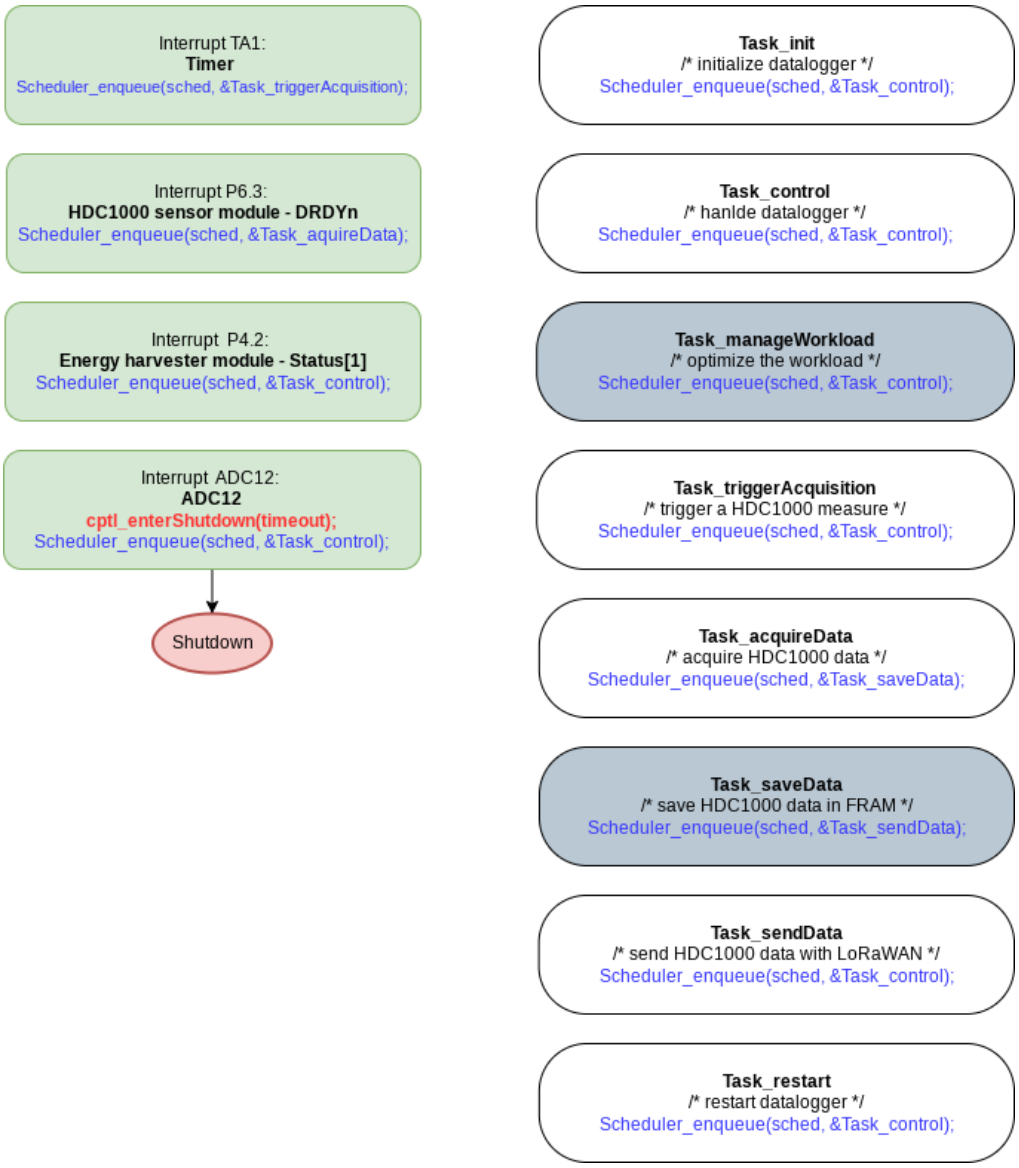


Figure 8.6: Datalogger: *control-based with deferring* computing strategy.

This solution partially reduces the controller execution as tasks and ISRs can decide the task to enqueue, not necessarily Task_control. Anytime TA1 expires, a new measure is triggered. Indeed, the TA1 ISR enqueues the Task_triggerAcquisition task. As soon as the measure is available, the P6.3 ISR is called and the Task_acquireData task is enqueued. After acquiring the measure, the Task_acquireData enqueues the Task_sendData whose role is to send data over the LoRaWAN® network. Finally, Task_sendData enqueues Task_control.

As you can see, the Task_saveData is shown with a grey color in figure 8.6. The reason is that we did not save the measure in FRAM after the acquisition. The same holds for the Task_manageWorkload task. This is another advanced topic we will resume in chapter 10.

8.5.4 As soon as possible

In this section we are going to present the solution we called **as soon as possible (ASAP)**. Basically, we tried to reduce overheads and promote the fast execution of tasks and ISRs under the supervision of the controller. The problem to face was about the duration of a **session** of the datalogger: the time elapsed between resuming and suspending. As we will see in chapter 9, one session lasted a few seconds. The point is, in a few seconds, the datalogger should resume, perform the mission and suspend in a safe way. We remember the reader that you need time to set up and perform a LoRaWAN® communication. Furthermore, transmitting is a power-hungry activity. Kind of features are challenging for a batteryless intermittent electronic system.

The ASAP strategy is shown in figure 8.7. The controller resorts to the application_status variable to orchestrate the application. Data flags are not taken into account, although set up by tasks for the sake of coherence. Here is its implementation:

```
static void Task_control(void)
{

    switch (application_status) {

        case APPLICATION_STATUS_INIT:
        Scheduler_enqueue(sched, &Task_init);
        break;

        case APPLICATION_STATUS_RUNNING:
        /* Future work */
        break;

        case APPLICATION_STATUS_SHUTDOWN:
        /* Future work */
        break;

        case APPLICATION_STATUS_RESTART:
        Scheduler_enqueue(sched, &Task_restart);
        break;

        case APPLICATION_STATUS_NOTXRX:
        /* The system cannot test or join a LoRaWAN network */
        break;

        case APPLICATION_STATUS_READY:
        Scheduler_enqueue(sched, &Task_triggerAcquisition);
        break;

        default:
        break;
    }
}
```

As we have seen in section 8.4, the control is taken by the scheduler after initializing the application. In particular, the controller is enqueued by the Application_setup function before running the scheduler:

```
/* Enqueue Task_control */
```

```
Scheduler_enqueue(sched, &Task_control);
```

In practice, the controller is run as the first task ever. Moreover, if you programmed the datalogger or pushed the reset button, the Application_setup function will have set the **application_status** variable to APPLICATION_STATUS_INIT.

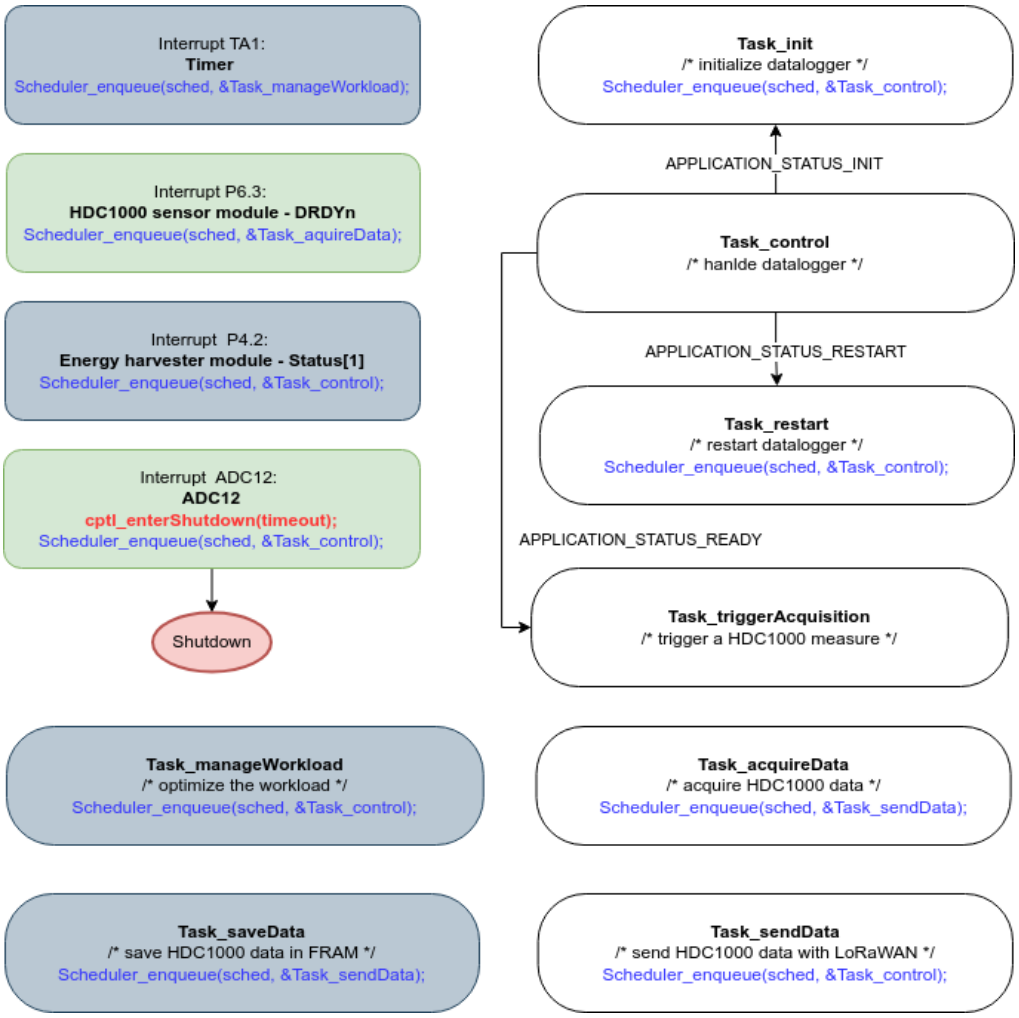


Figure 8.7: Datalogger: *as soon as possible (ASAP)* computing strategy.

When the application status is set to init, the Task_init is enqueued. Here is its implementation:

```
static void Task_init(void)
{
    /* Reset flags */
    DATA_TRIGGER  = FALSE;
    DATA_READY    = FALSE;
    DATA_SEND     = FALSE;
    DATA_SENT     = FALSE;
    TIMER_EXPIRED = FALSE;
```

```
9
10      /* Initialize the HDC1000 sensor */
11      HDC1000_init(CONFIG_MODE1);
12
13      /* Initialize the LoRaE5 transceiver */
14      if (LoRaE5_init(TEST_MODE)) {
15          application_status = APPLICATION_STATUS_READY;
16      } else {
17          application_status = APPLICATION_STATUS_NOTXRX;
18      }
19
20      /* Enqueue Task_control */
21      Scheduler_enqueue(sched, &Task_control);
22  }
```

After resetting the data flags to FALSE, the task correctly initializes the HDC1000 sensor to MODE1 and the LoRaWAN® transceiver to TEST mode. If the latter operation is successfully executed, the application_status variable is set to APPLICATION_STATUS_READY, otherwise is set to APPLICATION_STATUS_NOTXRX as the module transceiver is not available. Finally, the task gives the control to the controller.

Since the application status is ready, the controller enqueues the Task_triggerAcquisition task:

```
1  static void Task_triggerAcquisition(void)
2  {
3      /* Trigger data acquisition */
4      HDC1000_trigger();
5
6      /* Set flag */
7      DATA_TRIGGER = TRUE;
8  }
```

In this implementation, the DATA_TRIGGER variable is set to TRUE although unused. The task calls the HDC1000_trigger function we presented in section 5.3.2. Here a key point. We do not need to enqueue any task. The reason is that we are waiting for the P6.3 interrupt:

```
1  /*
2   * ======================================================
3   * This is the P6.3 interrupt vector service routine.
4   * HDC1000 sensor module - DRDYn
5   * ======================================================
6   */
7  #pragma vector=PORT6_VECTOR
8  __interrupt void Port6_ISR_handler(void)
9  {
10     uint32_t status;
11
12     /* Get P6.3 interrupt status */
13     status = GPIO_getInterruptStatus(GPIO_PORT_P6, GPIO_PIN3);
14
15     /* Clear P6.3 interrupt flag */
16     GPIO_clearInterrupt(GPIO_PORT_P6, GPIO_PIN3);
17
18     /* Check whether the pin is set */
```

```
19      if (status) {
20
21          /* The measure can be acquired */
22          DATA_READY = TRUE;
23
24          /* Enqueue Task_acquireData */
25          Scheduler_enqueue(sched, &Task_acquireData);
26      }
27  }
```

In addition to setting the DATA_READY variable useful for other computing strategies, the ISR enqueues the Task_acquireData task.

The Task_acquireData task has the role of acquiring the measure that was previously triggered. Here is its implementation:

```
1  static void Task_acquireData(void)
2  {
3      /* Reset flag */
4      DATA_READY = FALSE;
5
6      /* Acquire data */
7      HDC1000_acquire();
8
9      /* Get data */
10     HDC1000_getData(HDC1000_data);
11
12     /* Set flag */
13     DATA_SEND = TRUE;
14
15     /* Enqueue Task_sendData */
16     Scheduler_enqueue(sched, &Task_sendData);
17 }
```

The main point is to call the HDC1000_acquire followed by the HDC1000_getData function we presented in section 5.3.2. Finally, the task enqueues the Task_sendData task.

The Task_sendData task sends the measure over the LoRaWAN® network:

```
1  static void Task_sendData(void)
2  {
3      /* Reset flag */
4      DATA_SEND = FALSE;
5
6      sprintf(LoRaE5_data, "%013llu%5hu%5hu", HDC1000_data->serialID,
       HDC1000_data->temperature, HDC1000_data->humidity);
7
8      /* Send data */
9      if (LoRaE5_sendData(LoRaE5_data)) {
10         DATA_SENT = TRUE;
11     } else {
12         DATA_SENT = FALSE;
13     }
14
15     /* Set the application status */
16     application_status = APPLICATION_STATUS_RUNNING;
17
18     /* Enqueue Task_control */
```

```
    Scheduler_enqueue(sched, &Task_control);
}
```

After creating a suitable data format, data are sent over the LoRaWAN® network by exploiting the LoRaE5_sendData function we presented in section 5.3.2. Finally, the control is given to the controller.

That's all. Since the application status is running, the controller does not enqueue any other task and waits for the upcoming shutdown. From the application point of view, the **logical context** is perfectly saved and under control as the measure has been sent and nothing else has to be performed. The scheduler queue is empty and nothing happens.

Here is the key point of the project. The most important implementation step concerned the **physical context** saving and restoring to correctly manage the intermittent behavior of the datalogger. Let us report the ADC12 ISR:

```
/*
 * ======================================================
 * This is the ADC12 interrupt vector service routine.
 * ======================================================
 */
#pragma vector = ADC12_VECTOR
__interrupt void ADC12_ISR(void)
{
    switch(__even_in_range(ADC12IV, ADC12IV_ADC12LOIFG)) {
        case ADC12IV_NONE:        break;
        case ADC12IV_ADC12OVIFG:  break;
        case ADC12IV_ADC12TOVIFG: break;
        case ADC12IV_ADC12HIIFG: // Window comparator high side

        /* Disable the high side and enable the low side interrupt */
        ADC12IER2 &= ~ADC12HIIE;
        ADC12IER2 |= ADC12LOIE;
        ADC12IFGR2 &= ~ADC12LOIFG;
        break;
        case ADC12IV_ADC12LOIFG: // Window comparator low side

        /* Enter device shutdown with 64ms timeout */
        ctpl_enterShutdown(CTPL_SHUTDOWN_TIMEOUT_64_MS);

        /* Set the application status */
        application_status = APPLICATION_STATUS_RESTART;

        /* Enqueue the Task_control task to resume the logical context */
        Scheduler_enqueue(sched, &Task_control);

        /* Disable the low side and enable the high side interrupt */
        ADC12IER2 &= ~ADC12LOIE;
        ADC12IER2 |= ADC12HIIE;
        ADC12IFGR2 &= ~ADC12HIIFG;

        break;
        default: break;
    }
}
```

We already discussed this ISR, but let us focus on the window comparator low side interrupt. The ctpl_enterShutdown API is called to save the physical context. We were sure that the system shutdown would have executed as Vcc would have decreased to SVS TH voltage before the CTPL_SHUTDOWN_TIMEOUT_64_MS expiration. In this scenario, the physical and logical context are perfectly saved. After resuming, or better, after restoring the physical context, the code continues the execution from the next line of code following the ctpl_enterShutdown API. Here is where we take the control of the code execution by setting up the application_status variable to APPLICATION_STATUS_RESTART and enqueueing the Task_control task. This solution works correctly as the scheduler queue is empty and the Task_control task is the first one to be dequeued.

Since the application status is restart, the controller enqueues the Task_restart task:

```
static void Task_restart(void)
{
    /* Reset flags */
    DATA_TRIGGER  = FALSE;
    DATA_READY    = FALSE;
    DATA_SEND     = FALSE;
    DATA_SENT     = FALSE;
    TIMER_EXPIRED = FALSE;

    /* Delay */
    __delay_cycles(1500000);

    /* Clear P4.2 interrupt (issue) */
    GPIO_clearInterrupt(GPIO_PORT_P4, GPIO_PIN2);

    /* Enable P4.2 interrupt (issue) */
    GPIO_enableInterrupt(GPIO_PORT_P4, GPIO_PIN2);

    /* USCI_A3 RX interrupt Clear interrupt (issue) */
    EUSCI_A_UART_clearInterrupt(UART_BASE, EUSCI_A_UART_RECEIVE_INTERRUPT
    );

    /* Enable USCI_A3 RX interrupt (issue) */
    EUSCI_A_UART_enableInterrupt(UART_BASE,
    EUSCI_A_UART_RECEIVE_INTERRUPT);

    /* Restart the LoRaE5 transceiver */
    if (LoRaE5_restart(TEST_MODE)) {

        /* Set the application status */
        application_status = APPLICATION_STATUS_READY;

        /* Clear P6.3 interrupt (issue) */
        GPIO_clearInterrupt(GPIO_PORT_P6, GPIO_PIN3);

        /* Enable P6.3 interrupt (issue) */
        GPIO_enableInterrupt(GPIO_PORT_P6, GPIO_PIN3);

     } else {

        /* Set the application status */
```

```
40          application_status = APPLICATION_STATUS_NOTXRX;
41    }
42
43      /* Enqueue Task_control */
44      Scheduler_enqueue(sched, &Task_control);
45  }
```

After resetting the flags, the task introduces a crucial **delay**. Indeed, according to the datasheet, after power-up, the HDC1000 sensor needs at most 15 ms to be ready to start relative humidity and temperature measurement [17]. Actually, the delay mainly targets the LoRaWAN® module which needs more time to be ready, although we could not retrieve the value from the datasheet. We estimated that at least one second was necessary. Later, the task reinitializes the LoRaWAN® transceiver in TEST mode to start sending measures; obviously, this activity will modify the application_status variable. Finally, the task gives the control to the controller.

We point out that the Task_restart task restores the logical context from a very precise point. Indeed, after every shutdown, the data structure concerning the scheduler and sensor are perfectly saved and everything restarts from a suitable point in the code. The last point in the code is represented by an empty scheduler and a place inside the ADC12 ISR from which we can restart. The new point in the code, after restoring, is represented by the execution of the Task_restart task. Everything is coherent and under control. Without this key code and ADC12 ISR, the system would have been reset after each shutdown as we would expect in a *normal* application.

As a final comment, thanks to the ASAP computing strategy, we achieved the **expected outcomes (M1)** as we discussed in section 3.7.2. That is:

- the datalogger was batteryless and worked intermittently and autonomously. Data were sent to the gateway by using the TEST mode;
- the gateway's LoRaWAN® module was connected to a target PC via USB interface. A Python program was able to control the LoRaWAN® module and work as a REST Client performing HTTP requests to the Web Service;
- a Python REST Web service was reachable on the same PC;
- each system was easily configurable.

We had a **working solution** capable of closing the chain datalogger-gateway-server.

8.6 Optimizations

After achieving the expected results, we focused on the datalogger to maximize its intermittent working sessions. Since this goal highly depended on its power consumption, we tried to reduce it as much as possible. The power consumption could be reduced at the hardware level and application level. In particular, we followed the *Principles for Low-Power Applications* suggested by the target MCU user's guide at section 1.5 [21].

8.6.1 Hardware level

To implement the principles at hardware level, as we already described in this chapter and chapter 5, we:

- used interrupts to wake the processor and control program flow;
- switched on peripherals only when needed;
- used low-power integrated peripheral modules in place of software driven functions;
- did not use low-power modes directly, but we resorted to the CTPL library which offers ease-of-use with low-power modes LPMx.5.

Furthermore, we set all GPIO outputs to low values as this is a good practice able to minimize power consumption. Finally, we set the system frequency to 1 MHz: a sixteenth of the available maximum frequency.

8.6.2 Application level

To implement the principles at application level, we:

- avoided frequent subroutine and function calls due to overhead. Although we tried to implement as much as possible the *separation of concerns*, the number of function calls was not so high and atomic operations were executed in a reasonable amount of time;
- avoided flag polling and long software calculations when possible.

We used LEDs only during the development to understand the current application status, but not at deployment. Although their power consumption might be negligible when targeting *normal* applications, when you are targeting a batteryless low-power application they represent a waste of current. Current becomes a precious resource as we will see in the next chapter. Actually, we simply blinked the red LED twice, after resuming, to be sure that the Task_restart task was executed.

We used software delays only when necessary. As an example, as we have seen previously, the LoRaWAN® module needs at least one second to be ready at each restart. The delay was calculated according to the system frequency and we were not allowed to by-pass this time.

8.6.3 Optimized application

Thanks to the previous optimizations, we got an **optimized application (M2)**, that is (§3.7.2):

- the power consumption of the datalogger was optimized to maximize its intermittent working sessions;
- a user could display the available data by running a Browser.

We hope the reader had a positive experience in facing all the implementation details we discussed throughout the chapters. In the next one, we will present the main results we achieved thanks to this experimental thesis.

Chapter 9

Experimental Results

In this chapter, we will report the main results we achieved thanks to this work. In particular, we will compare the contents of the requirements document with the outcome of an experiment.

9.1 Requirements vs implementation

We closed the previous chapter stating that we developed an optimized application. In particular:

- the power consumption of the datalogger was optimized to maximize its intermittent working sessions;
- a user can display the available data by running a Browser.

Such statements summarize the expected outcomes achieved through the development process that will be demonstrated and discussed in the next sections. From now on, we are going to compare the implementation with the requirements document with the goal of pointing out the differences.

Figure 9.1 shows the physical appearance of the datalogger whose hardware implementation was highly discussed in chapter 4. The datalogger is a batteryless system (NFR1) that harvests energy thanks to an indoor (NFR3) photovoltaic panel (NFR2). Actually, the Solar Development Kit by PowerFilm® (§4.5) also provides an outdoor solar panel which could be eventually used for further experiments. The 220 mF super-capacitor can charge up to 4.2 V. The sensing module by MIKROE (§4.3) is designed around the HDC1000 IC which is functional within the full −40 °C to 125 °C temperature range (NFR12). The percentage of relative humidity is measured (NFR13). By looking at the figure, the reader may have noticed that there is no package wrapping the datalogger. The absence of a package had an high impact on the measures performed by the system as we will see later on. Furthermore, the matrix board hosting all the sub-systems, gave us the possibility of easily handling wires and connectors as well as mounting and unmounting the jumpers, super-capacitor and solar panel. Finally, the matrix board could be easily placed anywhere to conduct experiments.

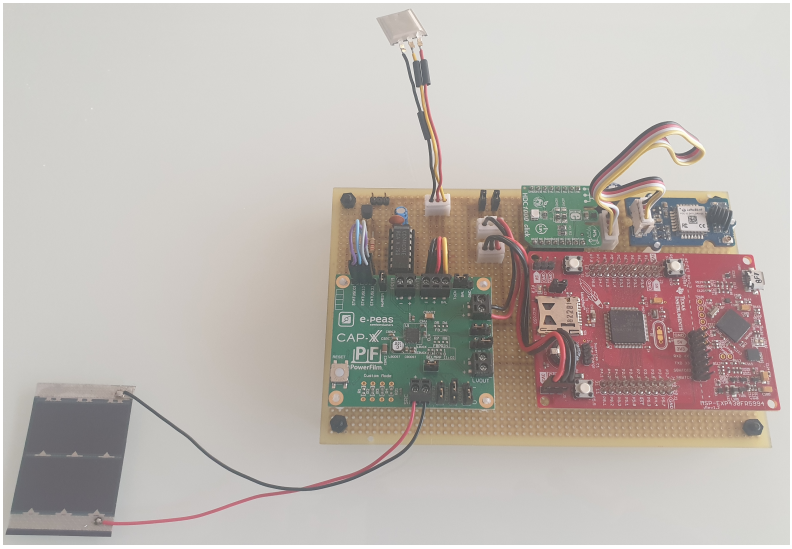


Figure 9.1: Datalogger: final assembly.

Figure 9.2 shows the physical appearance of the gateway whose implementation was highly discussed in chapter 7. We simply mounted the Wio-E5 mini module by seeed studio

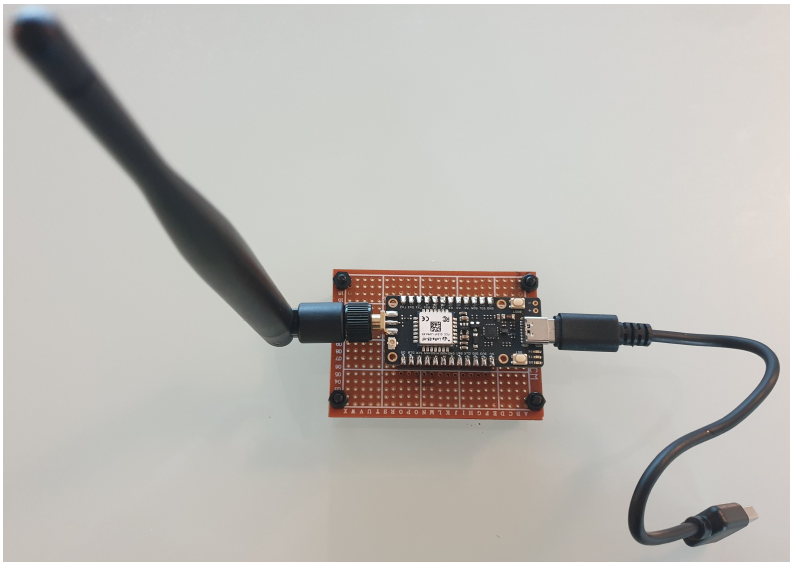


Figure 9.2: Gateway: final assembly.

on a suitable matrix board to improve the system's mechanical stability. The module should be connected to a host PC via USB cable.

Logical and physical interactions among systems, involved in the application, were described thanks to *interfaces* technique (§3.3.3). Table 9.1 has been updated as the LoRaWAN® technology implemented the wireless communication. The table is short, but complex applications might have hundreds of rows.

Actor	Logic Interface	Physical Interface
Datalogger	Data sender	LoRaWAN® module
Gateway	Data concentrator, REST Client	LoRaWAN® module, Internet link
Server	REST API	Internet link
Browser	Graphical User Interface (GUI)	PC, Smartphone

Table 9.1: Interfaces of the implemented application.

We report here again the story we told in the requirements document. We have colored in red the features of the target application that we did not implement. The reason will be discussed in section 10.1.

A datalogger works intermittently due to its nature. It resumes when energy harvested thanks to the photovoltaic panel allows the system to be powered. After resuming, it performs an initialization in order to restore the context that was saved before the last shutdown. Then, it captures a temperature and humidity measure. This measure is sent to a known gateway thanks to a long-range wireless interface in the form:

datalogger_id temperature humidity

The gateway receives the measure and returns a timestamp to be added to the measure. The measure is sent back to datalogger in the form:

datalogger_id timestamp

The datalogger stores the measure in its internal memory. It collects measures regularly and consistently. The datalogger must be notified about a power failure; in this case it runs a shutdown procedure in order to save its context correctly. Since a datalogger works indoor, a suitable photovoltaic panel is selected among those available in the market. Low-power consumption is a key metric. The range of temperature and humidity measures is compatible with indoor ones. A datalogger is easily manageable in order to allow developers to conduct experiments.

A gateway has two main purposes. First of all, it works as a data concentrator thanks to its long-range wireless interface. It can receive measures from any datalogger with a relationship many to one. The second purpose is to send each received measure to a REST Web service. This means it acts as a REST client capable of performing POST HTTP requests to the server. Each gateway can be recognized thanks to either IPv4 and IPv6 address. IPv4 address is mandatory.

A server hosts a REST Web service exposing suitable endpoints. It is able to deal with more than one gateway with a relationship many to one. The Web service saves each measure into a database. The database contains tables storing data about gateways, dataloggers and measures. In particular, the server knows each gateway, dataloggers managed by each gateway and measures performed by each datalogger managed by a gateway. Data can be retrieved thanks to the REST spirit and thanks to a Browser. The Browser allows users to display and analyze data with the goal of maximizing the user experience.

Finally, table 9.2 reports the implementation of functional requirements. Each row has been updated with the file unit containing the peace of code implementing the functional requirement. Also in this case the table is short, but complex applications might require several tables with hundreds of rows.

ID	Description	File unit	Implementation
FR1	Handle Datalogger	Node/application/application.c	Task_control
FR1.1	Restore the context	Node/ctpl/ctpl.c	ctpl_enterShutdown
FR1.2	Capture a measure	Node/application/application.c	Task_triggerAcquisition
FR1.3	Store a measure in memory	Node/application/application.c	Future work
FR1.4	Collect measures	Node/application/application.c	Task_acquireData
FR1.5	Send a measure to a gateway	Node/application/application.c	Task_sendData
FR1.6	Save the context	Node/ctpl/ctpl.c	ctpl_enterShutdown
FR1.7	Manage workload	Node/application/application.c	Future work
FR1.8	Monitor power failures	Node/application/application.c	ADC12_ISR
FR1.9	Manage shutdown	Node/application/application.c	Future work
FR2	Handle Gateway	Gateway/Gateway.py	Gateway
FR2.1	Receive a measure from a datalogger	Gateway/SerialCommunicationHandler.py	SerialCommunicationHandler
FR2.2	Send a measure to the Web service	Gateway/RESTClient.py	RESTClient
FR3	Handle Web service	Server/Server.py	Application
FR3.1	Add a gateway	Server/Controller/Controllers.py	GatewayController (POST)
FR3.2	Retrieve a gateway	Server/Controller/Controllers.py	GatewayController (GET)
FR3.3	Retrieve all gateways	Server/Controller/Controllers.py	GatewayController (GET)
FR3.4	Add a datalogger to a gateway	Server/Controller/Controllers.py	DataloggerController (POST)
FR3.5	Retrieve a datalogger of a gateway	Server/Controller/Controllers.py	DataloggerController (GET)
FR3.6	Retrieve all dataloggers of a gateway	Server/Controller/Controllers.py	DataloggerController (GET)
FR3.7	Add a measure of a datalogger	Server/Controller/Controllers.py	MeasureController (POST)
FR4	Handle Browser	-	User selected Browser
FR4.1	Display measures	Server/Static/js/app.js	renderGateways

Table 9.2: Implementation of functional requirements.

We emphasize the effectiveness of the simplified development process we resorted to. The tables we have reported here contain precious information and belong to the project's **documentation**. Indeed, the requirements is one of many file units that would document a complex project. Resorting to a development process, although simple, represents a methodology and strategy which gives developers benefits and advantages. In the next section, we are going to compare the requirements with the outcome of an experiment.

9.2 The experiment

According to the dictionary, an **experiment** is a test done in order to learn something or to discover if something works or is true [8]. For our purposes, we were interested in proving that our application would have worked, that is, it would have met our specifications. In the following, we are going to use a practical approach helping us showing the reader how to deploy the application and start collecting data. Thus, let us describe the experiment we conducted step by step.

Above of all, we ran the **server** on the host PC we used to develop the application. After running a VS Code IDE instance, we targeted the Server folder. Then, we deleted the Database.sqlite file located into the Repository sub-folder to create a new one at server start-up. The latter operation is normally not necessary; for our purposes, we wanted to store only the data collected by the experiment that we were conducting. Furthermore, when you delete the database, the contents of Measures.txt file are deleted as a consequence. The server can be easily run thanks to the following command:

```
python3 Server.py
```

The result of the previous command is shown in figure 9.3. After creating the database file and tables, the server will start serving at request reaching host and port you defined into the Server/Config/Server.conf configuration file. The server will be up until you push ctrl+C buttons on the keyboard.

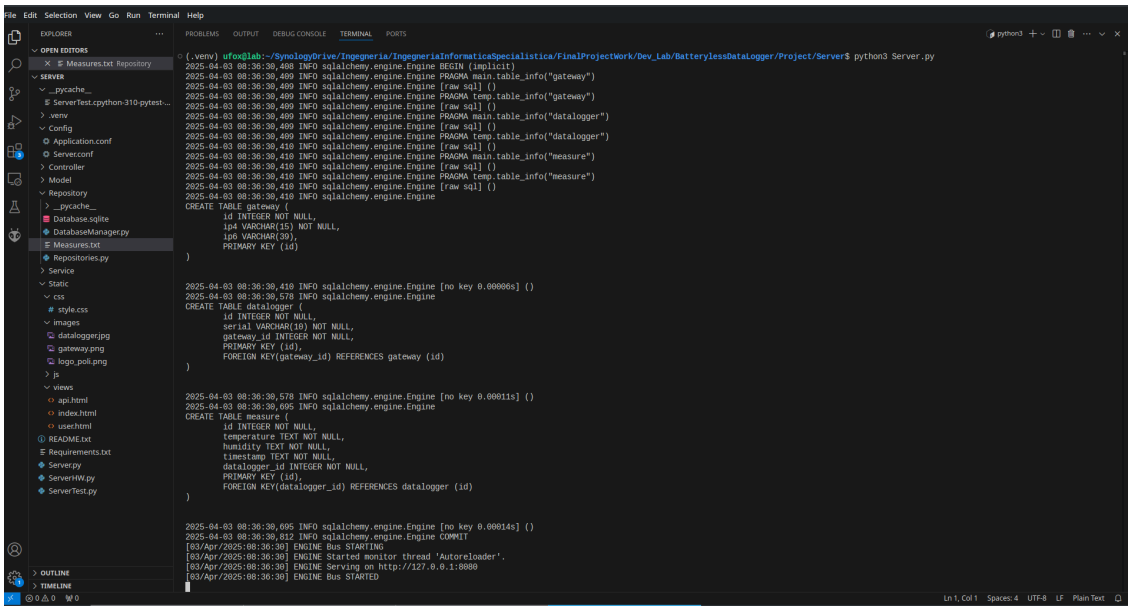


Figure 9.3: Server started.

Then, we ran the **gateway** on the same host PC we were running the server. We remember the reader that server and gateway can be run on any host computer, not necessarily the same. In particular, the gateway can be configured thanks to the Gateway/Gateway.conf configuration file. We decided to use the same PC for the sake of convenience. Thus, we connected the Wio-E5 mini LoRaWAN® module to an available USB port. Later, we ran another VS Code IDE instance to target the Gateway folder. The gateway can be easily run thanks to the following command:

```
python3 Gateway.py
```

If the command fails, it means that the selected USB port is not correctly configured or the server is not reachable. Otherwise, the result of the previous command is shown in figure 9.4. After testing the selected USB port and the server reachability, the gateway enters

Figure 9.4: Gateway started.

the TEST mode and waits for incoming massages from a datalogger. Indeed, it works as a data concentrator and can concentrates data coming from many dataloggers (§10.1.2). By looking at the figure, the reader may have noticed that the gateway receives the endpoint of the resource created by the Web service. The endpoint will be used by the gateway to interact with the Web service aiming at requesting a service. The creation of the resource can be further analyzed by looking at figure 9.5. Also the gateway can be stopped by pushing ctrl+C buttons on the keyboard.

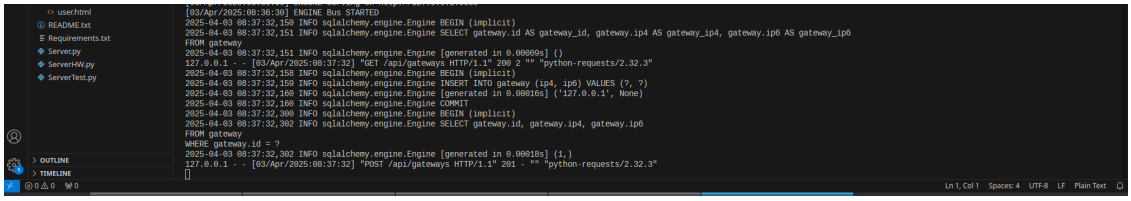


Figure 9.5: The Web service created the resource and returned its endpoint.

Finally, we deployed the **datalogger**. After programming it with the final version of the firmware and waiting for the transmission of the measure was completed, we removed the Micro-USB cable. The latter operation saved the context. We immediately measured the initial voltage of the super-capacitor and annotated the time instant we performed the measure. This data were useful to understand how much time was passed between the deployment and the first measure received by the gateway. Then, we performed the following set of steps in sequence to prepare the datalogger for deployment (§4.8.2):

- we disconnected all JC jumpers placed in the computing module;
- we reset the energy harvesting module by pushing its reset button;
- we connected the super-capacitor;
- we connected JH jumpers and, successively, the photovoltaic cell;

We chose a suitable place for the deployment. It was a **spring sunny day**.

Figure 9.6 reports the first measure received by the gateway and sent to the Web service (NFR11). The server created new resources: the datalogger which performed the measure and the measure itself. In this case, only the endpoint related to the datalogger returned to the gateway.

Figure 9.6: The first measure received by the gateway and sent to the Web service.

Table 9.3 reports the useful data that we have previously mentioned and that we will be taken into account for data analysis.

Datum	Value
Super-capacitor initial voltage	3.3 V
Deployment instant	2025-04-03 08:37:00 AM
First measure instant	2025-04-03 09:17:38 AM

Table 9.3: Useful values for data analysis.

We could display all the measures captured by the datalogger thanks to a Browser as you can see in figure 9.7. The Web page reflects the modularity and scalability of the application which can serve multiple gateways and dataloggers.

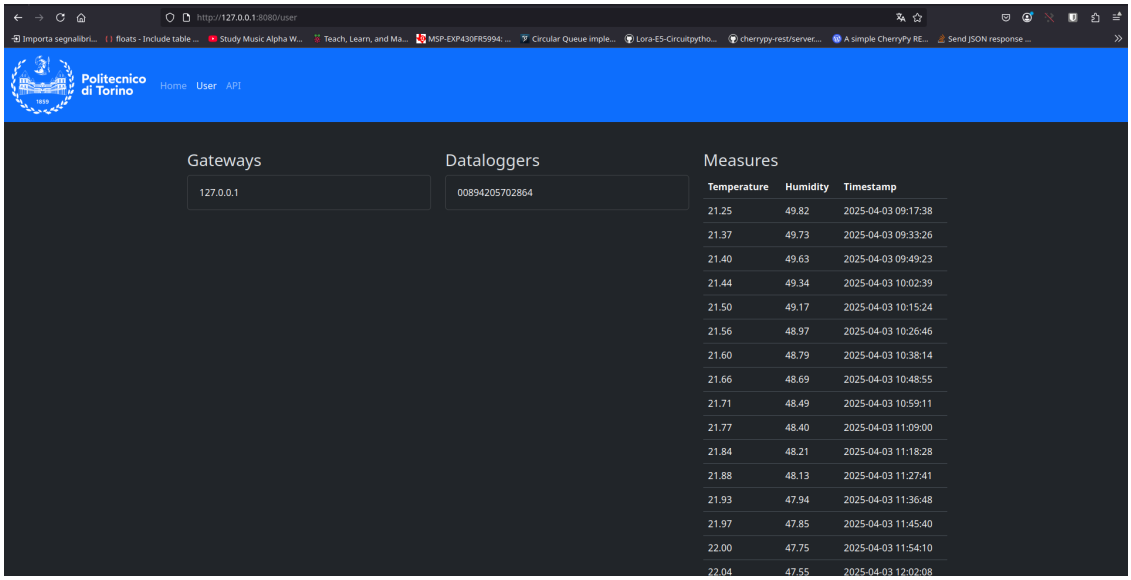


Figure 9.7: A user can display the available data with a Browser.

9.3 Data analysis

Results of the experiment demonstrate that the batteryless datalogger is able to save and restore the context correctly.

To perform the data analysis, we exploited data collected by the Web service and stored into the database as well as the Measure.txt file. The latter gave us the chance of producing the graphs shown in figure 9.8.

The experiment lasted ten hours roughly, where the datalogger captured and sent to the gateway **ninety-two** measures in total (NFR6). Anytime the datalogger resumed, it firstly restored the context, then, it captured the room's temperature and humidity to be immediately sent to the gateway (NFR4). When the upcoming power failure occurred, it

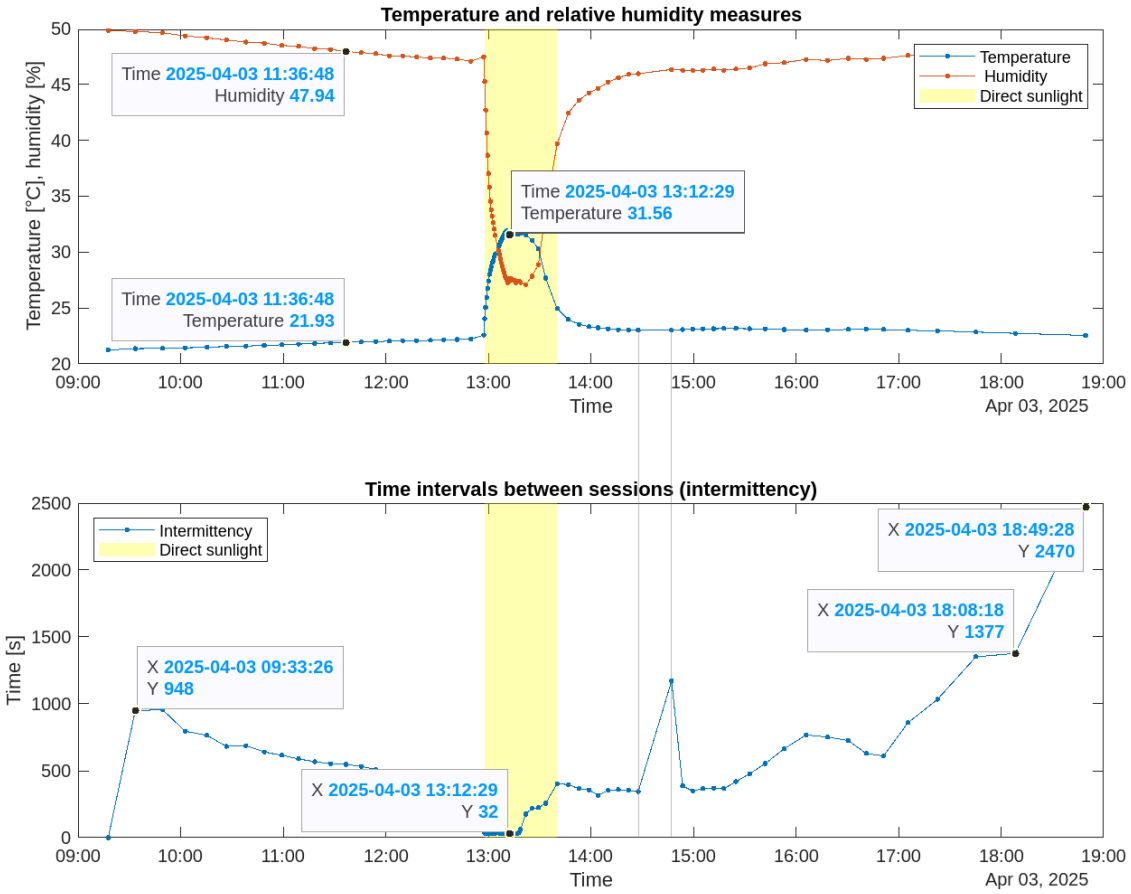


Figure 9.8: Results obtained thanks to data analysis.

saved the context in order to achieve a safe shutdown. The working session lasted a few seconds. The yellow zone points out that measures became unreliable due to temperature hotspots, as soon as the sunlight directly hit the datalogger. This happened because the system was not equipped with a suitable package. Nevertheless, the datalogger, with some latency, started again capturing both temperature and humidity correctly when the sunlight no longer hit it directly. Moreover, the bottom part of figure 9.8 reports the intermittent behavior of the datalogger. Each point relates to the number of seconds elapsed between two consecutive sessions. In normal environmental conditions, the system captured a measure every ten minutes approximately. The intermittency became more frequent as soon as the sunlight directly hit the indoor solar panel. Indeed, the datalogger resumed every thirty seconds roughly.

Figure 9.8 also reports two gray straight lines. The first line marks the last collected measure before the server became unreachable. The second line marks the first collected measure after restarting the server. In practice, there was a time gap in which the server was not available. Such issue is known, but we will return to this topic in section 10.1. By the way, the server did not take into account availability (NFR7), confidentiality (NFR8) and integrity (NFR9) as this was an experimental application. Kind of features would have

required much more work that would have been out of the scope. Actually, the gateway was mostly available (NFR6), making the application robust.

Finally, the entire application worked with a Plug&Play strategy (NFR10). This requirement cannot be deduced through this experiment, but we could simulate the presence of more than one gateway as well as more than one datalogger by testing the Web service (§6.6). Also this topic will be resumed in section 10.1.

9.4 Power end energy analysis

To better understand the intermittent behavior of the datalogger shown in figure 9.8, we need to introduce power and energy analysis. Moreover, we should clarify the non-functional requirement NFR5 stating: *"a datalogger reduces as much as possible the power consumption"*.

As we discussed in section 2.2, a metric is a measurable feature of a system implementation. Metrics are not suggestions, but constraints that developers must meet. NFR5 should state something like: *"the power consumption must be less than or equal to* 100 mW*"*. However, an experimental work can target a metric as outcome. In practice, one of the main objectives might be the evaluation of power and energy consumption of the developed application. This was exactly the case concerning the batteryless datalogger.

We exploited the EnergyTrace++™ technology to perform power and energy analysis. As we briefly discussed in section 4.2.3, this technology can be used to measure the current draw of the MCU as the MSP-EXP430FR5994 LaunchPad™ module fully features it. Using the technology was straightforward with Code Composer Studio IDE. After connecting the datalogger to a USB port, we opened the EnergyTrace++™ technology window by pushing the proper button, that is, a little magnifier symbol. Then, we set the measurement duration to **ten** seconds to simulate a session of the datalogger. Actually, according to the experiment, a real session lasted few seconds on average, certainly less than ten. The chosen value for simulation ensured us to cover most of the cases. Finally, we pushed the Start trace collection button to start collecting data. The IDE shows the results of measurements on a single window with tabs, thus, we will comment them step by step.

Figure 9.9 shows CPU and peripherals states. The CPU worked in active mode as it never entered a low-power mode directly. We cannot see the execution of ctpl_enterShutdown API function as we measured the current draw via USB port; there were no power failures. Indeed, we could not disconnect and reconnect the USB cable to simulate a power loss as we would have stopped the measurement process. The power loss monitor involved the timer TA0, reference voltage REF and ADC12_B peripherals, as we saw in section 8.3. The processor worked at 1 MHz, whereas the ADC12_B trigger frequency was set to 1 kHz. That's why such peripherals were mostly used. eUSCI_B2 and eUSCI_A3 peripherals related to the I2C and UART communication respectively. They were used to capture and send the measure to the gateway. As you can see, their execution time is very short compared to other peripherals, but they had a significant impact on power consumption as we will see briefly. An important remark. What we measured, concerned the startup of the datalogger involving the code executed once, followed by the capture and send of the measure, as we discussed in section 8.2. By the way, when deployed as batteryless, after resuming, the datalogger performs similar operations, as we saw in section 8.5.4. In

practice, the power consumption is comparable with the one we are analyzing here.

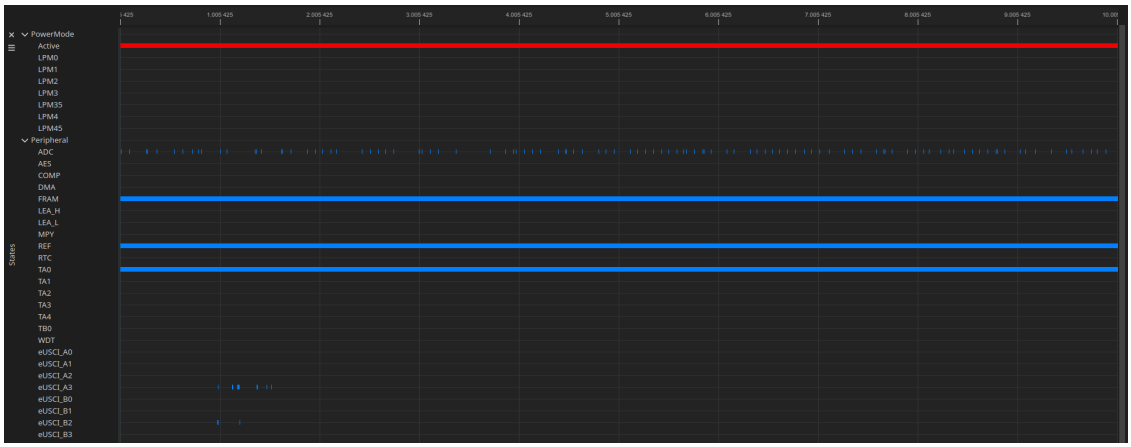


Figure 9.9: EnergyTrace++™technology: CPU and peripherals states.

Figure 9.10 reports the current draw profile. The reader may immediately guess that the peak was related to the transmission of the measure to the gateway. Although low-power, the LoRaWAN® technology is challenging for **ultra low-power** batteryless applications like the one we developed. We also remember that we were working in TEST mode which is less power-demanding than OTAA, as we will discuss in section 10.1.

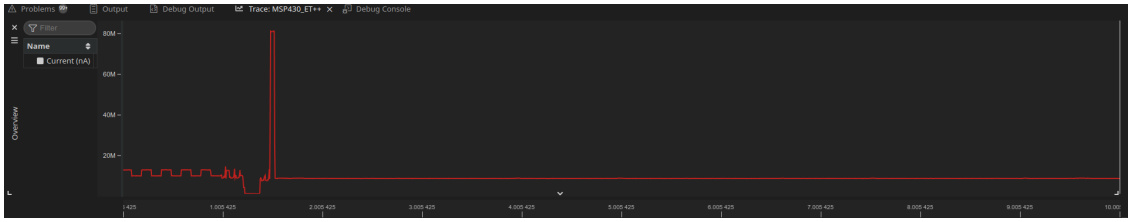


Figure 9.10: EnergyTrace++™technology: current draw profile.

The best way for understanding the current draw profile is to compare it with CPU and peripherals states. Let us refer to figure 9.11. In the first part of the execution, the datalogger performed startup operations; the current consumption ranged from 9.90 mA to 12.90 mA. Then, the LoRaWAN® module was initialized and put in SLEEP mode. As you can see, when the module entered the SLEEP mode, the current draw was 1.05 mA which represented the minimum value. As soon as the measure became available, it was acquired and sent to the gateway. The transmission over the LoRaWAN® network required a significant amount of current whose peak was 81.17 mA which represented the maximum value. After sending the measure, the datalogger waited for the upcoming shutdown consuming 8.60 mA as no tasks were scheduled, but the scheduler was ready.

Figure 9.12 shows the summary. In addition to the data we have mentioned, the summary reported other useful values. The total energy consumption was 301.41 mJ. The minimum power consumption was 2.97 mW, whereas the maximum was 268.21 mW. The relevant data is about the mean, that is **30.12** mW, that we could consider as main outcome.

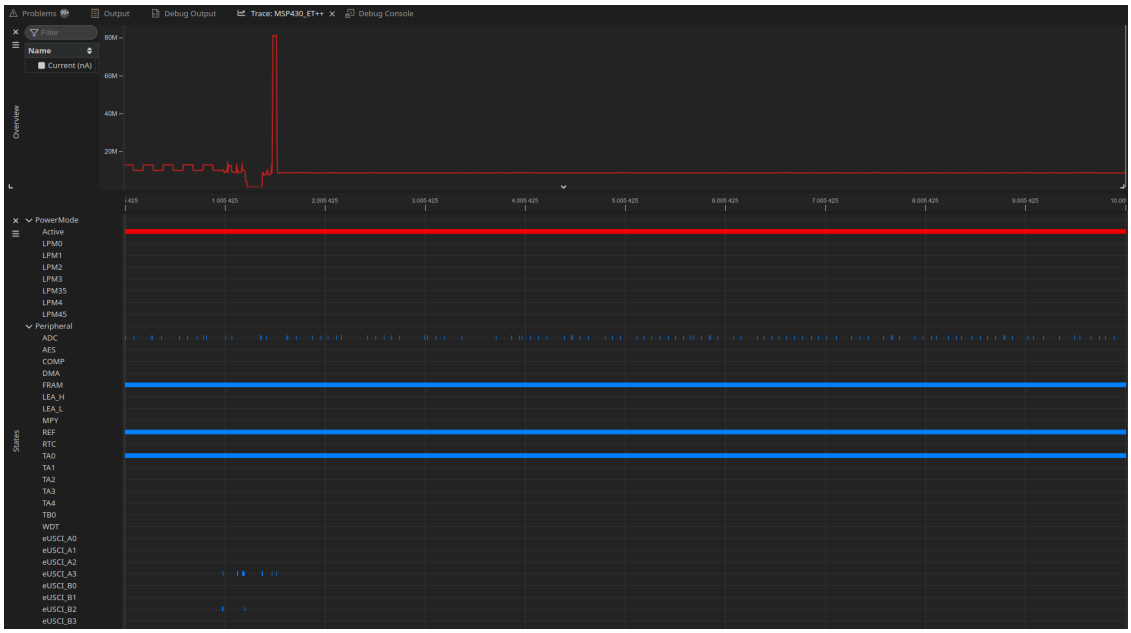


Figure 9.11: EnergyTrace++™technology: overview.

The same consideration holds for current mean: **9.12** mA. Finally, the EnergyTrace++™ technology also reported the estimated life of an hypothetical CR2032 battery powering the system. This information might be interesting for future work, but does not target batteryless applications.

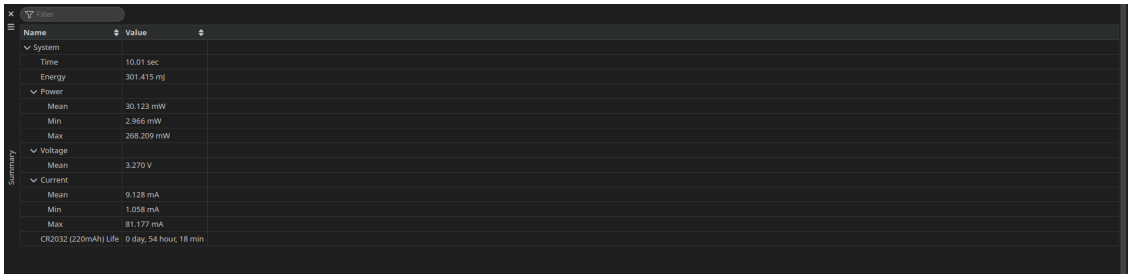


Figure 9.12: EnergyTrace++™technology: summary.

The last point we would like to discuss here is about the interval between sessions. Table 9.3 reports useful values for data analysis that can be exploited now. Let us call $V_{init} = 3.3\,V$ the initial voltage of the super-capacitor. Moreover, let us recall what we saw in section 4.5.1 about the energy harvesting module threshold voltage levels working as protection levels, that is:

- $V_{ovch} = 4.50\,V$ is the maximum voltage accepted on the super-capacitor before disabling the boost converter;
- $V_{chrdy} = 3.92\,V$ is the minimum voltage required on the super-capacitor after a cold start before enabling the LDOs;

- $V_{ovdis} = 3.60\,V$ is the minimum voltage accepted on the super-capacitor before considering the storage element as depleted.

The energy harvesting module will enable LDOs the first time, after the deployment, when the voltage on the super-capacitor reaches V_{chrdy}. As a consequence, the datalogger resumes. Let us define ΔV_0 as:

$$\Delta V_0 = V_{chrdy} - V_{init} = 3.92\,V - 3.30\,V = 0.62\,V \tag{9.1}$$

After resuming, the datalogger will start consuming power and the solar panel charging capability will not prevail on the load's power consumption. As a consequence, the voltage on the super-capacitor will start decreasing. When the voltage reaches V_{ovdis}, after 600 ms, LDOs will be disabled. The datalogger will perform the shutdown, according to the mechanism we studied in section 8.3, when the voltage reaches the SVS TH voltage. Thus, approximately further 100 mV are lost in the process. Let us define ΔV as:

$$\Delta V = V_{chrdy} - (V_{ovdis} - 0.1V) = 3.92\,V - 3.50\,V = 0.42\,V \tag{9.2}$$

Thanks to the previous simple equations, we are now able to better understand why the first measure required roughly forty minutes to be captured. The time obviously depended on environmental conditions, but $\Delta V_0 > \Delta V$ and the super-capacitor had to store more energy to reach V_{chrdy}. The second measure was captured after 948 s as environmental conditions were still poor. The higher the brightness, the shorter the interval between two consecutive sessions, as shown in figure 9.8. In particular, when the sun hit the solar panel directly, the interval between sessions lasted thirty seconds roughly, although ΔV was always the same; the super-capacitor was charged more rapidly. When the datalogger resumed, the energy stored into the super-capacitor was:

$$E_{ready} = \frac{1}{2} C V_{chrdy}^2 = \frac{1}{2} \cdot 220\,mF \cdot (3.92\,V)^2 = 1690.30\,mJ \tag{9.3}$$

whereas, after shutdown:

$$E_{shutdown} = \frac{1}{2} C (V_{ovdis} - 0.1\,V)^2 = \frac{1}{2} \cdot 220\,mF \cdot (3.50\,V)^2 = 1347.50\,mJ \tag{9.4}$$

The difference represented the **available energy** of a session:

$$E_{available} = E_{ready} - E_{shutdown} = 1690.30\,mJ - 1347.50\,mJ = 342.80\,mJ \tag{9.5}$$

The result shown in figure 9.12 about energy, that is 301.41 mJ, is coherent with our analysis as the energy used to perform the mission must be:

$$E \leq E_{available} \tag{9.6}$$

In this chapter, we have reported the main results we achieved thanks to this work. In particular, we have compared the contents of the requirements document with the outcome of an experiment. In the last chapter, we will discuss about the future work and draw conclusions.

Chapter 10

Conclusions and Future Works

In this chapter, we will suggest possible future works to developers who will be going to face a similar project or will be able to update this project. Lastly, we will draw conclusions.

10.1 Future work

10.1.1 Datalogger

We implemented a *proof-of-concept* batteryless intermittent datalogger using commercial devices. Indeed, we decided to *buy* rather than *make*. Anyway, the embedded system may be improved both from the hardware and firmware point of view.

Hardware

The very basic improvements concern the possibility of:

- selecting from the market a more efficient **indoor photovoltaic panel** to harvest more energy from the surrounding environment;
- selecting from the market a larger **dual-cell super-capacitor** to store more energy.

These components mainly target the energy harvester module we presented in section 4.5. Two goals might be achieved. First, the **intermittency** or the number of seconds elapsed between two sessions, may decrease; the system would resume more times than the original solution. Second, the **uptime** or the duration of a session, may increase. We remember the reader that the ASAP computing strategy does not exploit the **status pins** of the energy harvester module whose role is reported here again:

- STATUS[0]: alerts the user that LDOs are operational and can be enabled;
- STATUS[1]: alerts the user around 600 ms before the shutdown of the module;
- STATUS[2]: alerts the user when a MPPT calculation is being performed.

Actually, as discussed is section 4.6, we implemented a circuit adapter to interface STATUS[1] pin to P4.2 of MCU to have a clean CMOS logic level. The same adapter could be used for STATUS[0] and STATUS[2].

In general, the modules implementing the datalogger's sub-systems are updated, but new improved versions might be available in the market. In this case, it may be wise to upgrade the current solution.

Firmware

A longer uptime might allow developers to improve the firmware. Let us give some hint:

- the **status pins** of the energy harvester module may be somehow exploited. In particular, the STATUS[1] signal may be used to perform a clean system *shutdown*. A Task_manageShutdown task may be developed to activate the **shutdown manager**, a suitable component targeting such activity. The Node/application folder contains the shutdown_manager.c file, a template that the reader might fill in the future. Furthermore, the code already implements the P4.2 ISR to deal with STATUS[1] interrupt.

- we implemented the POLICY_ONCE **policy** which captures and sends exactly one measure over the LoRaWAN® network during a session. The POLICY_CONTINUOUS policy would use the TA1 timer module with the aim of performing more measures during the same session. Indeed the timer, working in *continuous* mode, could be started and stopped as needed. The code already implements the WorkloadManager_startTimer and WorkloadManager_stopTimer functions as well as the WorkloadManager_applyPolicy to apply a given policy. Furthermore, the code already implements the TA1 ISR to deal with TA1 interrupt. The reader might obviously implement a more sophisticated policy, being aware that it may require a longer uptime;

- the **workload manager** initializes the system with a given frequency and applies a simple policy. This unit may be highly exploited to improve the performance. As an example, we implemented the WorkloadManager_setFrequency function capable of changing the current working frequency at runtime. The workload manager would be activated thanks to the Task_manageWorkload task scheduled as needed. We point out that a more sophisticated workload manager may require a deep knowledge of the target MCU to exploit its features, such as low-power modes we presented in section 8.3.1;

- we developed the ASAP **computing strategy** (§8.5.4). Moreover, we introduced the *control-based* (§8.5.2) and *control-based with deferring* (§8.5.3) computing strategies that may be modified to target a new feasible solution. The execution of a more sophisticated strategy would schedule the Task_manageWorkload task regularly and the Task_manageShutdown task once, before the upcoming shutdown;

- we added a 220 µF/16V **electrolytic capacitor** close to the Grove Wio-E5 module to enable the CTPL API by TI. As discussed in section 8.3.2, when a power loss is detected, the device invokes the ctpl_enterShutdown function which saves the application and peripheral state and waits for the device to enter BOR with a 64 ms timeout.

In particular, the context is saved and the system shutdown is performed, if the voltage gets lower than SVS TH before the timeout expiration. The crucial point relates to ramp conditions as the supply voltage may ramp down too slowly or too quickly. In our case, it ramp down too quickly, but the capacitor solved the problem. A change in the circuit may require a different value for the capacitor according to new ramp conditions.

Real-time operating system (RTOS)

The firmware might be totally based on a *real-time operating system (RTOS)*. Among many possibilities, we suggest using **TI-RTOS** by TI. According to developers,

> "TI-RTOS is a scalable, one-stop embedded tools ecosystem for TI devices. It scales from a real-time multitasking kernel (SYS/BIOS) to a complete RTOS solution including additional middleware components and device drivers. By providing essential system software components that are pre-tested and preintegrated, TI-RTOS enables developers to focus on creating the application"[24].

We created a TI-RTOS based project, called NodeRTOS (§A.1.5), into the Datalogger folder. We point out that we did not implement anything, but we just configured the project and added the libraries we used in the Node solution. The idea was to help the reader to enter the topic in a more friendly way and start developing an RTOS-based solution from scratch. Such solution might increase performance as RTOSs have powerful schedulers with priority and can create and handle tasks easily. Furthermore, many resources are available, such as semaphores and software interrupts. The reader will start using the NodeRTOS project as a template, with the goal of implementing a new solution to be compared with the one we presented in this work of thesis. TI-RTOS might better exploit the system's uptime and increase the overall performance.

Nonetheless, a RTOS increases the complexity of the final solution as it introduces a further level of abstraction. This layer should interface with MSP430 Peripheral Driver library and MSP MCU FRAM utilities which are already high level programming methods. The overall complexity might affect the Principles for Low-Power Applications we introduced in section 8.6. In particular, the number of function calls through APIs may worsen instead of improve performance. After implementing a feasible RTOS-based solution, developers should compare results with the ones achieved in this work (§9.3). Our optimized application could be used as a *golden model* to evaluate differences, benefits and drawbacks.

New solution

The last point concerns a new solution. Our work might be used as a starting point for performing a new *make-or-buy* decision, as we discussed in section 4.1. The system may be redesigned by substituting one ore more components or sub-systems with the aim of increasing performance. We highly exploited a modular approach, thus substituting any component should not be complicated, although it may require writing new code or additional electronics.

The best, but most complicated solution, would target a printed circuit board (PCB) hosting SMD components or basically exploiting the surface mount technology (SMT). Requirements should carefully describe the target embedded system with metrics allowing

developers to evaluate optimizations and trade-offs. As a trivial but meaningful example, all LEDs should be connected via jumpers to be opened during the deployment. LEDs power consumption is usually negligible, but batteryless systems must avoid wasting power. Obviously, such solution requires software and hardware skilled developers targeting a highly optimized embedded system which might be used by researchers to conduct their studies.

As a final suggestion, a PCB might be developed to easily substitute the matrix-board we used in our project. This choice would target a more robust embedded system interconnecting modules without using wire-wrap and soldering methods. Moreover, the board may be designed with the goal of hosting other useful electronic components as well as updating modules. This solution may run our code or a RTOS-based one. There is room for creativity to improve the entire system.

10.1.2 Gateway

Our batteryless datalogger might be *potentially* able to join a LoRaWAN® network via OTAA mode we introduced in section 7.3.4. As usual, the key point is about the system's uptime. Joining a network is time and power-hungry operation as a sequence of commands is required. If the reader will be able to increase the uptime, the possibility of joining a network will become real and new scenarios will open.

First of all, the datalogger will totally exploit the LoRaWAN® protocol features. We introduced LoRaWAN® in section 7.1 and we saw that it operates in unlicensed radio frequency bands and is known for its long-range capabilities (up to 10 km in rural areas), low power consumption, and the ability to connect a large number of devices to a single network. The architecture includes end devices, gateways, network servers, and application servers, enabling a scalable and flexible network for diverse IoT applications. In practice, the datalogger would work as **end device** joining a selected LoRaWAN® network.

Now the reader should have understood why we stated our batteryless datalogger might be *potentially* able to join a network. We wrote the code to perform this activity, but we could not try it. One reason was the system's uptime, but the main reason concerned the gateway. In section 7.2, we saw that our solution was based on a *pseudo-gateway* working in TEST mode. This choice did not affect our experiments, but it is less scalable. To proceed, there are basically two solutions. The most common one is to *buy* an already existing **LoRaWAN® gateway** from the market. Gateways are expensive as they work as data concentrators like IP routers. They usually have more than one channel and are managed by a complex software or an operating system. The second solution is to *make*. In this case, both hardware and software have to be carefully designed to be compliant with LoRaWAN® specifications. The reader might find open source projects out there, although the available documentation is tricky and poor. However, regardless the choice, the gateway requires a *configuration* to be deployed into the network as data concentrator with the aim at receiving measures from any datalogger with a **many to one** relationship.

Finally, our gateway could not return the **timestamp** to the datalogger. Again, this feature might be possible by joining the LoRaWAN® network via OTAA mode. In this case, after joining the network, datalogger and gateway could easily exchange data such as the timestamp to be added to the measure. At this point, the datalogger would **store measures** in its internal memory performing the datalogging process. The policy of storing data in FRAM would open new scenarios to developers.

To sum up, our solution was fine for experimental purposes and did not affect our goals. We suggest the reader to start reading the LoRaWAN® documentation to choose or design a fully-compliant LoRaWAN® gateway. This knowledge will help developing the *right* application.

10.1.3 Server

We developed a REST Web service to deal with more than one gateway with a relationship many to one. The Web service could be configured to run into any server. CherryPy technology allowed us to implement a solid and reliable software. However, we do not want to bore the reader with other details as everything was explained in chapter 6.

Actually, the service might be improved to deal with more than one gateway. In particular, a more sophisticated code may be written with CherryPy technology to better handle HTTP requests. The same holds for GUI as we implemented simple pages which might be enriched with more data and controls. Furthermore, pages should be slightly modified to better display data. Also in this case, smart and skilled programmers may use their creativity and knowledge to implement a more responsive and sophisticated Web service.

10.1.4 Final remarks

As a final recommendation, we suggest again the reader to start reading the LoRaWAN® documentation to understand how to set up a complete LoRaWAN® application as we discussed in section 7.1.2. Basically, a batteryless intermittent datalogger would work as *end system* or *end node* joining a network thanks to a selected LoRaWAN® gateway working as *concentrator*. The *network server*, managing the entire network, and the *application server*, securely processing application data, would close the chain. End nodes, gateways, network server and application server constitute a typical LoRaWAN® network architecture. What really matters is to implement one ore more batteryless dataloggers able to join a LoRaWAN® network. The rest depends on the user needs, such as using one or more *indoor* or *outdoor* LoRaWAN® gateways and whom is providing the network server and the application server.

10.2 Conclusions

We implemented an **ultra low-power** LoRaWAN® application whose main actor is a batteryless intermittent environmental datalogger. Experimental results demonstrated that, during a session lasting between five and ten seconds on average, the datalogger's power consumption mean is roughly **30 mW**. Furthermore, the context is saved and restored correctly.

We did our best to explain the development process step by step with a good level of detail, without omitting any important information.

We wish this work is useful for those who will be going to specify, design and implement batteryless systems. Such background might be exploited to target new solutions. The field of research is very wide, but technology improvements, more sophisticated algorithms and new scientific discoveries will certainly open up new horizons for future developers.

Appendix A

Development tools

The purpose of this appendix is to guide the reader to correctly install and set up the development tools we used for the project. Such activities are not complex, at least for electronic and computer engineers, but the main goal is to ensure that all the packages and drivers required by the project are managed successfully.

We based the development on the following setup:

- **Ubuntu**[1]Linux® - version **24.04** (Operating System);
- **Code Composer Studio**[2]- version **20.0.1** (IDE);
- **Visual Studio Code**[3]- version **1.96.2** (IDE).

Although we will mainly refer to an Ubuntu Linux® distribution, all the installation steps can be easily performed on Windows®and macOS® platforms.

A.1 Code Composer Studio

We used **Code Composer Studio (CCS)** by Texas Instruments (TI) to write the firmware of the datalogger we discussed in chapters five and eight. According to developers,

> "it is an integrated development environment for TI's microcontrollers and processors. It is comprised of a rich suite of tools used to build, debug, analyze and optimize embedded applications. CCS is available for download across Windows®, Linux® and macOS® platforms [15]."

It can also be used in the cloud, but we will not take into account this option.

CCS is built on top the Eclipse® Theia framework which leverages some of the same components that power VS Code®. As a consequence, VS Code® users will find the CCS interface quite familiar.

---

[1] https://ubuntu.com.

[2] https://www.ti.com/tool/CCSTUDIO.

[3] https://code.visualstudio.com/.

A.1.1 System Requirements

It is strongly advised to have a system that exceeds the recommended requirements reported in table A.1. While CCS will run on a system that meets the minimum requirements, the performance will likely be poor. This situation might be frustrating even for patient developers.

	Memory	Disk space	Processor
Minimum	8 GB	3 GB	x86_64 multi-core
Recommended	16 GB+	6 GB+	x86_64 multi-core (8+ threads)

Table A.1: CCS: hardware requirements [16].

CCS is a **64-bit** application. Table A.2 reports the officially supported and tested operating systems. Please note that CCS will run on other Linux® distributions, but they have not been tested.

Linux®	Windows®	macOS®
Ubuntu Linux® 24.04 64-bit	Windows® 11 64-bit	macOS® 15 (Sequoia)
Ubuntu Linux® 22.04 64-bit	Windows® 10 64-bit	macOS® 14 (Sonoma)
Ubuntu Linux® 20.04 64-bit		macOS® 13 (Ventura)

Table A.2: CCS: supported operating systems [16].

A.1.2 Installation

Since the installation process is the same across Windows®, Linux® and macOS®, we will perform the required steps as described in the **CCSv20** user's guide [16].

We are going to focus on a Linux® Debian and Ubuntu based distribution. Here are the steps:

1. visit the product webpage at the following address, scroll down the page and press the *Download options* button concerning CCSTUDIO:
   https://www.ti.com/tool/CCSTUDIO#downloads.
   Download the Linux single file (offline) installer for Code Composer Studio IDE (all features, devices), the installation image for CCS. Extract the archive file before proceeding with the installation. For Linux® users it is recommended to install as a **normal user** and not sudo/root. Thus, open a terminal and run the installer from the extracted folder:

   ```
   ./ccs_setup_20.0.1.00004.run
   ```

   The CCS installation is dependent on other software packages. For Windows® and macOS® users these packages will be installed automatically as part of the installation.

For Linux® users, the installer will attempt to identify which packages are missing and display the list;

2. accept the license agreement to continue;

3. since the installer performs a system check, you are required to install the missing dependencies. Open another terminal and perform the following steps:

   (a) update the system:

   ```
   sudo apt update
   ```

   (b) solve the 32-bit library dependency check:

   ```
   sudo apt install libc6-i386
   ```

   By the way, libc6-i386 is only required when trying to install additional older compilers that use a 32-bit installer, otherwise this dependency can be ignored;

   (c) solve the USB dependency check:

   ```
   sudo apt install libusb-0.1-4
   ```

   (d) solve the libtinfo dependency check:

   ```
   wget http://security.ubuntu.com/ubuntu/pool/universe/n/ncurses/
      libtinfo5_6.3-2ubuntu0.1_amd64.deb
   sudo apt install ./libtinfo5_6.3-2ubuntu0.1_amd64.deb
   ```

   Note that Ubuntu® 24.04 comes with libtinfo6. CCS requires libtinfo5, which is obsolete with 24.04 version. This package is required by the TI Arm Clang Compiler. If this compiler will not be used, then resolving this dependency is optional;

   (e) solve the Python dependency check:

   ```
   sudo apt install software-properties-common
   sudo add-apt-repository ppa:deadsnakes/ppa
   sudo apt install python3.9
   sudo apt install libpython3.9
   ```

4. at this point, by pushing the *Back* button in the installation window, accepting the agreement again and pushing the *Forward* button, you will notice that everything is *OK* and you can move forward with the installation process. If you experience other issues, although this should not happen, try to browse the web to find possible solutions;

5. choose a location to install CCS. This location must not already contain a CCS installation. If multiple versions of CCS will be installed, then, it is recommended to install them in versioned folders. We suggest **/home/user/ti** as a location;

6. choose the desired device families to install support for. You might check all the boxes;

7. push the *Forward* button to start the installation process. As the installation proceeds, the progress screen will show what the installer is doing. Depending on how many device families were selected the installation may take a significant amount of time;

8. then, as a post installation step, it is necessary to run the driver install script as **sudo/root** user after the installation is complete. Run:

```
sudo ./<CCS_INSTALL_DIR>/ccs/install_scripts/install_drivers.sh
```

9. finally, if Ubuntu® 24.04 is being used, the below additional steps are required. Browse to: <CCS_INSTALL_DIR>/ccs/theia and run the below commands:

```
sudo chown root chrome-sandbox
sudo chmod 4755 chrome-sandbox
```

A.1.3 Libraries

The firmware needs two important libraries that have to be downloaded from TI's website: MSPDRIVERLIB and MSP-FRAM-UTILITIES. Here we report two possible solutions.

Solution 1 (highly suggested)

1. run CCS;

2. open the Resource Explorer: *View* → *Resource Explore*. CCS will open a new tab;

3. unfold the MSP430® microcontrollers folder;

4. unfold the Embedded Software folder;

5. download and install the **MSP430Ware - x.xx.xx.xx** tool by clicking the *Download and install* button you should find at the upper right corner. The tool should be installed into the **ti** folder, the one in which you already installed CCS. MSP430Ware is a tool for navigating the available resources that support MSP430 and MSP432 devices. It contains libraries and precious code examples.

Now, your file system should look like:

- /home/user/ti
  - ccs2001
  - housekeeping_example_pack_1_00_08
  - msp430ware_3_80_14_01

Solution 2

Another solution is to directly download the libraries from their repositories:

- MSPDRIVERLIB
  1. visit: https://www.ti.com/tool/MSPDRIVERLIB;
  2. scroll down the webpage and select the DriverLib for MSP430 devices. In particular, download the BSD Licensed Driver Library for MSP430 zipped file;
  3. unzip the file by auto-detecting the sub-folder;
  4. copy the latter into the **ti** directory, the one in which you should see the CCS installation folder.
- MSP-FRAM-UTILITIES
  1. visit: https://www.ti.com/tool/MSP-FRAM-UTILITIES;
  2. scroll down the webpage and download the FRAM Utilities Linux Installer. You may need your own TI's account. If you do not have one, please create one from scratch and save your credentials;
  3. open a terminal targeting the folder in which the file is located and give it the execution permission:

     ```
     sudo chmod 764 FRAMUtilities_03_10_00_10_setup.run
     ```

  4. run the installer:

     ```
     sudo ./FRAMUtilities_03_10_00_10_setup.run
     ```

  5. accept the agreement;
  6. select **ti** as parent directory and perform the installation.

  Now, your file system should look like:

  – /home/user/ti
    – ccs2001
    – FRAM-Utilities_03_10_00 _10
    – msp430_driverlib_2_91 _13_01

A.1.4 TI-RTOS

In chapter ten, we introduced the possibility to write a RTOS-based firmware for the datalogger. This section guides the reader to start setting up the environment for Linux® Debian and Ubuntu based distributions. The reader might easily perform the same setup for Windows® and macOS® operating systems by visiting the product website.

You should perform the following steps to set up the environment:

1. visit the webpage to download the TI-RTOS — Real-Time Operating System (RTOS) provided by Texas Instruments:
   https://www.ti.com/tool/TI-RTOS-MCU

   Scroll down the page and push the *Download options* button;

2. select the **2.20.00.06 - 22 Jun 2016** version for **MSP430**;

3. after downloading, change the file permissions:

   ```
   sudo chmod 764 tirtos_msp43x_setuplinux_2_20_00_06.bin
   ```

4. install TI-RTOS:

   ```
   ./tirtos_msp43x_setuplinux_2_20_00_06.bin
   ```

   You will have to accept the agreements and choose the installation directory. We suggest using the same of CCS: /home/user/ti. Press *Enter* and wait for the installation to complete;

5. visit the following webpage:
   https://www.ti.com/tool/download/MSP-CGT/16.12.0.STS
   to download and install a proper version of the compiler. The required TI compiler version for TI-RTOS is **16.12.0.STS**;

6. download the Linux installer for MSP430 CGT;

7. after downloading, change the file permissions:

   ```
   sudo chmod 764 ti_cgt_msp430_16.12.0.STS_linux_installer_x86.bin
   ```

8. install the compiler:

   ```
   ./ti_cgt_msp430_16.12.0.STS_linux_installer_x86.bin
   ```

   You will be asked to choose the destination directory. Please select the one in which you installed CCS, but give a suitable name to the final directory like /home/user/ti/ti-cgt-msp430_16.12.0.STS. Press *y* and wait for the installation to complete.

Now, your file system should look like:

- /home/user/ti
  - ccs2001
  - housekeeping_example_pack_1_00_08
  - msp430ware_3_80_14_01
  - ti-cgt-msp430_16.12.0.STS
  - tirtosmsp43x_2_20_00_06
  - xdctools_3_32_00_06_core

A.1.5 Datalogger setup

In this section, we will see how to set up CCS to correctly build the firmware. The **Node** folder relates to the firmware implementation of the datalogger we discussed in chapters five and eight. **NodeRTOS** is a TI-RTOS-based template that can be used for future work.

Node

1. clone the batteryless-data-logger repository from GitHub:
   git@github.com:eml-eda/batteryless-data-logger.git
2. run CCS;
3. import the code: *File* → *Import Project(s)*. Browse:
   .../batteryless-data-logger/Project/Datalogger/Node
   Click *Ok*, then *Finish*;
4. delete the driverlib and fram-utilities folders;
5. right-click on the root project folder (Node) and create again the driverlib and fram-utilities folders. This operation is necessary to correctly set up their buildpath. Be aware of using exactly those names;
6. update the driverlib folder. Right-click on driverlib folder and select *Add Files/Folders...* then click the *plus* button. Choose *Select folders to copy* as option and press the three dots for selecting the path. Target:
   ...ti/msp430ware_3_80_14_01/driverlib/driverlib/MSP430FR5xx_6xx
7. update the fram-utilities folder. Right-click on fram-utilities folder and select *Add Files/Folders...* then click the *plus* button. Choose *Select folders to copy* as option and press the three dots for selecting the path. Target:
   .../ti/msp430ware_3_80_14_01/fram_utilities/src/ctpl
8. repeat the same operation to update the fram-utilities folder by targeting:
   .../ti/msp430ware_3_80_14_01/fram_utilities/src/lz4
   .../ti/msp430ware_3_80_14_01/fram_utilities/src/nvs
   .../ti/msp430ware_3_80_14_01/fram_utilities/src/rng
   Although unused in this project, these libraries might be targeted for future work;
9. focus on the **ctpl** sub-folder located into the fram-utilities folder in the project:
   (a) delete the devices sub-folder;
   (b) delete the ctpl_pre_init.c file;
   (c) edit the ctpl_low_level_macros.asm file. Modify, line **35** and line **54**:

```
clr.b   &DMA0CTL_L      ; sw trigger, channel 0
clr.b   &DMA0CTL_L      ; sw trigger, channel 0
```

   as follows:

```
clr.w   &DMA0CTL        ; sw trigger, channel 0
clr.w   &DMA0CTL        ; sw trigger, channel 0
```

These modifications are crucial as they are part of the code performing copy and fill with either CPU or DMA to handle the context. Such issue prevented the correct compilation of the code due to a typo;

10. toggle the toolbar: *View* → *Toggle Toolbar*;

11. right-click on the root folder (Node) and clean the project: *Clean Project(s)*;

12. build the project. Push the *Build Project(s)* button in the toolbar. The code should be built with no errors;

13. CCS is a smart IDE. It will open a *Get Started* tab stating that no compatible board is detected. Connect the MSP-EXP430FR5994 LaunchPad™ Development Kit to a USB port of your host PC. The board will be detected;

14. last point relates to the debug activity. Right-click on the project root folder (Node) an select *Properties*. You should tick the *Manage the project's target-configuration automatically* box. CCS will create a .theia folder containing the launch.json file. You are now ready to *debug* and *flash* the project.

NodeRTOS

1. run CCS;

2. import the code: *File* → *Import Project(s)*. Browse:
   .../batteryless-data-logger/Project/Datalogger/NodeRTOS
   Click *Ok*, then *Finish*;

3. setup CCS: *File* → *Preferences* → *Code Composer Studio settings*. Add to the *Product discovery-path* the TI-RTOS installation folders:
   .../ti/tirtos_msp43x_2_20_00_06
   .../ti/xdctools_3_32_00_06
   Please add these two folders to the project *Product dependencies*: Right-click on the root project folder (NodeRTOS) → *Properties* → *Dependencies*;

4. everything should work. If you experience any issue, try to solve it by repeating the steps 4 to 14 we have performed for the Node project.

A.1.6 Extensions

It is highly suggested to install the **Doxygen Documentation Generator**. This can be easily done by using the *Extensions* manager in CCS and searching for the tool. You will be able to generate Doxygen comments in VS Code as we discussed in section 5.4.

A.2 Visual Studio Code

We used **Visual Studio Code (VS Code)** by Microsoft® to develop the gateway and Web service we discussed in chapters six and seven. According to developers,

> "VS Code is a cross platform free code editor, which runs on the macOS®, Linux®, and Windows® operating systems. It is lightweight and should run on most available hardware and platform versions. Getting up and running with VS Code is quick and easy. It is a small download so developers can install in a matter of minutes and start working [48]."

A.2.1 Installation

Linux® Debian and Ubuntu based distributions

The easiest way to install VS Code for Debian/Ubuntu Linux® based distributions is to:

1. download the .deb package (64-bit) from the product website. The file will be saved in your *home* directory after clicking the proper button;
2. move to the directory in which you saved the file;
3. perform the installation process:
   - **option 1** - right-click on the file and select a manager such as Discover;
   - **option 2** - right-click on the file and select *Open with QApt package installer*;
   - **option 3** - open a terminal and perform the following command:

     ```
     sudo apt install ./<file>.deb

     # For older Linux distributions, you will need to run this
         instead:
     # sudo dpkg -i <file>.deb
     # sudo apt-get install -f # Install dependencies
     ```

     Installing the .deb package will automatically install the apt repository and signing key to enable auto-updating using the system's package manager;
   - **option 4** - the repository and key can also be installed manually with the following script:

     ```
     sudo apt-get install wget gpg

     wget -qO- https://packages.microsoft.com/keys/microsoft.asc | gpg
         --dearmor > packages.microsoft.gpg

     sudo install -D -o root -g root -m 644 packages.microsoft.gpg /
         etc/apt/keyrings/packages.microsoft.gpg

     sudo sh -c 'echo "deb [arch=amd64,arm64,armhf signed-by=/etc/apt/
         keyrings/packages.microsoft.gpg] https://packages.microsoft.
         com/repos/code stable main" > /etc/apt/sources.list.d/vscode.
         list'

     rm -f packages.microsoft.gpg
     ```

Windows®

The easiest way to install VS Code for Windows® is to:

1. download the Visual Studio Code installer for Windows® from the product website;
2. move to the directory in which you saved the file;
3. run the installer VSCodeUserSetup-version.exe. This will only take a minute.

By default, VS Code will be installed under:
C:\Users\Username\AppData\Local\Programs\Microsoft VS Code.

Setup will add VS Code to your $\%PATH\%$, so from the console you can type code. to open VS Code on that folder. You will need to restart your console after the installation for the change to the $\%PATH\%$ environmental variable to take effect.

A.2.2 Server setup

After installing VS code, you should perform the following steps to set up the environment and work correctly aiming at developing for the Web service:

1. run VS Code;
2. open the .../batteryless-data-logger/Project/Server folder;
3. install the Python extension for VS Code from the *Visual Studio Marketplace*. The Python extension is named Python and it's published by Microsoft®;
4. install a Python interpreter. This step depends on the operating system you are using:
   - **Linux®**. The built-in Python 3 installation on Linux® works well, but to install other Python packages you must install pip with python3-pip. In this case you should open a terminal in VS Code and run the following command:

     ```
     sudo apt install python3-pip
     ```

     Verify that you installed Python successfully on your machine with the following command:

     ```
     python3 --version
     ```

   - **Windows**. Install Python interpreter from *python.org*. You can typically use the *Download Python* button that appears first on the page to download the latest version. Verify that you installed Python successfully on your machine with the following command:

     ```
     py -3 --version
     ```

5. install CherryPy. It is a pure Python library that can run anywhere Python runs and does not require a C compiler. Currently, CherryPy supports Python **3.6** through to **3.11**. If your operating system is running a higher version, please create a virtual environment and select Python **3.11** as target (*View* → *Command Palette...* → type: *Python: Select Interpreter* → *Create Virtual Environment...* → *Venv*). If this version is not available in your system, you can install it:

   ```
   sudo apt install python3.11
   ```

   CherryPy can be easily installed via common Python package managers such as setuptools or pip. We suggest using pip by typing the following command on the terminal:

   ```
   pip install cherrypy
   ```

6. install sqlalchemy:

   ```
   pip install sqlalchemy
   ```

7. install marshmallow_sqlalchemy:

   ```
   pip install marshmallow_sqlalchemy
   ```

After restarting VS Code for completeness, you may run the Python test application we developed, ServerHW.py, in order to verify your setup. Move to Server project folder and run the following command:

```
python3 ServerHW.py
```

The test program starts a server and hosts an application that will be served at request reaching: http://127.0.0.1:8080/. You can easily verify that the test program is perfectly working by opening your preferred Browser at the previous URL.

A.2.3 Gateway setup

For the sake of convenience, it is better to open another VS Code instance to develop the gateway. Then, perform the following steps:

1. open the .../batteryless-data-logger/Project/Gateway folder. You might also create a virtual environment in VS Code to manage libraries (as we have done for the server);

2. install pySerial API. This module encapsulates the access for the serial port. It provides backends for Python running on Windows®, OSX®, Linux®, BSD® (possibly any POSIX compliant system) and IronPython®. The serial module automatically selects the appropriate backend. Please use the following command:

   ```
   pip install pyserial
   ```

3. install requests. It is a Python library allowing developers writing code that interacts with REST APIs. Its main role is to abstract away the complexities of making HTTP requests. Please use the following command:

   ```
   pip install requests
   ```

Appendix B

LoRa-E5 AT Commands

Command	Description
AT	Test command
FDEFAULT	Factory data reset
RESET	Software reset
DFU	Force bootloader to enter dfu mode
LOWPOWER	Enter sleep mode
VER	Version[Major.Minor.Patch]
MSG	LoRaWAN® unconfirmed data
MSGHEX	LoRaWAN® unconfirmed data in hex
CMSG	LoRaWAN® confirmed data
CMSGHEX	LoRaWAN® confirmed data in hex
PMSG	LoRaWAN® proprietary
PMSGHEX	LoRaWAN® proprietary in hex
CH	LoRaWAN® channel frequency
DR	LoRaWAN® datarate
ADR	LoRaWAN® ADR control
REPT	Unconfirmed message repetition
RETRY	Confirmed message retry
POWER	LoRaWAN® TX power
RXWIN2	LoRaWAN® RX window2
RXWIN1	LoRaWAN® RX window1
PORT	LoRaWAN® communication port
MODE	LWABP, LWOTAA, TEST
ID	LoRaWAN® DevAddr/DevEui/AppEui
KEY	Set NWKSKEY/APPSKEY/APPKEY
CLASS	Choose LoRaWAN® modem class(A/B/C)
JOIN	LoRaWAN® OTAA JOIN
LW	LoRaWAN® misc configuration (CDR, ULDL, NET, DC, MC, THLD)
BEACON	LoRaWAN® Class B utilities
TEST	Send test serious command
UART	UART configure
DELAY	RX window delay
VDD	Get VDD
RTC	RTC time get/set
EEPROM	Write/Read EEPROM
WDT	Watchdog control
TEMP	Get Temperature
LOG	Log DEBUG/INFO/WARN/ERROR/FATAL/PANIC/QUIET

Table B.1: LoRa-E5 AT Commands [41].

Appendix C

ASCII table

The ASCII table is fundamental for embedded systems developers. Although we could have selected a picture from the Internet, we preferred reporting the outcome of a simple, but powerful Ubuntu Linux® utility: ascii. This choice might be strange, but modern operating systems allow developers to get the required information easily.

After opening a terminal, the following command will produce the ASCII table in decimal and hexadecimal, as shown in figure C.1:

```
ascii -a
```

```
[redacted]:~$ ascii -a
Usage: ascii [-adxohv] [-t] [char-alias...]
   -t = one-line output  -a = vertical format
   -d = Decimal table  -o = octal table  -x = hex table  -b binary table
   -h = This help screen -v = version information
Prints all aliases of an ASCII character. Args may be chars, C \-escapes,
English names, ^-escapes, ASCII mnemonics, or numerics in decimal/octal/hex.

Dec Hex    Dec Hex    Dec Hex  Dec Hex  Dec Hex  Dec Hex   Dec Hex   Dec Hex
  0 00 NUL  16 10 DLE  32 20    48 30 0  64 40 @  80 50 P   96 60 `  112 70 p
  1 01 SOH  17 11 DC1  33 21 !  49 31 1  65 41 A  81 51 Q   97 61 a  113 71 q
  2 02 STX  18 12 DC2  34 22 "  50 32 2  66 42 B  82 52 R   98 62 b  114 72 r
  3 03 ETX  19 13 DC3  35 23 #  51 33 3  67 43 C  83 53 S   99 63 c  115 73 s
  4 04 EOT  20 14 DC4  36 24 $  52 34 4  68 44 D  84 54 T  100 64 d  116 74 t
  5 05 ENQ  21 15 NAK  37 25 %  53 35 5  69 45 E  85 55 U  101 65 e  117 75 u
  6 06 ACK  22 16 SYN  38 26 &  54 36 6  70 46 F  86 56 V  102 66 f  118 76 v
  7 07 BEL  23 17 ETB  39 27 '  55 37 7  71 47 G  87 57 W  103 67 g  119 77 w
  8 08 BS   24 18 CAN  40 28 (  56 38 8  72 48 H  88 58 X  104 68 h  120 78 x
  9 09 HT   25 19 EM   41 29 )  57 39 9  73 49 I  89 59 Y  105 69 i  121 79 y
 10 0A LF   26 1A SUB  42 2A *  58 3A :  74 4A J  90 5A Z  106 6A j  122 7A z
 11 0B VT   27 1B ESC  43 2B +  59 3B ;  75 4B K  91 5B [  107 6B k  123 7B {
 12 0C FF   28 1C FS   44 2C ,  60 3C <  76 4C L  92 5C \  108 6C l  124 7C |
 13 0D CR   29 1D GS   45 2D -  61 3D =  77 4D M  93 5D ]  109 6D m  125 7D }
 14 0E SO   30 1E RS   46 2E .  62 3E >  78 4E N  94 5E ^  110 6E n  126 7E ~
 15 0F SI   31 1F US   47 2F /  63 3F ?  79 4F O  95 5F _  111 6F o  127 7F DEL
```

Figure C.1: ASCII table.

Bibliography

[1] Saad Ahmed et al. "The Internet of Batteryless Things". In: *Commun. ACM* 67.3 (Feb. 2024), pp. 64–73. ISSN: 0001-0782. DOI: 10.1145/3624718. URL: https://doi.org/10.1145/3624718.

[2] Walter Benenson et al. *Handbook of Physics*. 2nd ed. Springer, 2002, pp. 87, 92–93.

[3] Hur Byul. *Learning Embedded Systems with MSP430 FRAM Microcontrollers: MSP430FR5994 with Code Composer Studio*. 2022.

[4] CherryPy. *CherryPy — A Minimalist Python Web Framework*. [Online - accessed: 23-9-2024]. URL: https://docs.cherrypy.dev/en/latest/.

[5] Passerone Claudio. *Analog and Digital Electronics for Embedded Systems*. 1st ed. CLUT, 2015.

[6] Vito Daniele et al. *ELETTROTECNICA*. 1st ed. Monduzzi editore, 1996, pp. 10–11, 20–23.

[7] Carmen Delgado and Jeroen Famaey. "Optimal Energy-Aware Task Scheduling for Batteryless IoT Devices". In: *IEEE Transactions on Emerging Topics in Computing* 10.3 (2022), pp. 1374–1387. DOI: 10.1109/TETC.2021.3086144.

[8] Cambridge Dictionary. *experiment*. [Online - accessed: 1-5-2025]. URL: https://dictionary.cambridge.org/dictionary/english/experiment.

[9] Diffen. *Data vs. Information*. [Online - accessed: 20-9-2024]. 2024. URL: https://www.diffen.com/difference/Data_vs_Information.

[10] doxygen. *doxygen*. [Online - accessed: 22-10-2024]. URL: https://www.doxygen.nl/.

[11] e-peas. *AEM10941*. Accessed: 17-9-2024. URL: https://e-peas.com/aem10941_datasheet/.

[12] Mohammed Farag. "Lithium-Ion Batteries: Modelling and State of Charge Estimation". PhD thesis. Oct. 2013. URL: http://hdl.handle.net/11375/15253.

[13] Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures". PhD thesis. 2000. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf.

[14] IMPO. *Common Applications for Data Loggers*. [Online - accessed: 20-9-2024]. 2021. URL: https://www.impomag.com/maintenance/article/21342729/5-common-applications-for-data-loggers.

[15] Texas Instruments. *Code Composer Studio*. Accessed: 10-1-2025. 2025. URL: https://www.ti.com/tool/CCSTUDIO.

[16] Texas Instruments. *Code Composer Studio User's Guide*. Accessed: 10-1-2025. 2025. URL: https://software-dl.ti.com/ccs/esd/documents/users_guide_ccs_20.0.0/index.html.

[17] Texas Instruments. *HDC1000*. Download - accessed: 17-9-2024. URL: https://download.mikroe.com/documents/datasheets/HDC1000_datasheet.pdf.

[18] Texas Instruments. *Intelligent System State Restoration After Power Failure With Compute Through Power Loss Utility*. Accessed: 25-10-2024. URL: https://www.ti.com/lit/ug/tidu885/tidu885.pdf.

[19] Texas Instruments. *MSP MCU FRAM Utilities version 03.10.00.10*. Accessed: 25-10-2024. URL: http://software-dl.ti.com/msp430/msp430_public_sw/mcu/msp430/FRAM_Utilities/latest/exports/FRAM-Utilities-UsersGuide.pdf.

[20] Texas Instruments. *MSP430 DriverLib MSP430F5xx 6xx User's Guide*. Accessed: 02-10-2024. URL: https://dr-download.ti.com/software-development/driver-or-library/MD-7R8r1GV3NB/2.91.13.01/MSP430F5xx_6xx_DriverLib_Users_Guide-2_91_13_01.pdf.

[21] Texas Instruments. *MSP430FR58xx, MSP430FR59xx, and MSP430FR6xx Family User's Guide (Rev. P)*. Accessed: 17-9-2024. URL: https://www.ti.com/lit/pdf/slau367.

[22] Texas Instruments. *MSP430FR5994 LaunchPad Development Kit (MSP-EXP430FR5994) User's Guide (Rev. C)*. [Download - accessed: 16-9-2024], pp. 1–30. URL: https://www.ti.com/lit/pdf/slau678.

[23] Texas Instruments. *MSP430FR599x, MSP430FR596x Mixed-Signal Microcontrollers datasheet (Rev. D)*. Accessed: 17-9-2024. URL: https://www.ti.com/lit/gpn/msp430fr5994.

[24] Texas Instruments. *TI-RTOS 2.20 User's Guide*. [Download - accessed: 8-5-2025], pp. 1–30. URL: https://www.ti.com/lit/pdf/spruhd4.

[25] Investopedia. *Make-or-Buy Decision Explained: How to Make Outsourcing Decisions*. [Online - accessed: 25-9-2024]. URL: https://www.investopedia.com/terms/m/make-or-buy-decision.asp.

[26] JSON. *ECMA-404 The JSON Data Interchange Standard*. [Online - accessed: 26-9-2024]. URL: https://www.json.org/json-en.html.

[27] James F. Kurose and Keith W. Ross. *Computer Networking*. 7th ed. Pearson, 2017, pp. 28–31.

[28] MIKROE. *HDC1000 click - User Manual*. Accessed: 17-9-2024. URL: https://download.mikroe.com/documents/add-on-boards/click/hdc1000/hdc1000-click-manual-v100.pdf.

[29] MIKROE. *mikroBUS^{TM} standard specifications*. Accessed: 25-9-2024. URL: https://download.mikroe.com/documents/standards/mikrobus/mikrobus-standard-specification-v200.pdf.

[30] MIKROE. *MIKROE*. [Online - accessed: 25-9-2024]. URL: https://www.mikroe.com/.

[31] THE THINGS NETWORK. *LoRaWAN Architecture*. Accessed: 17-5-2025. 2024. URL: https://www.thethingsnetwork.org/docs/lorawan/.

[32] Lea Perry. *IoT and Edge Computing for Architects*. 2nd ed. Packt Publishing, 2020, pp. 74–78, 82–86, 294–298.

[33] postman. *Postman*. [Online - accessed: 26-9-2024]. URL: https://www.postman.com/.

[34] PowerFilm. *PowerFilm*. [Online - accessed: 25-9-2024]. URL: https://www.powerfilmsolar.com/.

[35] PowerFilm. *Solar Development Kit with e-peas PMIC and CAP-XX Supercapacitors (DEV-EPEAS-CAPXX)*. [Online - accessed: 25-9-2024]. URL: https://www.powerfilmsolar.com/products/development-kits/solar-development-kit-with-e-peas-pmic-cap-xx-supercapacitors.

[36] PowerFilm. *Solar Development Kit with e-peas PMIC and CAP-XX Supercapacitors User Guide*. Accessed: 17-9-2024. URL: https://www.powerfilmsolar.com/hubfs/documents/instructions/development%20kits/AEM10941%20Evaluation%20Kit%20with%20PowerFilm%20Flexible%20PV%20Cell%20and%20Cap-XX%20Supercapacitor%20User%20Guide.pdf.

[37] pytest. *pytest: helps you write better programs*. [Online - accessed: 26-9-2024]. URL: https://docs.pytest.org/en/stable/.

[38] Jason Van Schooneveld. *Python and REST APIs: Interacting With Web Services*. Accessed: 17-9-2024. 2024. URL: https://realpython.com/api-integration-in-python/.

[39] seeedstudio. *Grove - Wio-E5*. Accessed: 17-9-2024. 2024. URL: https://wiki.seeedstudio.com/Grove_LoRa_E5_New_Version/.

[40] seeedstudio. *STM32WLE5JC*. Accessed: 25-9-2024. URL: https://files.seeedstudio.com/products/317990687/res/LoRa-E5%20module%20datasheet_V1.0.pdf.

[41] seeedstudio. *Wio-E5 AT Command Specification*. Accessed: 17-9-2024. URL: https://files.seeedstudio.com/products/317990687/res/LoRa-E5%20AT%20Command%20Specification_V1.0%20.pdf.

[42] seeedstudio. *Wio-E5 mini Dev Board*. Accessed: 17-9-2024. 2024. URL: https://wiki.seeedstudio.com/LoRa_E5_mini/.

[43] SQLAlchemy. *The Python SQL Toolkit and Object Relational Mapper*. [Online - accessed: 26-9-2024]. URL: https://www.sqlalchemy.org/.

[44] SQlite. *SQLite*. [Online - accessed: 23-9-2024]. URL: https://www.sqlite.org/.

[45] Bettles Stephen. *A Complete Guide to Data Loggers*. Accessed: 20-9-2024. 2023. URL: https://uk.rs-online.com/web/content/discovery/ideas-and-advice/data-loggers-guide.

[46] Hellegouarch Sylvain. *CherryPy Essentials - Rapid Python Web Application Development*. 1st ed. 2007, pp. 38, 46, 49.

[47] Tinytag. *Benefits of using data loggers*. [Online - accessed: 20-9-2024]. URL: https://www.geminidataloggers.com/support/knowledge-base/benefits-of-using-data-loggers.

[48] vscode. *vscode*. [Online - accessed: 25-9-2024]. URL: https://code.visualstudio.com/.

[49] the free encyclopedia Wikipedia. *Attribute*. [Online - accessed: 27-9-2024]. URL: https://en.wikipedia.org/wiki/Attribute_(computing).

[50] the free encyclopedia Wikipedia. *Edge computing*. [Online - accessed: 18-10-2024]. URL: https://en.wikipedia.org/wiki/Edge_computing.

[51] the free encyclopedia Wikipedia. *Electrical grid*. [Online - accessed: 24-9-2024]. URL: https://en.wikipedia.org/wiki/Electrical_grid.

[52] the free encyclopedia Wikipedia. *Entity*. [Online - accessed: 27-9-2024]. URL: https://en.wikipedia.org/wiki/Entity.

[53] the free encyclopedia Wikipedia. *Model–view–controller*. [Online - accessed: 26-9-2024]. URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller.

[54] the free encyclopedia Wikipedia. *Modularity*. [Online - accessed: 25-9-2024]. URL: https://en.wikipedia.org/wiki/Modularity.

[55] the free encyclopedia Wikipedia. *Rechargeable Battery*. [Online - accessed: 24-9-2024]. URL: https://en.wikipedia.org/wiki/Rechargeable_battery.

[56] the free encyclopedia Wikipedia. *Remote procedure call*. [Online - accessed: 26-9-2024]. URL: https://en.wikipedia.org/wiki/Remote_procedure_call.

[57] the free encyclopedia Wikipedia. *Unified process*. [Online - accessed: 24-9-2024]. URL: https://en.wikipedia.org/wiki/Unified_process.

[58] WordReference. *data*. [Online - accessed: 21-9-2024]. 2024. URL: https://www.wordreference.com/definition/data.