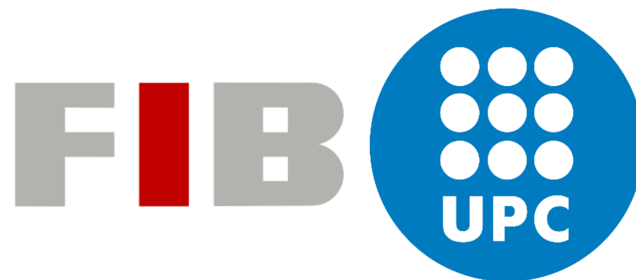
RT-DATA: A real-time data acquisition framework

Guillem Castro I Olivares

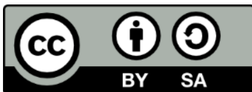
Director: Xavier Franch

Degree in Informatics Engineering

Specialization in Software Engineering



1st July 2019



Copyright © 2019, Guillem Castro Olivares

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, please visit http://creativecommons.org/licenses/by-sa/4.0/. You are free to copy, share, redistribute, adapt and remix this work for any purpose, even commercially, as long as you give appropriate credit and redistribute your contributions under the same license.

Abstract

In the last few years the popularity of Internet of Things solutions has grown to unsuspected levels. With the popularization of low-cost low-power embedded devices like Arduino or Raspberry Pi, the "Do It Yourself" community has been developing very interesting solutions for home automation and others.

Furthermore, we at Cosmic Research, have seen a great opportunity (as many other companies and research institutions have done) to use these same devices to be the brain of our data acquisition systems.

The purpose of this project is to build an easy-to-use software framework that can be used to implement simple data acquisition systems.

Resum

En els últims anys la popularitat de les solucions de la internet de les coses ha crescut fins a nivells mai vists. Amb la popularització dels dispositius encastats de baix cost i consum com l'Arduino o la Raspberry Pi, la comunitat "Do It Yourself" (fes-ho tú mateix) ha estat desenvolupant solucions molt interessants dins del món de la domòtica i d'altres.

A més, a Cosmic Research hem vist una gran oportunitat (igual que d'altres organitzacions i grups de recerca) d'utilitzar els mateixos dispositius com a cervell dels nostres sistemes d'adquisició de dades.

El propòsit d'aquest projecte és de construir un framework software fàcil d'utilitzar que pugui ser utilitzat per implementar sistemes d'adquisició de dades.

Resumen

En los últimos años la popularidad de las soluciones de la internet de las cosas ha crecido hasta niveles insospechados. Con la popularización de los dispositivos encastados de bajo coste y consumo, hemos visto como la comunidad “Do It Yourself” (hazlo tu mismo) ha estado desarrollando soluciones muy interesantes dentro del mundo de la automatización del hogar y otros.

Además, en Cosmic Research hemos visto la oportunidad (igual que han hecho otras organizaciones y grupos de investigación) de utilizar los mismos dispositivos como el cerebro de nuestros sistemas de adquisición de datos.

El propósito de este proyecto es el de construir un framework software fácil de utilizar que permita implementar sistemas de adquisición de datos.

Table of contents

Abstract 3

Resum 4

Resumen 5

1. Introduction and context 10

    1.1. Introduction 10

    1.2. Context 11

    1.3. State-of-the-art 14

    1.4 Stakeholders 16

2. Scope and methodology 17

    2.1. Scope 17

    2.2. Methodology 19

3. Planification and resources 22

    3.1. Task description 22

    3.2. Initial plan 23

    3.3. Final plan 26

    3.4. Resources 28

    3.5. Metrics 28

4. Budget and cost 30

    4.1. Project budget 30

    4.2. Project costs 33

5. Involved technologies 35

    5.1. Linux 35

    5.2. C++ 38

    5.3. SQLite 38

    5.4. I2C, SPI and UART 39

5.5. JSON 41
6. Requirements analysis 43
6.1. User stories 43
6.2. Non-functional requirements 56
7. Design and architecture 59
7.1. Logical architecture 59
7.2. Layer’s design 60
8. Implementation 62
8.1. Events management 64
8.2. Sensors management 68
8.3. Control 74
8.4. Serialization 77
8.5. IO 80
8.6. Timestamping 85
8.7. Concurrency 87
8.8. Configuration 90
9. Verification and validation 95
10. Laws, regulations and licenses 98
10.1. Laws and regulations 98
10.2. Licenses 99
11. Sustainability and social commitment 100
11.1. Environmental sustainability 100
11.2. Economical sustainability 101
11.3. Social sustainability 101
11.4. Sustainability matrix 102
12. Conclusions 104

12.1. Future work 104

13. Bibliography 106

Table of figures

Figure 1 Simplified data acquisition system architecture 12

Figure 2 Distributed control system architecture 13

Figure 3 Initial scheduling 25

Figure 4 Final scheduling 27

Figure 5 Burndown chart 29

Figure 6 Velocity chart 29

Figure 7 Kernel space and user space 36

Figure 8 System calls with device files 37

Figure 9 Logical architecture in layers 60

Figure 10 Package diagram 63

Figure 11 Data class 64

Figure 12 Broker and listeners 65

Figure 13 Sequence diagram for the dispatch method 66

Figure 14 Sequence diagram for the subscribe method 66

Figure 15 Sensor and SensorsManager 69

Figure 16 Sequence diagram for the fetch method 70

Figure 17 Sequence diagram for the read method and example 71

Figure 18 State and StateManager 75

Figure 19 Sequence diagram of the handle method 76

Figure 20 Serializable and SerializedObject 77

Figure 21 Writer and writers 80

Figure 22 Data buses 83

Figure 23 The 'concurrent' package 87

Figure 24 SchedulingPolicy enumeration 88

Figure 25 Configuration package 91

Figure 26 Configuration tree 91

Figure 27 Sequence diagram of the at method from class JSONConfiguration............ 93
Figure 28 Failed and successful builds.. 95

1. Introduction and context

1.1. Introduction

In the last few years the popularity of IoT (Internet of Things) and DIY (Do-It-Yourself) solutions using low cost and low power platforms has grown to unsuspected levels. According to a report by Transparency Market Research [1], the expected compound annual growth rate for the IoT market is of 20.55% between the years 2016 and 2024.

More so, some scientific and academic institutions are starting to use the same embedded platforms IoT solutions use for the implementation of their scientific experiments. For example, EPICS (Experimental Physics and Industrial Control System), a library to create real-time distributed systems used by scientific institutions from around the world, has been reported to be used with Raspberry Pi, a popular low cost mini-computer [2].

In recent years new technologies and platforms have appeared to reduce the cost and time of developing these systems. In the software segment most systems are still being developed using one-time solutions, although some components could be reused in more than one system.

This project did not begin because of an innovative idea from its author, but from a real necessity. For the last 3 years I've been part of Cosmic Research, a project that aims to launch a rocket into space. One of the most critical part of a rocket is its telemetry and control system. I am currently in charge of implementing this system for Bondar, our next rocket that will reach an apogee of 15 km approximately.

This project begun because we needed a reusable software framework that allowed us to implement the telemetry system for Bondar and its successors, and to implement a data acquisition system for our motor test bench. The only difference between these two systems is the collected data, so both systems can share the same implementation for the major part of its software components.

These two systems are not very different to what we are used to see in the IoT industry. They all gather data from a set of sensors, dispatch it to a remote location, and sometimes they use it to activate some kind of actuators.

1.2. Context

Before explaining the project in detail, it is important to introduce some concepts that will help to understand it, and that will be part of the discussion about the project. I will also introduce the major stakeholders and the role they will play.

1.2.1. Data acquisition

Data acquisition is the process of sampling signals that represent measures from the real world and then converting them into values that can be processed by a computer. The typical components of a data acquisition system are [3],

- A real-world source of data. Normally a physical phenomenon or property, like temperature, speed or even position.
- A sensor that converts the physical property to an electrical signal. It can be either digital or analogic signals. Digital signals transfer the data encoded in bits while analogic signals normally transfer the data encoded with the voltage level.
- An analogic-to-digital signal converter (ADC). As computers are not able to understand analogic signals, ADCs are used to convert them to a digital signal that the computer will be able to understand.
- A computer that will perform the sampling of the sensor and process the acquired values.

In Figure 2 you can see the architecture of a simplified data acquisition system with two sensors, one analogic and one digital, an anlogic-to-digital converter and a computer.

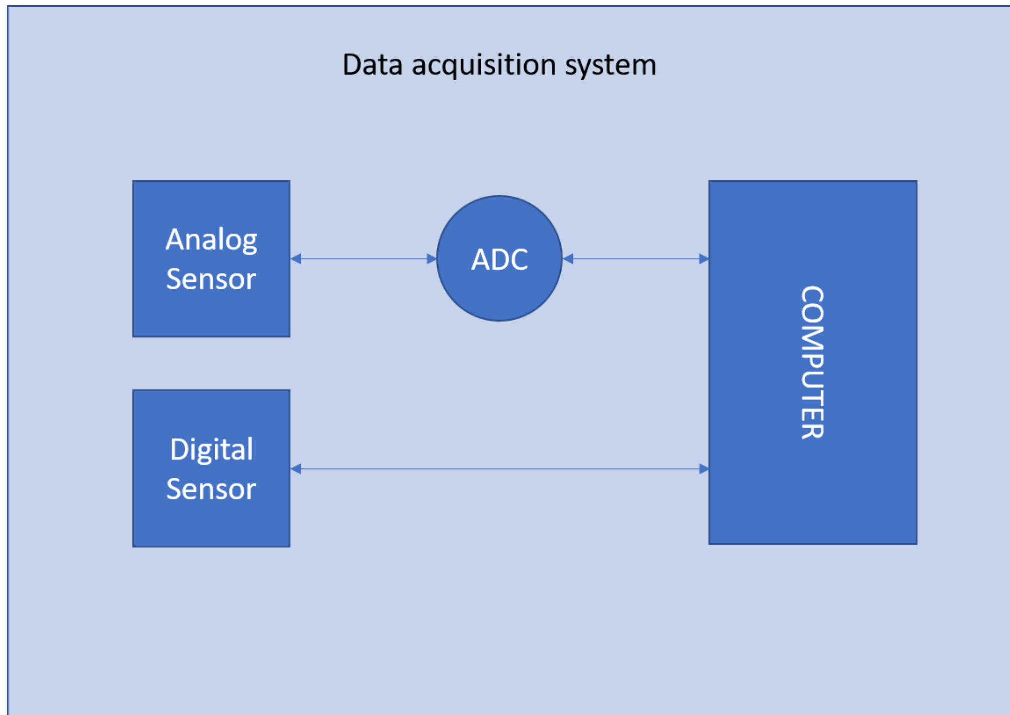


Figure 1 Simplified data acquisition system architecture

The hardware used to perform the data acquisition will vary between systems, although PLCs (Programmable Logic Controllers) and microcontrollers are almost always used as computers.

In the software part, there is no *standard* library or platform, but some of the most popular are EPICS and ROS (Robot Operating System) that I will introduce later. It's not uncommon to implement data acquisition systems using general purpose languages like C++ or even Java, without using any kind of existing framework or library.

1.2.2. Control Systems

Control systems are systems in charge of managing or controlling the behaviour of other devices. Typically control systems use data acquisition systems to obtain input data that will be used either directly by the system or by an operator to control the behaviour of the managed devices.

Control systems are everywhere in our lives and we interact with them daily. For example, home thermostats are a simple example of it. The thermostat acquires temperature data from a temperature sensor and uses it to control the heating system.

This kind of systems usually use a low-cost low-power hardware and software specifically designed to perform such task, as its operation is not critical, and delays are acceptable.

But normally when we talk about control systems, we are usually talking about industrial control systems, such as the ones used at factories or at power plants. These systems usually use PLCs or other high-end industrial computers. Software-wise, DCSs are usually implemented using existing libraries and frameworks like EPICS.

Large control systems, as in big factories or large scientific facilities, are implemented in a distributed manner [4]. Distributed control systems (DCS) are structured in different levels, as can be seen in Figure 2.

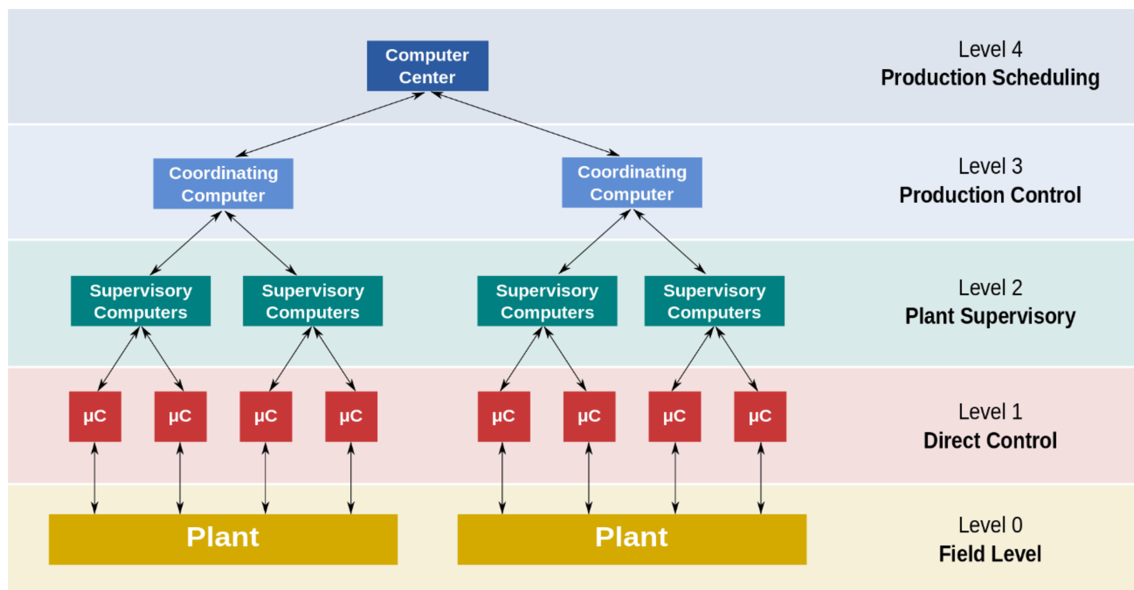


Figure 2 Distributed control system architecture[1]

Levels 0 (devices and sensors) and 1 (PLCs and industrial computers) are in fact control systems by itself, called direct control systems. Higher levels include operator supervision and coordination and scheduling. This can range from human-machine interfaces to algorithms that perform automated actions.

---

[1] Original work from Daniele Pugliesi, distributed under Creative Commons 3.0. Originally uploaded to Wikimedia Commons.
https://commons.wikimedia.org/wiki/File:Functional_levels_of_a_Distributed_Control_System.svg

SCADAs (Supervisory Control and Data Acquisition) are very similar to DCS and is the most common type of industrial control system. They are the combination of data acquisition and control systems, a communication infrastructure and a human-machine interface (HMI). The most common human-machine interfaces are graphical user interfaces (GUI).

The HMIs are used by the operators of the industrial facility to supervise all the systems in a plant, from water tanks levels to the current status of a steam valve. SCADAs allow direct control of all the devices by the operator.

1.3. State-of-the-art

In the field of data acquisition and control systems it shall come to no surprise that there is an infinite amount of different solutions. All of them with different levels of complexity and even targeted for different applications.

I've selected two open-source frameworks that can be used to implement data acquisition and control systems, EPICS and ROS.

1.3.1. EPICS (Experimental Physics and Industrial Control System)

EPICS is an open-source set of tools and libraries that can be used to implement distributed soft real-time control systems [5]. It is mainly targeted to scientific institutions, such as the Argonne National Laboratory, or ITER.

EPICS has three main components,

- **OPI (Operator Interface)**. A workstation that presents the data managed by EPICS to the operator.
- **IOC (Input Output Controller)**. A computer that will perform the data acquisition and exposes the data as records. Each record type has a fixed set of fields.
- **A local network (LAN)**. This network is used to allow the communication between the IOCs and OPIs. This LAN supports an infinite number of OPIs and IOCs, as long as the network is not saturated.

Other hosts connected to the network can also access the data provided by the IOCs, and even modify it using a mechanism called Channel Access (CA). It uses a server client

model where IOCs act as the server. OPIs are, in fact, Channel Access clients subscribed to the records provided by CA servers.

EPICS follows an event-driven paradigm which means that CA clients instead of polling IOCs, they can request subscribe to record types. When new records are produced or when a change is produces, all subscribed CA clients are notified.

The usage of EPICS in Cosmic Research projects was quickly discarded, as it is targeted at big distributed control systems, such as the ones at CERN or ITER.

1.3.2. ROS (Robot Operating System)

The Robot Operating System is a software framework for building robots [6]. Despite its name, ROS is not an operative system. ROS is targeted to much smaller systems than EPICS and is targeted to embedded platforms, although it also supports building distributed control systems. Each ROS instance is a node, and one host can contain more than one node.

ROS also follows a publish-subscribe paradigm, somewhat similar to the event-driven paradigm from EPICS. Data is distributed in messages, and each message is published to a topic. When a ROS node publishes a message to a topic, all subscribed nodes are notified. This means that it is not limited to robots and can also be used to implement data acquisition systems.

Messages are specified in its own file using C-like syntax. At compile-time an external tool converts the message specification to the target language. ROS supports both C++ and Python.

For data storage and logging ROS provides bags. ROS automatically stores messages from selected topics in raw files. They can be later analysed, processed and replayed. ROS also provides an interface to store messages in standard databases, like SQLite or PostgreSQL.

ROS also provides a several tools and libraries to implement robot features, like the Robot Geometry Library, the Robot Description Library, and a diagnostics library.

ROS was mainly discarded because it does not provide a standard interface to define sensor nor data acquisition implementations, which makes it harder to reuse them between systems. This is one of the main requirements that we had, to be able to reuse as much code as possible between systems.

Furthermore, not providing a standard interface for sensors makes harder for non-proficient users to implement data acquisition systems, even simple ones.

1.4 Stakeholders

As happens with all projects there is a set of people that will be affected by it. These are the most relevant ones,

- **The developer**, myself. I will be the only developer for this project which means that I will do all the planning, implementation and testing.
- The project director, **Xavier Franch**. He is a full professor from the ESSI department of the UPC. He is also part of the GESSI research group. He will be my guidance through all the project giving me advice.
- **Cosmic Research** will be the primary user of the framework. The requirements provided by its engineers will be fundamental on the design of the framework. They will use it to implement a telemetry system for a rocket, and the electronics for a motor test bench.
- **Other users**. Other research groups, companies or even technology enthusiast might want to use this framework to implement their data acquisition systems. This framework can be used in a variety of different environments from rockets to drones, or even to implement some IoT solutions.
- **Beneficiaries**. People that will benefit directly or indirectly with the systems built with the framework, or from the research made with those systems.

2. Scope and methodology

After a brief introduction to the subject and to the context of this project, it is the moment to explain in more detail the scope and the methodology of the project.

2.1. Scope

At the start of the project, after talking with the stakeholders, I made a list of objectives for the project. During the execution no major changes have been made.

Let's take a look at the original scope of the project and the changes it has suffered.

2.1.1. Original scope

As explained in the introduction, the objective of this project is to implement a reusable software framework that can be used to implement data acquisition systems for low-cost low-power embedded environments. Also, it will be possible to use it to implement simple control systems, or to implement the levels 0 and 1 of a distributed control system or SCADA.

It is required that it has real-time capabilities, with very low delays between the data acquisition and the decision process execution. It must also provide mechanisms to implement such decision processes.

The framework will be responsible of the management of the system's sensors, the data acquired by them, the dispatch of the data to any interested actor (including remote hosts or databases) and the execution of different decision and control processes.

It would be desirable that it provides interfaces to interface with common-used sensors, databases or file systems to store the acquired data, and some communication protocols to send it to remote or local hosts.

The framework will be implemented in C++ and must provide full C++ interfaces without being necessary to use external tools or languages (except for the compilation process). It shall be noted that although all platforms have a working C/C++ compiler, not all might be compatible with a specific library or tool. Furthermore, it must provide a configuration mechanism so that it is possible to modify some configurations

parameters (such as the sampling interval of a sensor) without having to recompile the whole system.

As not all the expected users of this framework are proficient at C++ or even at programming, it shall provide simple interfaces and help libraries that make easier the implementation of data acquisition and control systems. Obviously, some experience is expected.

These objectives of the project can be summarized to the following list,

- Building a reusable software framework to implement data acquisition and (simple) control systems
- With real-time capabilities
- Can be executed on low-cost low-power embedded platforms
- Implemented in C++ with minimal external dependencies
- Provides interfaces for common-used sensors, databases, …
- Easy to use, even for the unexperienced users.

2.1.2. Final scope

As previously said, no major changes have been introduced to the original scope of the project. More than an actual change, there has been a shift in the priorities of the project.

At the start, I put the usability or user-friendliness at the bottom of the objectives list. During the development of the project I came to the realisation that nowadays with the computation power that even a small device like a Raspberry Pi has, the real objective or challenge was to develop an easy to use framework.

It's actually nothing new, the Arduino project made the IoT and embedded world more accessible. In the past, the embedded software was written with performance as the main priority because the computers and microcontrollers didn't have the resources they have now. In the present, that is not true anymore. A simple Raspberry Pi has more than 1GB of memory and can perform millions of operations per second. As it has happened in other environments, it is the time to make embedded code more usable and more maintainable.

One of my top priorities has been to develop rt-data with interfaces that are as easy to use as those we can find in Arduino. And to provide the tools to develop code that can be reused. Even if that means that we won't achieve the best performance.

2.2. Methodology

Having talked about the project context, objectives and state-of-the-art, it is time to start talking about how the project will be executed and validated.

2.2.1. Working methodology

At the start of the project, given the short time span of the project (about four months) and the more than possible changes in requirements after acceptance tests, I decided to work with an agile methodology, Scrum. As Scrum is very team oriented and I am a single person, I have used an adapted version of Scrum.

During the Inception (the first phase of the project), I held a set of meetings with the members of Cosmic Research to gather requirements. To that, I added some requirements I considered interesting from the products mentioned in the state-of-the-art, and some innovative features. All this form the product backlog for rt-data.

Instead of using a very formal specification for requirements, I chose to work with user stories. I took this decision because of two reasons; it puts more emphasis on what and why the user wants, and it allows a greater agility and is more tolerant to changes during the project.

Each user story has assigned a number of points, from 1 to 8 using the Fibonacci sequence, that represent the difficulty or the time it might take to implement it, the expected effort. This will be very useful when planning each sprint. User stories will be grouped into epics. Epics typically represent big chunks of work, like features.

The project has been executed in one-week sprints. At the start of each sprint, I picked the most relevant user stories given their importance for the stakeholders, time constraints and inter-dependencies. In a sprint user stories are planned, implemented and tested. The set of user stories to be worked on in a sprint is the sprint backlog.

The product backlog was categorized into 5 epics,

1. **Sensors**: everything related to data acquisition from sensors.
2. **Events management**: everything related to the asynchronous passing of data between components.
3. **Support libraries**: support libraries related to data timestamping, thread management and timers.
4. **Control**: everything needed for control of the I/O ports (UART, I2C, …).
5. **Configuration**: everything related to the configuration of sensors, I/O ports, …

The product backlog has a total of 37 user stories with a total of 165 points.

No more interviews have been held with any of the stakeholders, as there has not been any major deviation from the initial scope, and no requests have been received from the stakeholders.

2.2.2. Development tools

At the start of the project I proposed to work with the following tools,

- **Git** as the version control system (VCS), using a repository hosted at **Github**.
- **Visual Studio Code** as the integrated development environment (IDE).
- **Taiga.io** as the project planning and monitoring tool. Here I will define the product backlog and each sprint backlog.
- **Vagrant** and **VirtualBox** to perform local builds.
- **Jenkins** for test and build automation.
- **CppCheck** for static code analysis.
- **Doxygen** for generating the user documentation.

Since the project start until now, Vagrant and VirtualBox have not been used. I've been able to perform local builds on my Windows machine using the **Windows Subsystem for Linux**.

2.2.3. Validation method and software quality assurance

Validation is an important part of any software project, but it is critical for some of the applications of this framework. To guarantee the quality of the code I have been performing three types of quality assurance processes, static code analysis, unit testing and acceptance testing. The first two processes have been successfully automated using

Jenkins. After each change, all unit tests are executed to ensure that it does not introduce new bugs in existing code.

The objective of the acceptance tests is to validate that the software complies with the requirements negotiated with the stakeholders. Two acceptance tests have been successfully performed at the end of the 5th and 8th sprints.

Testing has allowed to identify and solve numerous bugs that in a real-world scenario would have caused a system crash. One of them was actually caused by a typo on the return type of a method. Instead of returning a `std::string` it was returning a reference to a `std::string` local to the method, causing a segmentation fault later in the execution. I was expecting that the static analysis of the code detected this kind of errors, and without acceptance testing I would have delivered a software with critical issues.

3. Planification and resources

An important part to make a project successful is the planification. In this section I will present the project plan that has been executed. This project begun the 18th of February 2019 and ended the 1st of July 2019.

3.1. Task description

In this section I will describe the main tasks of the project.

3.1.1 Inception

The Inception is the first phase of the project, done before starting the implementation of the framework. It included the first deliverables of GEP, the requirements analysis, the initial planification of the project and the setup of the development and testing environments. Part of this task has been investigating the subject and possible competing solutions, and interviews with some of the most relevant stakeholders.

After the inception everything was ready to begin the development of the rt-data framework.

3.1.2 Software design and implementation

As in all iterative and/or agile methodologies, the software has been designed and implemented in an iterative way. This means that in each sprint a chunk of features (in form of user stories) has been designed and implemented, along with the possible leftovers from previous sprints. All implemented features are subject to changes in future sprints, based on the stakeholder's inputs.

In theory user stories have no dependencies, they are independent. In practice, though, all features have some dependencies between them, as all are part of the same system. The planification was made considering the possible dependencies that might arose during development.

3.1.3 Software validation and testing

During each sprint part of the work has been to define unit tests to verify that the implemented features didn't introduce new bugs in the existing code. At the end of some sprints; 5 and 8; I have designed and implemented acceptance tests to validate that the implemented features met the stakeholders' needs.

Unit tests have been executed during all the project cycle, after each change was introduced thanks to the automation of the execution in Jenkins.

3.1.4 Documentation

A major task during the project has been its documentation. It includes,

- The user documentation generated with Doxygen.
- UML diagrams to help understand the design that will be presented later in this document.
- The present bachelor's thesis document
- The GEP presentation and the final thesis presentation

This work will be done in parallel to software design, implementation and planning.

3.2. Initial plan

During the Inception phase of the project, I made a tentative plan. First I will present it, and then we will see the changes it has suffered.

3.2.1. Timetable

For each week I estimated a workload of around 25 hours. The project duration was estimated to 19 weeks in total. The estimation of hours per task was the following,

Table 1 Initial timetable

Task	Estimated duration (hours)
Inception	75
Software design and implementation	310
Software validation and testing	30
Documentation	60
Grand total	475

3.2.2. Scheduling and Gantt chart

The Gant chart that follows was the tentative schedule for each sprint in the project, and its user stories. The definitive scheduling for each sprint was done at its start, taking into consideration the stakeholders' priorities, the result of the acceptance tests, and

possible delays in the execution of the project. Please note that the Scrum methodology allows to add more user stories to the backlog after the inception has ended.

The chart shows some of the dependencies between the user stories, but more could have been found during project execution. Although user stories should be independent in theory, in practice it's almost impossible. Most features re-use parts of other features, making them dependent.

In total it was planned to have 12 one-week sprints + the inception that would last three weeks. Between the end of the last sprint and the final deadline of the project, there is a gap of four weeks. These weeks have been used to prepare the final thesis document, the presentation, and, if needed, could have used them for additional sprints. For example, in case of delays or new user stories requested by the stakeholders.

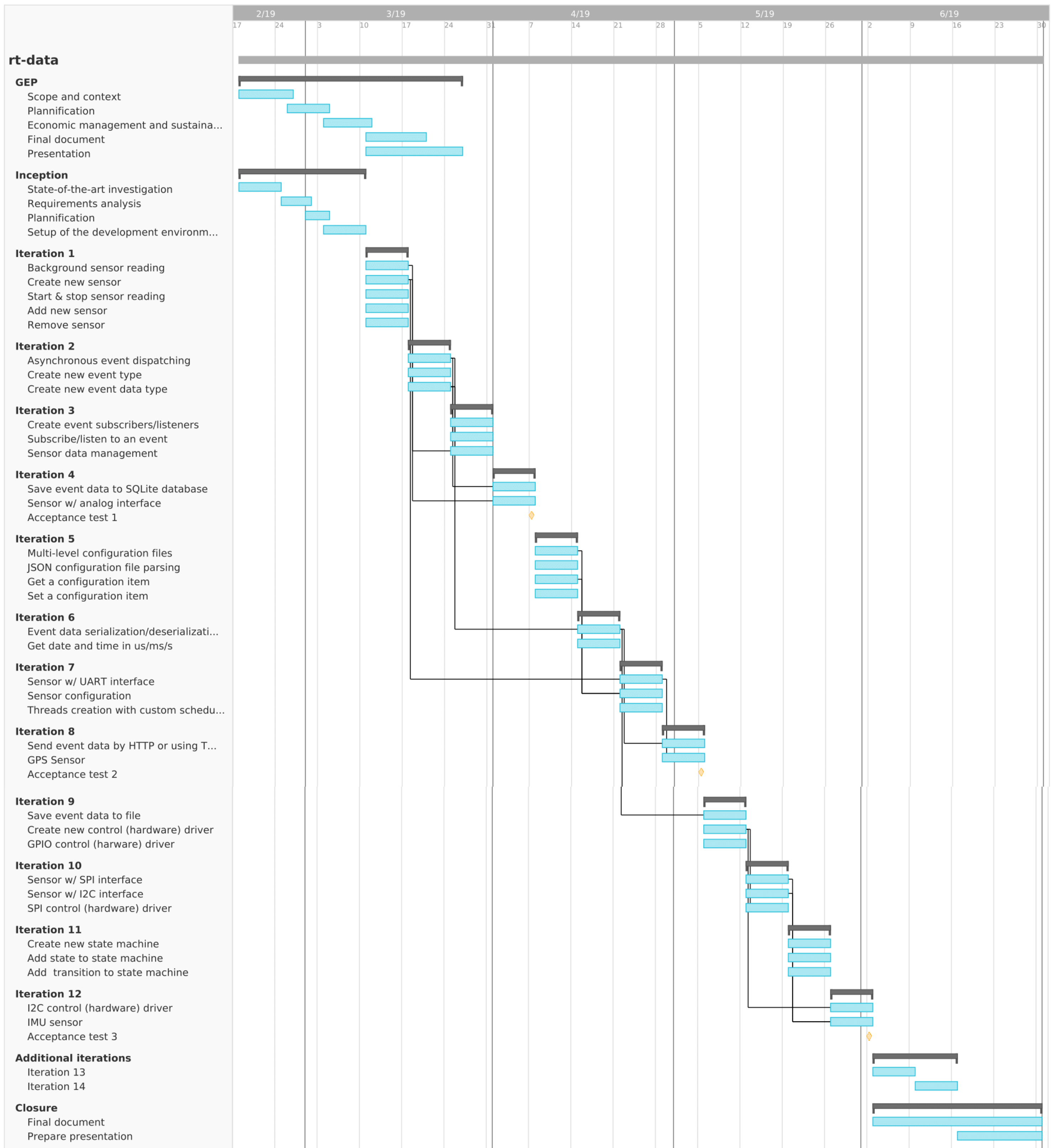


Figure 3 Initial scheduling

3.3. Final plan

In this section I will introduce the changes and deviations, if any, from the initial planification.

3.3.1. Timetable

As originally estimated, the weekly workload has been around 25 hours on average. The workload on the first and last weeks was been higher than 25 hours, as I had to do the GEP in parallel, and that the development environment was not yet mature, and at the final weeks I had to prepare the final thesis document in parallel. On the other end, on other weeks the workload was significantly lower. For example, during the holy week, when I decided to take a little break from work.

The total hours dedicated to the project follow,

Table 2 Final timetable

Task	Dedication (hours)
Inception	75
Software design and implementation	310
Software validation and testing	30
Documentation	60
Grand total	475

3.3.2. Scheduling and Gantt chart

The Gant chart that follows (Figure 4) is the Gantt chart of the progress so far, and the new planification for the next sprints. No major changes have occurred, only changes in priority and dependencies, and one delay. For example, at the start of the project I realized that the data for the events that are managed by the system, have to be timestamped. For that reason I rescheduled the story “Get data and time in us/ms/s” from sprint 6 to sprint 2.

During sprint 4, I encountered a major bug that took more than expected to solve. Therefore, I had to delay the story “Sensor analog driver” and the first acceptance test to sprint 5. All the stories planned for sprint 5 were moved to sprint 6.

During all sprints until now, no extra hours have been needed.

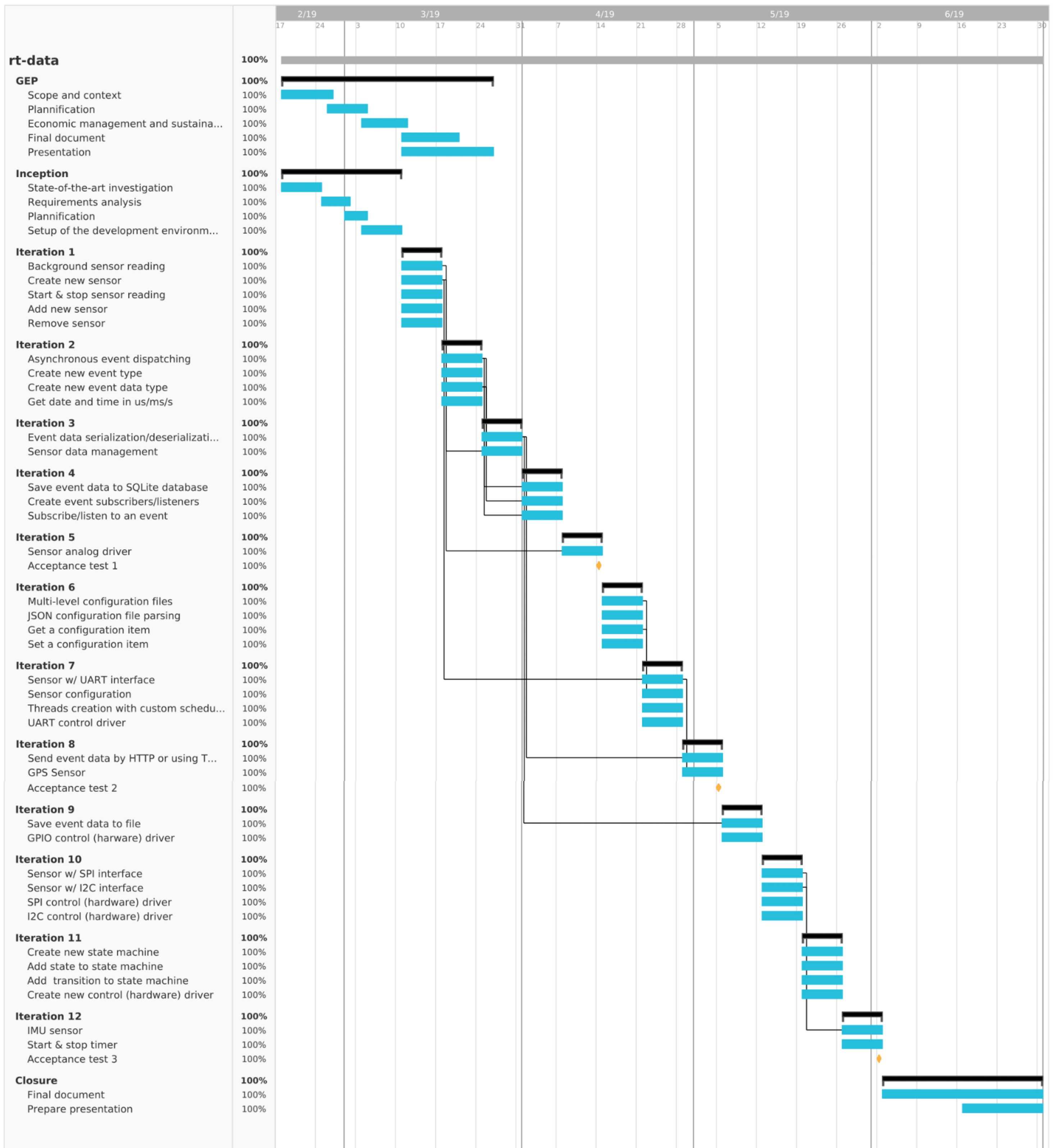


Figure 4 Final scheduling

3.4. Resources

In this section I will explain all the necessary resources that have been needed to accomplish the planification.

3.4.1 Human resources

For this project, just one person has been needed, me. The average workload has been of 25 hours per week for a total of 475 hours. I have been in charge of all the tasks of the project, like the planification, design, implementation, testing, etc.

3.4.2 Material resources

- A workplace. The Cosmic Research offices located at Terrassa.
- A laptop. I will use my own, a Dell XPS 15. I used it for almost all tasks, including design, implementation, testing, documentation, …
- A server for Jenkins hosted at Amazon AWS.
- Two target hosts. A Raspberry Pi 3B+ board and a RoadRunner board by AcmeSystems.
- A u-blox MAX-M8Q GPS module.
- A LSM9DS1 IMU by STMicroelectronics.

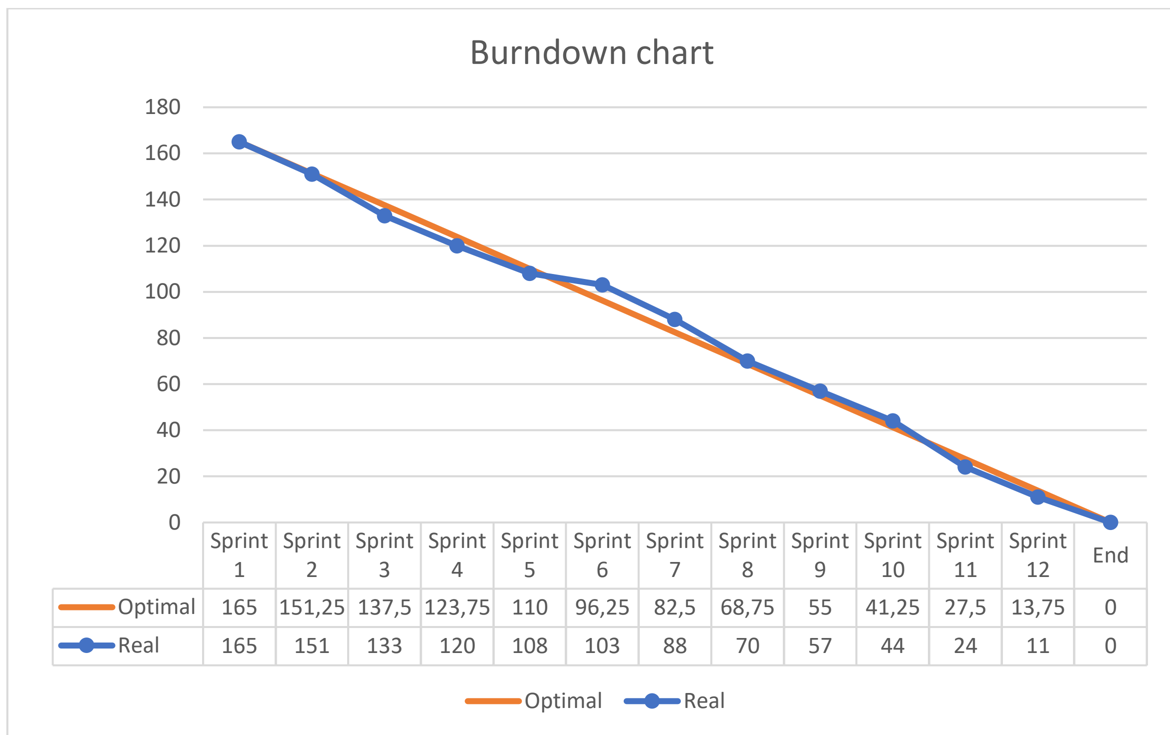
3.4.3 Software resources

- Jenkins for build and test automation.
- Git and Github for version control.
- Taiga.io and TeamGantt for project planning.
- Visual Studio Code as the integrated development environment.
- Debian GNU/Linux as the target host's operating systems.
- Windows 10 as the operating system of the development laptop.
- CppCheck for the static code analysis.
- Microsoft Office suite for the thesis documentation and presentations.
- Make and CMake for the build automation.

3.5. Metrics

I have been monitoring the project execution using Taiga.io. I have completed the 165 points of the project in 12 sprints. That means that in average, I have closed 13,75 points

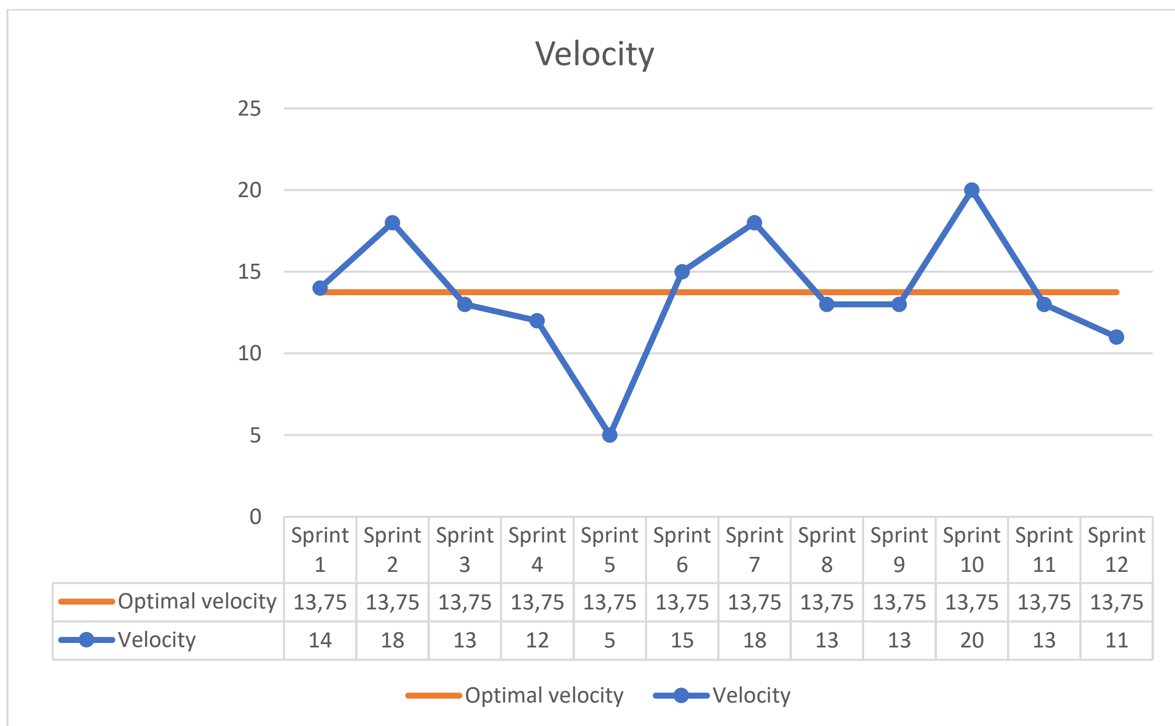
each sprint. A burndown chart (pending points at the start of the sprint) can be seen in Figure 5.



	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7	Sprint 8	Sprint 9	Sprint 10	Sprint 11	Sprint 12	End
Optimal	165	151,25	137,5	123,75	110	96,25	82,5	68,75	55	41,25	27,5	13,75	0
Real	165	151	133	120	108	103	88	70	57	44	24	11	0

Figure 5 Burndown chart

A velocity chart (points closed per sprint), can be seen in Figure 6.



	Sprint 1	Sprint 2	Sprint 3	Sprint 4	Sprint 5	Sprint 6	Sprint 7	Sprint 8	Sprint 9	Sprint 10	Sprint 11	Sprint 12
Optimal velocity	13,75	13,75	13,75	13,75	13,75	13,75	13,75	13,75	13,75	13,75	13,75	13,75
Velocity	14	18	13	12	5	15	18	13	13	20	13	11

Figure 6 Velocity chart

4. Budget and cost

4.1. Project budget

At the inception I made a budget, taking into consideration the resources exposed in the previous section. In this section, I have analysed the cost of such resources (human, material and software), and its amortizations, as well as other indirect costs.

4.1.1. Human resources

Table 3 has the total budget for human resources per role, its cost per hour and the total cost for the project. And Table 4 provides the same budget but specified per task, including the estimated hours that each role will dedicate to each task.

The estimated salaries have been extracted from Glassdoor [7]. For each role I have picked the average annual salary in the Barcelona area. To calculate the price per hour, I have assumed 14 pays per year and that each month has 20 working days in average.

Table 3 Human resources budget per role

ROLE	HOURS	PRICE PER HOUR	TOTAL
PROJECT MANAGER	105	20€	2100€
SOFTWARE ENGINEER	340	17€	5780€
TESTER	30	10€	300€
GRAND TOTAL	475		8180€

The tasks of the following table (except Inception), correspond to the tasks that are executed in each sprint of the project.

Table 4 Human resources budget per task and role

TASK	HOURS	HOURS PER ROLE			COST
		PROJECT MANAGER	SOFTWARE ENG.	TESTER	
INCEPTION	75	65	10	0	1470€
DESIGN & IMPLEMENTATION	310	10	300	0	5300€
DOCUMENTATION	60	30	30	0	1110€

TASK	HOURS	HOURS PER ROLE			COST
		PROJECT MANAGER	SOFTWARE ENG.	TESTER	
VALIDATION	30	0	0	30	300€
GRAND TOTAL	475	105	340	30	8180€

4.1.2. Material resources

In Table 5 it is listed the budget for all the needed material resources. The amortization has been made taking into consideration that the length of the project is approximately 4 months.

Table 5 Material resources budget

RESOURCE	PRICE	USEFUL LIFE	AMORTIZATION
DELL XPS 15	1600€	4 years	133,33€
SERVER[2]	0€	-	0€
RASPBERRY PI 3 B+	63,93€	2 years	10,66€
ROADRUNNER	66€	2 years	11€
UBLOX MAX-M8Q	47,24€	1 year	15,75€
LSM9DS1	29€	1 year	9,67€
GRAND TOTAL	1806,17€		180,41€

4.1.3. Software resources

Table 6 contains the budget for all the needed software resources. It has been taken into consideration that although some software products that will be used are not free, Cosmic Research has sponsorship agreements that provide free licenses for all its members.

Table 6 Software resources budget

RESOURCE	PRICE	USEFUL LIFE	AMORTIZATION
JENKINS	0€	-	0€

[2] The server will be rented to Amazon AWS and is eligible for the free tier

RESOURCE	PRICE	USEFUL LIFE	AMORTIZATION
GITHUB[3]	0€	-	0€
TAIGA.IO	0€	-	0€
TEAMGANTT[4]	0€	-	0€
VISUAL STUDIO CODE	0€	-	0€
VAGRANT	0€	-	0€
VIRTUALBOX	0€	-	0€
DEBIAN GNU/LINUX	0€	-	0€
WINDOWS 10	145€	3 years	16,11€
CPPCHECK	0€	-	0€
MICROSOFT OFFICE	149€	3 years	16,56€
CMAKE	0€	-	0€
MAKE	0€	-	0€
GRAND TOTAL	294€		32,67€

4.1.4. Other expenses

As previously said, I will work on the project on Cosmic Research's office in Terrassa. The costs in Table 7 are the real costs for our office.

The monthly rent of the offices is 395€ + IVA that comes into a cost of 477,95€. The monthly electricity bill is in a yearly average, 70€/month. The price of the kWh has been taken from Red Eléctrica de España [8]. In average the price of the kWh in the month of March has been of 0,11€/kWh.

For the transportation, I have considered that I'll need at least one T-10 each week to go to the office. From where I live to Terrassa, a 2-zone T-10 is needed that costs 20€.

Table 7 Other expenses

RESOURCE	PRICE	UNITS	COST
OFFICE	477,95€/month	4 months	1911,80€

[3] Github provides free licenses for all Cosmic Research's members due to a sponsorship agreement.
[4] TeamGantt provides free licenses for all Cosmic Research's members due to a sponsorship agreement.

RESOURCE	PRICE	UNITS	COST
INTERNET CONNECTION	45€/month	4 months	180€
ELECTRICITY	0,11€/kWh	20 kWh/day * 120 days	264€
TRANSPORTATION	20€/week	19 weeks	380€
OFFICE SUPPLIES	50€	N/A	50€
GRAND TOTAL			2885,80€

4.1.5. Total budget

Finally, Table 8 contains the total budget for the project with a total of 12.406,77 €. The biggest parts of the budget are the human resources and the office costs that shall come to no surprise as we are using low-cost hardware and free software.

I have included a contingency of a 10% of the total cost, to cover any unexpected expenses or a possible delay in the execution of the tasks.

Table 8 Total project budget

CONCEPT	COST
HUMAN RESOURCES	8.180€
MATERIAL RESOURCES	180,41€
SOFTWARE RESOURCES	32,67€
OTHER EXPENSES	2.885,80€
TOTAL	11.278,88€
CONTINGENCY (10%)	1.127,89€
GRAND TOTAL	12.406,77€

4.2. Project costs

The deviations in the project execution have not impacted the cost of the project estimated at the budget. The 10% contingency was not needed, so the total cost of this project has been 11.278,80€. Table 9 holds the total cost of the project.

Table 9 Total cost of the project

CONCEPT	COST
HUMAN RESOURCES	8.180€

CONCEPT	COST
MATERIAL RESOURCES	180,41€
SOFTWARE RESOURCES	32,67€
OTHER EXPENSES	2.885,80€
GRAND TOTAL	11.278,88€

5. Involved technologies

In this section I will present and discuss all the technologies that are used by rt-data.

5.1. Linux

Linux is the kernel of the GNU/Linux operating system [9] (sometimes called just Linux), and is the target platform of rt-data. The framework delegates all the I/O (input output) operations to the Linux kernel using its standard system calls.

I won’t explain how all the kernel works, but I consider necessary to explain how we can interface external devices from a Linux application.

5.1.1. User-space, kernel-space and drivers

In Linux, actually in almost all OS, there is a separation of concerns between what the user can do and what the kernel can do. The kernel is the responsible of managing all the system’s resources and devices, therefore it has full and direct access to them, without any intermediary. This has some important implications that we must know, for example, if we modify a position of memory that we are not supposed to modify, instead of generating a segmentation fault, it most certainly will crash the computer.

All the kernel code is stored in a special region of the memory, called **kernel space** [10]. All the code inside this region is executed in privileged mode or **kernel mode**. In kernel mode, any instruction supported by the CPU can be executed.

Part of the Linux kernel is a special kind of software called **drivers**. A driver tells the kernel how it should communicate with a hardware device. Drivers can also make use of other drivers. For example, the driver of your printer is probably using the USB driver for the communication.

It would be very naïve to think that the kernel contains drivers for every possible device in the market. When working in embedded solutions, it is very common to find that the device you want to use doesn’t have a Linux driver. If you can find one, installing a new driver is very easy, as it can be done by just executing one command.

The rest of the memory, the **user space** (also called userland), is dedicated to the user applications. All the code located in this area is executed in **user mode**. Typically, the

instructions that can be executed in this mode is limited but depends on the architecture of the CPU.

To communicate between user space applications and hardware devices, you must use system calls. Some of the most common to perform I/O operations are `read()` and `write()`. When the kernel receives a system call, it then dispatches the execution to the corresponding driver. Drivers cannot make use of system calls and have to rely on kernel primitive methods.

In the Figure 7, you can see how the different actors involved in an I/O operation interact.

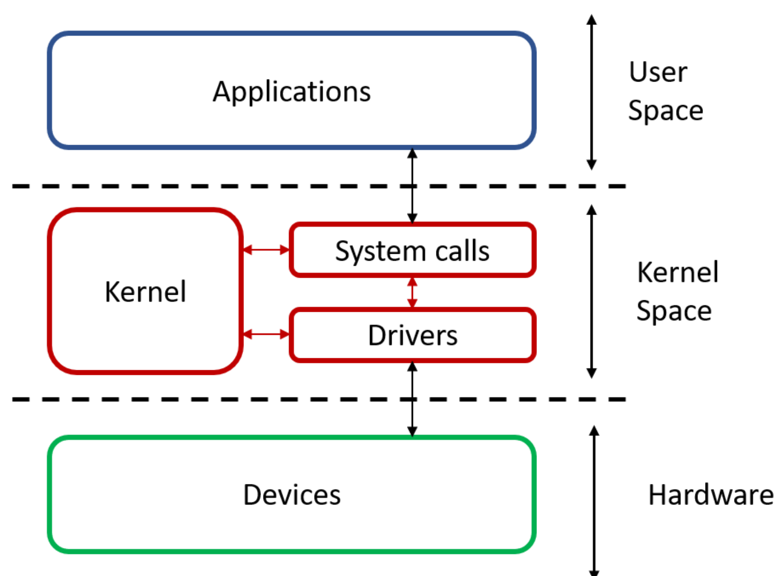


Figure 7 Kernel space and user space

5.1.2. /dev and udev

One of the main differences between Linux and other operating systems is the file system. In Linux everything is a file, even external devices can be interfaced from user-space software. This is achieved thanks to udev [11], and in previous versions of the kernel, devfs. The kernel with the help of udev, creates a virtual filesystem for devices under the `/dev` directory.

All this allows to use the same system calls for regular text files and for external devices. As I explained in the previous section, the kernel acts as an intermediary. When a system call is invoked with a device file, the kernel loads the driver for the device and executes

it. An example of this interaction can be seen on Figure 8. When opening the `/dev/i2c-1` file, which represents the I2C bus 1, the kernel calls the `dev_open()`[5] method from the I2C driver. Similarly, when we call `write()`, the kernel internally calls the `dev_write()` method from the same driver.

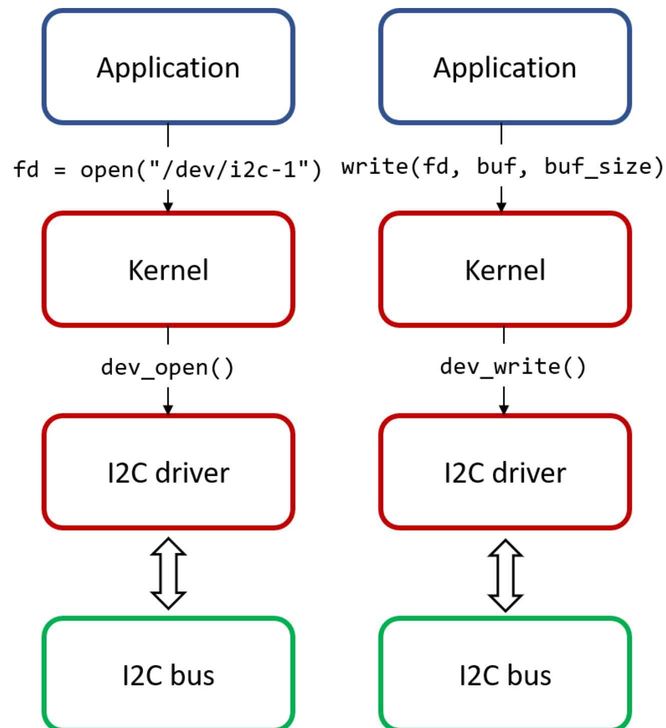


Figure 8 System calls with device files

This is only part of the truth. Regular text files don't need extra configuration for being read or written, but most devices do. For this there is an special system call, `ioctl()` [12], that takes arbitrary flags and arguments to configure the device.

Although reading and writing from a device is very easy, the configuration of the device tends to be very complicated if you don't have experience. Not because `ioctl()` is difficult to use, but because the documentation of what can be configured and how it can be configured is very scarce. Even for devices that are natively supported by the kernel.

[5] The name of the driver's method may vary. The consensus is to prepend the string "dev_", for device, to the name of the operations (open, read, write and release).

5.2. C++

C++ is a multiparadigm programming language. It was designed by Bjarne Stroustrup in 1979. At the start it was designed as an expansion of the C language, adding classes to it [13].

Over time the language has evolved from a "C with classes" to include concepts and constructions from object-oriented, functional and generic programming. C++ was standardized in 1998 with the release of C++98 (ISO/IEC 14882:1998), and now after four more standards, the ISO working group is preparing C++20. The last released standard is C++17 (ISO/IEC 14882:2017).

C++ has become one of the most popular languages for system programming. All major operative systems are either written in or have a large amount of components written in C++. It is also the one of the most popular languages in embedded environments according to a study by IEEE Spectrum [14]. It is commonly said that C and C++ have working compilers on almost all platforms.

Since its appearance, C++ has been appraised by its performance and its efficient use of resources, but it has also been heavily criticized for its lack of memory protection and its verbosity. Most of these issues have been (partially) solved with the inclusion of lambda expressions, smart pointers and type inference.

I've chosen this language because of its performance, efficiency, features, and my personal experience with it. I consider it has proven to be the best language for embedded programming in non-critical environments.

5.3. SQLite

SQLite is a relational database management system (RDBMS) that, unlike other RDBMs, does not have a server application. Instead the whole RDBMS is embedded in a C library [15]. This makes it the perfect database to embed to an application. Like most RDBMs it uses the SQL language for querying, inserting and updating information. Some popular web browsers and mobile applications use it to store information [16].

It's also a great database for embedded systems because it does not require a dedicated application for the database management that would consume both storage space and system resources.

Another great feature of SQLite is that the whole database is stored in a single file. This makes it a great candidate to implement a file format for data interchange between applications.

Although not the best database management system for real-time systems, with a couple of optimizations, it is capable of handling high amounts of data. By saving the journal in memory we not only save disk space, we save expensive IO operations. We can also save some time by "trusting" the OS that the data will be saved to disk at some point. Normally, SQLite waits for confirmation that the data has been written to disk. Note that these optimizations will violate the ACID principles.

I've used SQLite for the persistence of the data acquired by rt-data.

5.4. I2C, SPI and UART

I2C, SPI and UART are three serial protocols for intercommunication between microcontrollers.

5.4.1. I2C

I2C or I^2C (Inter-Integrated Circuit) is a serial bus and protocol, used to communicate integrated circuits with microcontrollers and processors. It follows a master-slave architecture, where only the master can initiate the communication.

It is possible to connect more than one slave to a I2C bus. For this reason, each slave is given a 7-bit address. Before each communication, the master will send first an address to the bus, and then it will start sending data.

The bus used two lines, one for data (SCA) and one for the clock (SCL). As it only has one data line, only half-duplex communication is possible (i.e. only one of the hosts, master or slave, can send data at a time).

I2C is used a lot in the embedded and IoT worlds. A lot of sensors use it to communicate with the microcontroller.

5.4.2. SPI

SPI (Serial Peripheral Interface) is also a serial interface used to communicate integrated circuits with a microcontroller. It also follows a master-slave architecture.

The SPI interface has 4 lines, the before mentioned chip select, a clock line, and two data lines. One data line is the master output slave input (MOSI), and the other master input slave output (MISO).

Unlike I2C, SPI is full duplex, it's possible that the master and slave send data simultaneously. Actually, all SPI communications are full duplex. For example, when the master wants to read a word from the slave, it must push a full word through the MOSI pin to receive a full word through the MISO pin. Its operation is similar to how two interconnected shift registers would work.

In a single SPI interface, it's possible to connect more than one slave. Instead of using addresses, SPI slave devices have a special pin, chip select (CS).

Like I2C, SPI is used a lot in the embedded and IoT worlds. Almost all sensors use either I2C or SPI to communicate with the microcontroller.

5.4.3. UART

UART (Universal asynchronous receiver-transmitter) is a device capable of asynchronous serial communication. By itself, UART is not a standard. Typically, the data is transmitted in two lines, transmission (TX) and reception (RX), to make full duplex communication possible. In most cases it's not possible to connect more than two UARTs together.

Unlike SPI and I2C, UART does not follow the master-slave architecture. Instead, as no clock is transmitted, the words have start and stop bits to separate them. Sometimes parity bits are also used to increase reliability. For the communication to work, both UARTs must use the same baud rate for transmission and reception.

Some parameters that can be configured in most UARTs include,

- Baud rate. Is the number of symbols per second. Normally when working with serial communication, a symbol is 1 bit long (i.e. the baud rate of a UART is the number of bits per second it receives/sends).

- The length of a character. The communication is done in characters, a packet of bits. The typical sizes are 5, 6, 7 or 8 bits.
- The type of parity. There are several possible meanings for the parity bit. The possible meanings (that has to be configured) are,
    - None. There's no parity bit.
    - Even. A parity bit with a value of 1 means that there is an even number of bits set to 1.
    - Odd. A parity bit with a value of 1 means that there is an odd number of bits set to 1.
    - Mark. There is a parity bit, but it's not used. It is always set to 1.
    - Space. There is a parity bit, but it's not used. It is always set to 0.

A lot of devices have UARTs for communication. For example, GPS receivers and Bluetooth transceivers. Most microcontrollers and microcomputers like the Raspberry Pi also have one or more UARTs.

5.5. JSON

JSON (JavaScript Object Notation) is a text format commonly used for data interchange [17]. One of its best features is that it's easy to read by humans and by machines. Its syntax is similar to the syntax of languages like C, C++ or Java.

JSON has six data types,

- Number. A sequence of decimal digits. Can hold both integers and real numbers. It can also have an exponent.
- Boolean. Either the value true or false.
- String. A sequence of Unicode characters. Must be surrounded by quotation marks. It can include escaped characters.
- Array. A sequence of zero or more values separated by commas. Each value can be of any valid data type.
- Object. A list of key-value pairs. Keys are always strings, and values can be any valid JSON data type. Each pair is called a property, and properties are separated by commas.
- null. An empty value.

I have decided to use JSON for the configuration files of rt-data.

The parsing of JSON is done using the json library by Niels Lohmann[6].

[6] Available at https://github.com/nlohmann/json

6. Requirements analysis

As previously explained (c.f. 2.2.1. Working methodology), I have followed an adapted version of Scrum. Requirements have been specified in the form of user stories.

As part of the requirements gathering process, I held several meetings with the Cosmic Research's engineers. To that, I added some requirements I considered interesting from the products mentioned in the state-of-the-art, and some innovative features. All this form the product backlog for rt-data, with a total of 37 user stories.

I categorized all 37 user stories in 5 epics,

1. **Sensors**: everything related to data acquisition from sensors.
2. **Events management**: everything related to the asynchronous passing of data between components.
3. **Support libraries**: support libraries related to data timestamping, thread management and timers.
4. **Control**: everything needed for control of the I/O ports (UART, I2C, …).
5. **Configuration**: everything related to the configuration of sensors, I/O ports, …

6.1. User stories

Below you can find the whole list of user stories. You will note that user stories' numbers are not consecutive, and some numbers are missing. It's the result of how Taiga give identifiers. Epics, user stories and even tasks are tracked using the same sequence. I have ordered them based on the sprint where they were implemented.

User story #7	**Points: 1** **Epic: Sensors**
Title:	Add new sensor
Description:	I want to be able to add a new sensor to the system so that it triggers the readings and dispatches the generated data
Acceptance criteria:	- When adding a new sensor, if the manager is already started, it must start the sensor. - When starting the manager, if the sensor was not previously started, it must start the sensor.

User story #8	Points: 1	Epic: Sensors
Title:	Remove sensor	
Description:	I want to be able to remove a sensor from the system so that it no longer manages it, for example in the case of a sensor failure.	
Acceptance criteria:	- After removing the sensor, no more events regarding that sensor can be produced.	

User story #5	Points: 5	Epic: Sensors
Title:	Background sensor reading	
Description:	I want to be able to read the system's sensors in the background asynchronously so that it does not lock the system	
Acceptance criteria:	- The sampling rate of the sensor must be configurable.	

User story #9	Points: 2	Epic: Sensors
Title:	Start & stop sensor reading	
Description:	I want to be able to start and stop the sensors readings at any moment so that I can control when data acquisition is performed.	
Acceptance criteria:	- After starting the sensor reading, the data acquisition starts. - After stopping the sensor reading, no more data is acquired from it. - After stopping the sensor reading, no more events regarding the sensor can be produced.	

User story #27	Points: 2	Epic: Sensors
Title:	Create new sensor	
Description:	I want to be able to create new sensors so that I can use my sensors with the framework.	

User story #27	Points: 2	Epic: Sensors
Acceptance criteria:	- Must provide an interface that allows to start and stop a sensor, acquiring and releasing the sensor's resources. - Must provide a default implementation of the background sensor reading (user story #5).	

User story #27	Points: 2	Epic: Sensors
Title:	Create new sensor	
Description:	I want to be able to create new sensors so that I can use my sensors with the framework.	
Acceptance criteria:	- Must provide an interface that allows to start and stop a sensor, acquiring and releasing the sensor's resources. - Must provide a default implementation of the background sensor reading (user story #5).	

User story #25	Points: 8	Epic: Events management
Title:	Asynchronous event dispatching	
Description:	I want to be able to asynchronously dispatch events (with or without attached data) to subscribed listeners so that the execution is not blocked.	
Acceptance criteria:	- When an event is dispatched all its subscribers are notified. - The listener's execution cannot block the execution of the event origin.	

User story #34	Points: 3	Epic: Support libraries
Title:	Get date and time in microseconds/milliseconds/seconds	

User story #34	Points: 3	Epic: Support libraries
Description:	I want to be able to get a timestamp since epoch in microsenconds, milliseconds and/or seconds so that I can keep track of execution time of certain tasks.	
Acceptance criteria:	- A timestamp can be converted between nanos, micros, millis and seconds at any time during the execution. - Timestamps must be comparable. - Timestamps added and substracted.	

User story #23	Points: 2	Epic: Events management
Title:	Create event type	
Description:	I want to be able to create new event types so that I can dispatch my own events.	
Acceptance criteria:	- An event must be able to contain (or to have attached) any kind of data. - The event must keep track of the origin of the event.	

User story #24	Points: 5	Epic: Events management
Title:	Create new event data type	
Description:	I want to be able to create new event data types so that I can dispatch events with my own data types.	
Acceptance criteria:	- The data of an event can contain a one or more data of any kind. - The data of an event must keep track of who generated it. - The data of an event must keep track of when it was generated.	

User story #31	Points: 8	Epic: Events management
Title:	Event data serialization/deserialization	

User story #31	Points: 8	Epic: Events management
Description:	I want to be able to serialize/deserialize an event and its attached data so that I can send it via a serial interface, or saving it to a custom filesystem.	
Acceptance criteria:	- At least it shall be possible to serialize C++ standard data types (int, bool, char, float, string). - It shall be possible to serialize in different formats like JSON or "raw bytes". - It shall be possible to add support to more formats in the future. - The result of the serialization, independently of the format, must be exported as an array of bytes. - Once a object is serialized, it shall be possible to deserialize it to obtain the same object.	

User story #16	Points: 5	Epic: Sensors
Title:	Sensor data management	
Description:	I want that the framework manages all the data produced by the sensors and dispatches it to subscribed listeners so that the user doesn't have to implement it every time.	
Acceptance criteria:	- When a new data is acquired by a sensor, it will be notified to all the sensor's subscribers.	

User story #26	Points: 8	Epic: Events management
Title:	Save event data to SQLite database	
Description:	I want to be able to store in an SQLite database the dispatched events and their attached data so that I can persist all the acquired data.	
Acceptance criteria:	- All SQL statements for table creation and insertion must be generated by the framework.	

User story #26	Points: 8	Epic: Events management
	- Must support "batch" inserts. - The execution of SQL statements cannot block the execution of the producer of the events.	

User story #28	Points: 2	Epic: Events management
Title:	Create event subscribers/listeners	
Description:	I want to be able to create new subscribers/listeners that will subscribe to events so that I can process the dispatched events.	
Acceptance criteria:	- Listeners won't be notified until subscribed.	

User story #29	Points: 2	Epic: Events management
Title:	Subscribe/listen to an event	
Description:	I want to be able to subscribe/listen a subscriber/listener to an event so that I can process it.	
Acceptance criteria:	- After subscribing, the listener will be notified when the next event is sent. Past events won't be notified.	

User story #10	Points: 5	Epic: Sensors
Title:	Sensor analog driver	
Description:	I want to be able to use analog sensors without having to implement the driver and value conversions each time so that I can use my analog sensors with the framework.	
Acceptance criteria:	- This driver must use the ADC kernel driver installed in the machine. - Must support any voltage level from the sensor or the machine. These parameters shall be configurable. - Must support any resolution (in bits) that the ADC might have.	

User story #10	Points: 5	Epic: Sensors
	- The result must be already converted to the desired units.	

User story #19	Points: 3	Epic: Configuration
Title:	JSON configuration file parsing	
Description:	I want the framework to be able to parse configuration files that use the JSON format so that I can edit the configuration in a human-readable format.	
Acceptance criteria:	- It must support all valid JSON data types (string, number, bool, null, array and object).	

User story #20	Points: 2	Epic: Configuration
Title:	Get a configuration item	
Description:	I want to be able to get a configuration item in a tree-like manner so that the configuration properties can be grouped.	
Acceptance criteria:	- From a configuration item it must be possible to access its children configuration. - A configuration item value must be able to be represented in any C++ standard data type (string, int, float, bool, char, array).	

User story #21	Points: 2	Epic: Configuration
Title:	Set a configuration item	
Description:	I want to be able to modify a configuration item even after a configuration file has been parsed so that I can "patch" configuration properties with a "faulty" value.	
Acceptance criteria:	- A modification in a configuration item will not have any side effect in its children nor its parent.	

User story #21	Points: 2	Epic: Configuration
	- When modifying a configuration item value, it must accept any C++ standard data type (string, int, float, bool, char, array).	

User story #22	Points: 8	Epic: Configuration
Title:	Multi-level configuration files	
Description:	I want the framework to provide a configuration mechanism using files that allow a tree-like (multi-level) access, so that I don't have to recompile everything to change a configuration parameter.	
Acceptance criteria:	- Conceptually, the configuration shall be accessed as if it was a m-ary tree. - Each configuration item can hold a value of any valid C++ type. - The configuration file could be implemented in any file format.	

User story #13	Points: 5	Epic: Sensors
Title:	UART driver for implementing sensors	
Description:	I want to be able to use sensors that have an UART interface without having to implement the UART driver each time so that I can use my sensors with an UART interface with the framework.	
Acceptance criteria:	- This driver must use the system (kernel) UART driver. - It must be possible to receive a single byte or an array of bytes. - The baud rate must be configurable. - The number of start/end bits and parity bits must be configurable.	

User story #17	Points: 3	Epic: Sensors
Title:	Sensor configuration	
Description:	I want the framework to provide an easy mechanism to configure the sensors without having to re-compile everything, preferably using a configuration file.	
Acceptance criteria:	- It must be possible to configure a sensor using a configuration file.	

User story #33	Points: 5	Epic: Support libraries
Title:	Threads with custom scheduling	
Description:	I want to be able to modify the scheduling of the threads managed by the framework so that I can give them more priority over other processes and threads to achieve a greater "real-timeliness".	
Acceptance criteria:	- It must be possible configure the application's thread to use a real-time thread scheduling (RT FIFO and RT Round Robin).	

User story #42	Points: 5	Epic: Control
Title:	UART control driver	
Description:	I want to be able to control the UART bus of my system so that I can control the behaviour of a connected device.	
Acceptance criteria:	- This driver must use the system (kernel) UART driver. - It must be possible to send a single byte or an array of bytes. - The baud rate must be configurable. - The number of start/end bits and parity bits must be configurable.	

User story #14	Points: 5	Epic: Sensors
Title:	GPS Sensor	
Description:	I want to be able to receive data from a GPS receiver and produce events based on the received data so that I can use my GPS receivers with the framework.	
Acceptance criteria:	- This sensor must use a system (kernel) driver. - At least data about the position (latitude and longitude), velocity, altitude and signal health shall be acquired.	

User story #32	Points: 8	Epic: Events management
Title:	Send event data by HTTP or using TCP sockets	
Description:	I want to be able to send an event and its attached data by HTTP or using TCP sockets so that I can notify another host of the event. This could be useful to implement a GUI.	
Acceptance criteria:	- When sending by TCP, it must be possible to send the data to any IP address and port. - When sending by HTTP, it must be possible to send the data to any host name or IP, and port. - When sending by HTTP, the data will be sent using a POST request.	

User story #30	Points: 8	Epic: Events management
Title:	Save event data to File	
Description:	I want to be able to store in a file the dispatched events and their attached data so that I can persist the acquired data.	
Acceptance criteria:	- It must be possible to save the file anywhere in the filesystem. - The file can have any format. At least JSON and “raw bytes” must be supported.	

User story #36	Points: 5	Epic: Control
Title:	GPIO control driver	
Description:	I want to be able to control the GPIOs of my system so that I can control the behaviour of a connected device.	
Acceptance criteria:	- This "driver" must use the system (kernel) driver. - It must be possible to request and set the status (HIGH or LOW) of a pin just providing its number. - It must be possible to set a pin as input or output.	

User story #11	Points: 5	Epic: Sensors
Title:	Sensor SPI driver	
Description:	I want to be able to use sensors with an SPI interface without having to implement the SPI driver each time so that I can use my SPI sensors with the framework.	
Acceptance criteria:	- This "driver" must use the system (kernel) driver. - It must be possible to receive a single byte or an array of bytes. - It must support all SPI modes. Default mode must be SPI mode 0, as is a *de facto* default mode. - The baud rate must be configurable.	

User story #12	Points: 5	Epic: Sensors
Title:	Sensor I2C driver	
Description:	I want to be able to use sensors with an I2C interface without having to implement the I2C driver each time so that I can use my I2C sensors with the framework.	
Acceptance criteria:	- This "driver" must use the system (kernel) driver. - It must be possible to receive a single byte or an array of bytes.	

User story #38	Points: 5	Epic: Control
Title:	SPI control driver	
Description:	I want to be able to control the SPI bus of my system so that I can control the behaviour of a connected device.	
Acceptance criteria:	- This “driver” must use the system (kernel) driver. - It must be possible to send a single byte or an array of bytes. - It must support all SPI modes. Default mode must be SPI mode 0, as is the *de facto* default mode. - The baud rate must be configurable.	

User story #37	Points: 5	Epic: Control
Title:	I2C control driver	
Description:	I want to be able to control the I2C bus of my system so that I can control the behaviour of a connected device.	
Acceptance criteria:	- This “driver” must use the system (kernel) driver. - It must be possible to send a single byte or an array of bytes.	

User story #39	Points: 3	Epic: Control
Title:	Create new state machine	
Description:	I want to be able to create a new state machine that represents the state of a system or subsystem.	
Acceptance criteria:	- The state machine must always have a current state.	

User story #40	Points: 3	Epic: Control
Title:	Add state to state machine	

User story #40	Points: 3	Epic: Control
Description:	I want to be able to add a new state to a state machine so that the new state of the system can be properly managed, and the system's state is represented correctly.	
Acceptance criteria:		

User story #41	Points: 5	Epic: Control
Title:	Add transitions to state machine	
Description:	I want to be able to add a new transition between two states to a state machine so that the states of the system can be properly managed.	
Acceptance criteria:	- When the state machine changes the current state, the coherence of the system must be preserved by making any necessary change to the system.	

User story #35	Points: 2	Epic: Control
Title:	Create new control driver	
Description:	I want to be able to define new control drivers for my control devices, extending existing drivers or creating new ones so that I can add support for new control devices.	
Notes:	Marked as obsolete.	

User story #15	Points: 8	Epic: Sensors
Title:	IMU sensor	
Description:	I want to be able to read data from an IMU sensor and produce events based on the received data so that I can use my IMUs with the framework.	
Acceptance criteria:	- At least it must acquire data regarding acceleration (in m/s^2) and rotation (in rad/s). - Shall support using system (kernel) drivers.	

User story #15	Points: 8	Epic: Sensors
	- Shall try to support as many sensor models as possible, or at least shall support to add compatibility to more models in the future.	

User story #43	Points: 3	Epic: Support libraries
Title:	Start & stop a timer	
Description:	I want to be able to start and stop timers so that I can run tasks periodically.	
Acceptance criteria:	- It must be possible to program a task to be executed with a precision of at least 10 milliseconds. A task is a piece of code with no inputs and no outputs. - It must be possible to program a timer with nanoseconds, microseconds, milliseconds or seconds. - When a timer is stopped, it won't execute any more task.	

6.2. Non-functional requirements

Although some non-functional requirements are already expressed in the acceptance criteria from the user stories, the following list has all of the non-functional requirements that the framework must comply with.

Non-functional requirement #1	
Title:	Asynchronous operations
Description:	All potentially long-running tasks without a delimited execution time must not block the execution of the rest of the application.
Justification:	In a real-time application, blocking the execution for an uncertain amount of time completely breaks the "real-timeliness" of the application.

Non-functional requirement #2	
Title:	Compatible operating systems
Description:	The framework must be fully compatible with the GNU/Linux operating system.
Justification:	The embedded devices this framework targets, primarily use GNU/Linux as their operating system.
Acceptance criteria:	The framework must work at least, under Debian Jessie 8.9 and Ubuntu 16.04.

Non-functional requirement #3	
Title:	Using kernel drivers
Description:	For performance reasons, when interacting with I/O devices, the framework must (at least) support working with kernel drivers.
Justification:	Kernel drivers are executed in kernel mode, meaning that have a much more direct access to the hardware that allows for a much lower latency.

Non-functional requirement #4	
Title:	Soft real-time
Description:	The framework must allow implementing soft real-time systems.
Justification:	Not having steady sampling rates can result in incomplete or "faulty" data.
Acceptance criteria:	The maximum "steady" sampling rate for the sensors must be at least of 50Hz.

Non-functional requirement #5	
Title:	User-friendliness

Non-functional requirement #5	
Description:	The framework must be usable by users with little prior experience in implementing data acquisition software for the devices this framework targets.
Justification:	The main objective of this framework is to build an easy-to-use tool for implementing data acquisition systems.

Non-functional requirement #6	
Title:	Extensibility
Description:	The framework must be extendable to support new sensors and devices.
Justification:	It's almost impossible to give support to all the sensors and devices that exist in the market.

Non-functional requirement #7	
Title:	Error logging
Description:	When an application implemented this framework encounters an error, a log explaining what happened must be generated.
Justification:	It's important for identifying what has failed without having to debug the application.
Acceptance criteria:	After the application finishes executing (successfully or not) a log file will be generated with all the errors it has encountered.

7. Design and architecture

When starting this project, I had to decide about how I would design rt-data, as a library or as a framework. There is a very thin line dividing what is considered to be a library and framework. The general consensus seems to be that a framework forces you to use a determinate architecture while a library does not.

In my (probably limited) experience, most data acquisition applications, embedded or desktop, tend to implement the same architecture over and over. Therefore, I decided to design rt-data as a framework.

The rt-data framework is built around events. They're a great way to represent changes in a system, instead of asking for changes we can simply wait for it. Instead of making a component responsible of constantly asking for changes, and possibly saturating the CPU, we can invert it and make the author of changes responsible of notifying them.

We can extend that to any periodic or background task. For example, for sensor reading. Instead of periodically asking a sensor to read, we can just wait to be notified by it.

Furthermore, we can attach data to an event so that the event author can send any relevant data. In the sensors example, we could not only notify about a new reading, we could also send it. This way we can communicate several components with a very loose coupling.

7.1. Logical architecture

This is not the typical three layers application (presentation, business and data layers), the user's requirements tell us that the user needs to be able to access directly to the underlying IO devices but with a more usable interface.

Instead, the framework has been designed so that the applications that use it, have three logical layers, see Figure 9.

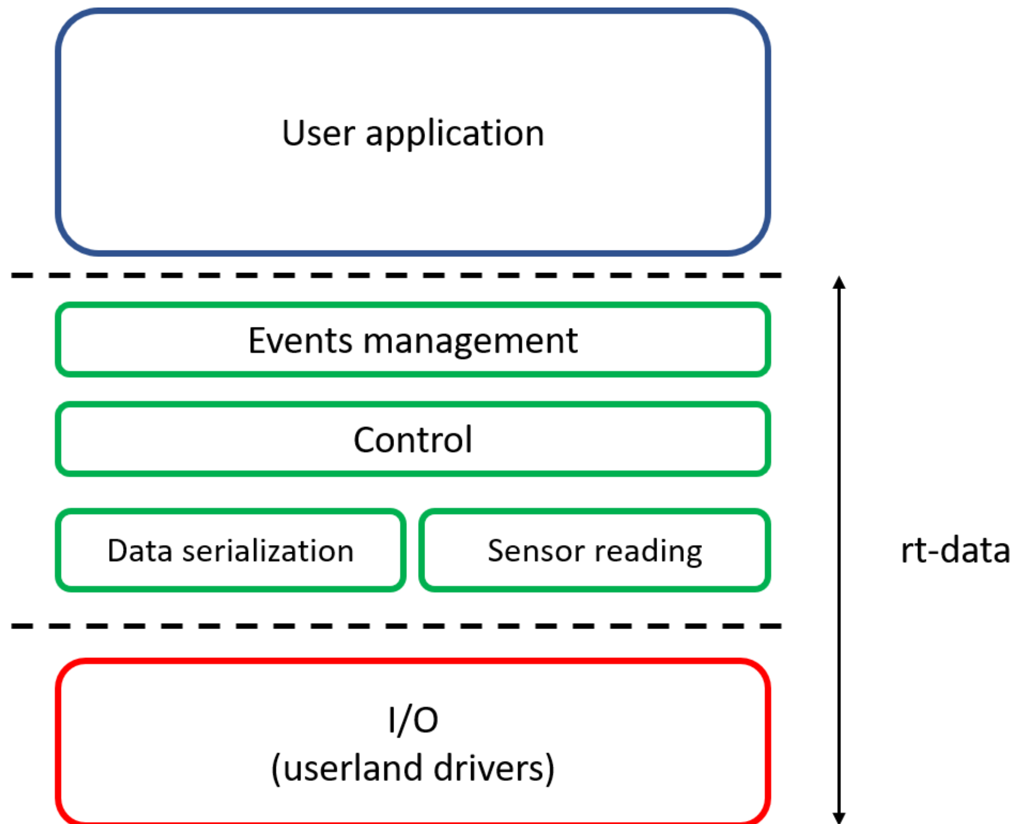


Figure 9 Logical architecture in layers

7.2. Layer's design

The responsibility of each layer, and its divisions, is explained in the following three sections.

7.2.1. User application layer

It's where the specific logic of the application. The idea is that the user has complete freedom on how to design this layer. The framework only provides an entry point for the application execution.

7.2.2. Events management layer

It's the main layer of the application. The primary way of interacting with rt-data is through events. Anyone can send or listen for an event. When an event is sent, the framework then dispatches it to all subscribed listeners. Listeners are programmable and can be considered to be part of the logic of the main application.

The interaction between the user application and this layer is done by adding or removing listeners, and by sending events.

Some listeners can be control components. They listen for state changes and using the IO layer, can make changes in the behaviour of external devices.

Another user of the IO layer are the sensors. They use it for reading from a device (a "hardware" sensor), and then make the necessary data conversions to obtain usable values[7]. Sensors are the only responsible of reading from their device, and of sending an event each time a new value is read.

Some listeners might also want to save the data in the filesystem or in a remote host. For this, they have access to a set of data serializers. They produce a byte representation of an event data that they can save using an I/O component.

7.2.3. I/O layer

This layer implements the access to I/O "devices" through the operating system. The access can be done by implementing userland drivers or by interfacing with kernel drivers installed in the system.

It is possible to interface two types of "devices". The ones that are used by control components, that are used to control the behaviour of external devices. It is very hard, near impossible, to create an abstraction for these devices, because each one of them is different.

The other type of devices are used to persist the event data. These devices range from a simple text file to a relational database. Obviously, these are much simpler to abstract.

---

[7] Some sensors send values encoded or packed in a way that it's not usable by the application. An example are analog-to-digital conversers, that send a value between 0 and 2^n (where n is the resolution of the ADC in bits) that is useless until it is converted.

8. Implementation

Having talked about the overall architecture of the solution, it is time to talk about how it was implemented. Previously I talked that I've followed an agile working methodology. It is important to note that agile methodologies encourages to instead of doing a full specification and design of the solution at the start of the project (Inception), to do a "basic" logic design (c.f. 7.1. Logical architecture) and in each sprint refining it. This is the reason why in this section I will mix what typically is done during the design phase and what is done during the implementation phase.

In this section I will talk about how the components presented in Figure 9 have been implemented, which design patterns I've applied and I'll also present the interactions between the different components.

The framework has been fully implemented in C++, except for the build tools and some testing components. It has been divided in packages (conceptually because packages don't exist in C++). A full diagram of the solution's package can be seen at Figure 10.

The rt-data framework has been designed to work with at least the C++14 specification. As for the C++ standard library, rt-data works with the GCC standard library. At least versions 7.3.0 and 4.9.0, and in x86-64 and ARM architectures.

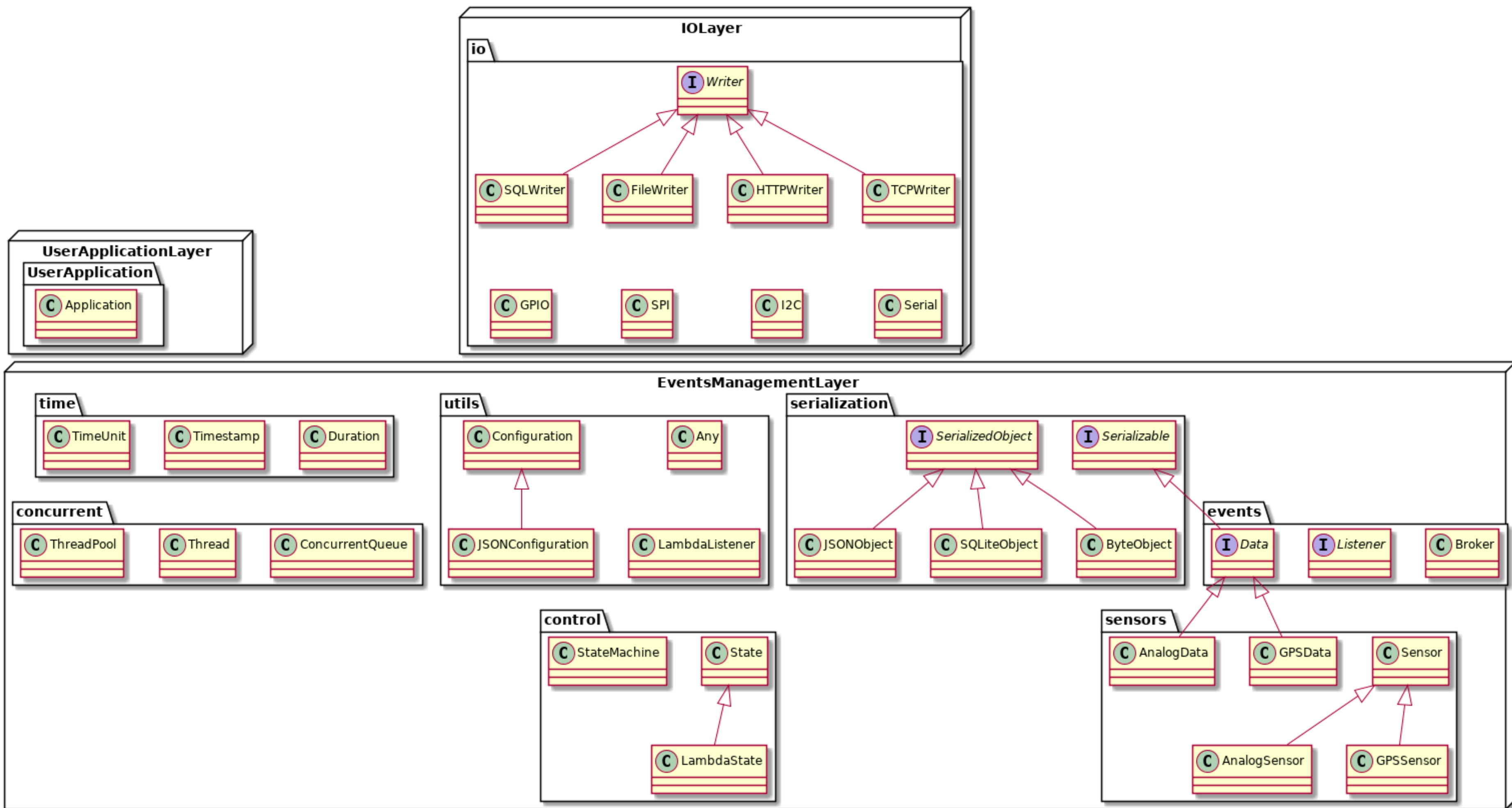


Figure 10 Package diagram

8.1. Events management

Event management is the core of this framework. It has been designed so that most of the interactions between its components are done through events. All events are part of a topic, which is just a short text description of all its events. For example, a topic for all the readings of a GPS sensor could be "`read_gps`".

Events can also have an associated data. Data that is common to all events are,

- A timestamp with the moment of creation.
- A text that represents the origin, or creator, of the event.

All this data is hold by a single object that is instance of class `Data` (c.f. Figure 11). This class can be subclassed to include more data. In the example of the GPS readings, we might want to include the coordinates.

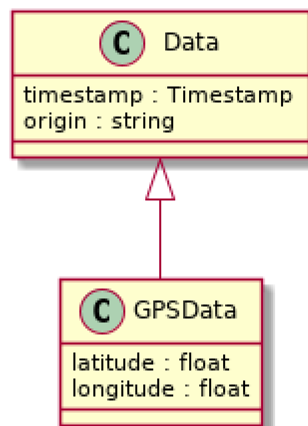


Figure 11 Data class

For the handling of the events, I have implemented a variation of the observer pattern. An instance of the class `Broker` holds a list of `Listeners`. These `Listeners` can subscribe to a topic, and then the `Broker` will notify them once it receives a new event from an event source. Listener is an interface that can be implemented by any class just by implementing a method with this signature, `void handle(std::string topic, std::shared_ptr<Data> data)`.

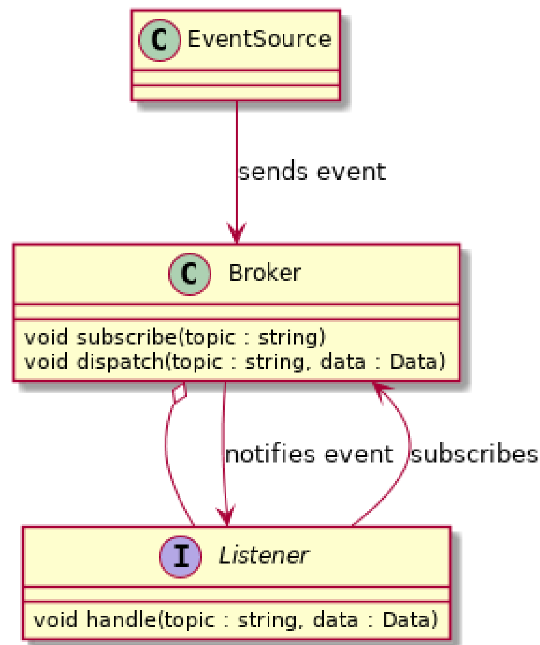


Figure 12 Broker and listeners[8]

An event is not directly handled in the same thread it was send to the Broker. Instead, it is handled asynchronously. The Broker has a queue of tasks to be executed and a pool of threads. Once a thread is available, it starts executing a new task. For each Listener to be notified, a new task is created.

Events are sent from the event source to the Broker, using the dispatch method. A pseudocode explanation of what the method does follows,

```
dispatch(topic, data):
    for listener in listeners[topic]:
        thread_pool.add_task(listener.handle, topic, data)
    endfor
end
```

The sequence diagram for this interaction is available at Figure 13.

[8] The class EventSource is just a placeholder for a class that generates events, not an actual class in the framework

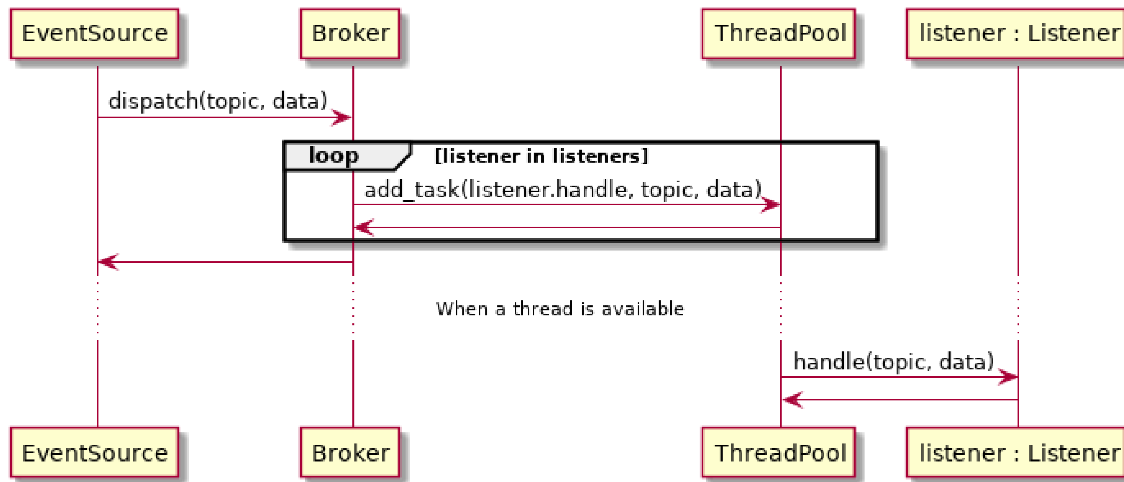


Figure 13 Sequence diagram for the dispatch method

Listeners are saved in the broker inside a `std::unordered_map` with the topic name as the key, and a list of Listeners as the value. This class is part of the C++ STL (standard library), and internally uses a hash table. The memory usage is worse than a standard map, but the performance is much better. The average cost of the search, insert and delete operations are, in average, O(1) for this container.

Listeners can be subscribed to a topic by calling the subscribe method from a Broker instance, with the topic name and a pointer to the Listener as a parameter. A pseudocode implementation of this method follows,

```
subscribe(topic, listener):
    listeners[topic].push_back(listener)
end
```

The sequence diagram for this interaction is available at Figure 14.

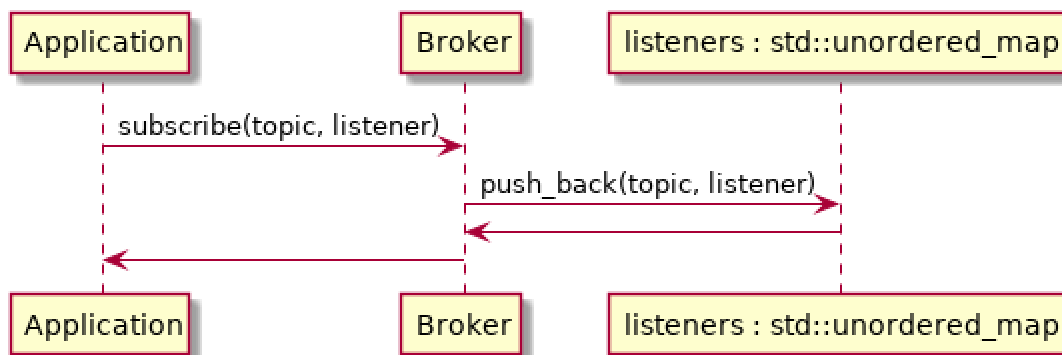


Figure 14 Sequence diagram for the subscribe method

As most components in rt-data, I have implemented the Broker with a two-phase initialization. Before starting using the Broker, the user must call the `start()` method and after using it, he must call the `stop()` method. These methods acquire and release the resources needed by the class.

I've implemented this two-phase initialization because in C++ declaring a class' instance, like,

```
Broker broker;
```

Will automatically call the Broker's constructor. This behaviour might be unwanted for some users. What could be even worse is that if another class has a Broker as a member, its constructor will be automatically called when constructing the class' instances. For example,

```
class AClass {
private:
    Broker broker;
}
```

In this case, when an instance of AClass is created, the constructor of Broker is called. This could mean that the instantiation of AClass could take longer because of the acquisition of the Broker's resources. At least with a `start()` method, the acquisition is more explicit.

For releasing the resources of a class, there are two options,

1. Releasing the resources in the destructor. This is done implicitly when the class goes out of scope. It could delay the return from a method.
2. Releasing the resources in a dedicated method. This is done explicitly. If the user does not call this method the resources are, potentially, never released.

I decided to implement a mixture between the two options. The user should call the `stop()` method to release resources, but if he doesn't do it, it is called from the destructor.

When passing instances of Data and Listener, and as these classes are meant to be subclassed, we cannot pass them by copy or reference. We must pass a pointer to the instance. This is a C++ “limitation”, the compiler only reserves space for the declared type. Passing a subclass is allowed by the compiler but the members of the subclass will be lost.

Passing “raw pointers”, such as `int*`, is discouraged by the newer C++ standards (C++11 and later) and instead it’s recommended to use “smart pointers”. Smart pointers’ behaviour is similar to how references work on Java, although there’s no garbage collector. Smart pointers have a raw pointer and an atomic counter with the number of instances. Every time a copy of the smart pointer is made, the counter is incremented. Each time that the destructor is called, the counter is decreased. If the counter reaches 0, the raw pointer is freed. I used one type of smart pointers called shared pointers (`std::shared_ptr`) that allows the pointer to be copied and to be passed as a parameter in a method.

8.2. Sensors management

Sensors management is another core component of the framework. It has been designed so that the user doesn’t have to directly interact with it. Sensors are managed by two classes,

- An instance of the class `Sensor` per hardware sensor. Responsible of managing the status of the sensor and acquiring its data. The data acquisition is done by default through polling. This class is meant to be subclassed for every sensor type.
- An instance of the class `SensorsManager` per application. It’s responsible of recollecting the data acquired from all the sensors from the system, and of dispatching it to a Broker instance. This class is not meant to be sublassed.

The architecture of this package can be seen at Figure 15.

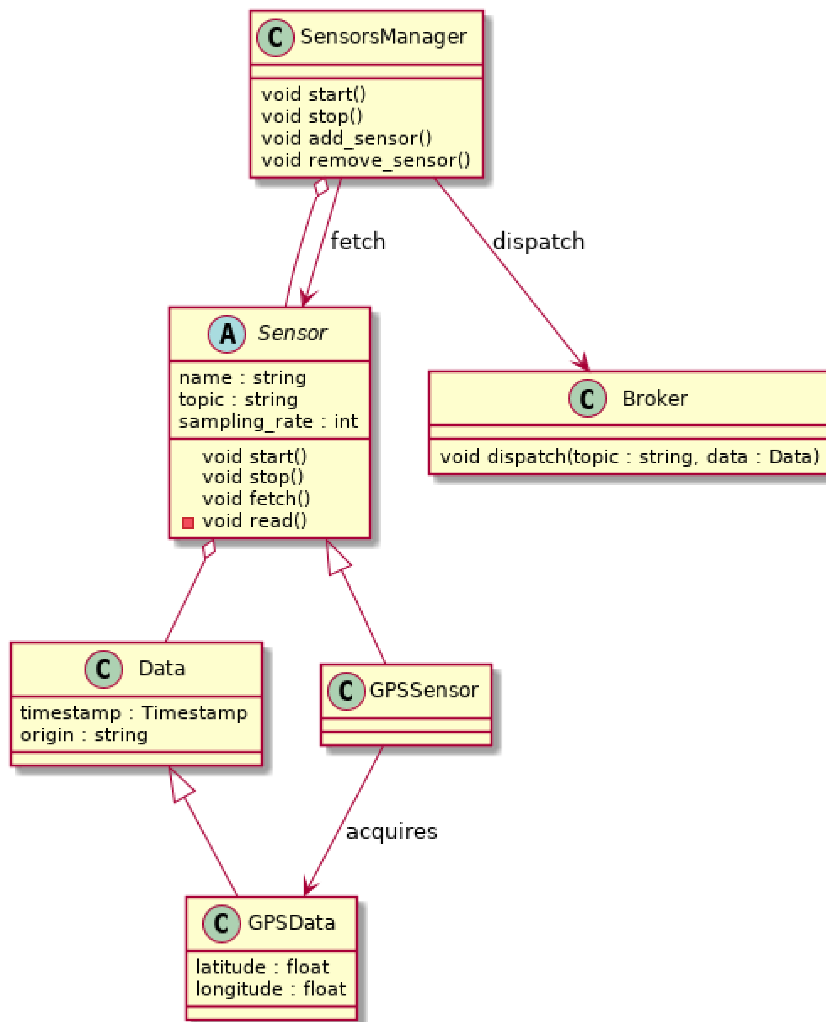


Figure 15 Sensor and SensorsManager

The data acquired by a `Sensor` is then stored in a subclass of `Data`. Typically, each subclass of `Sensor` will have its own subclass of `Data`. In the previous diagram, the `GPSSensor` saves its data into instances of `GPSData`.

Every 10 milliseconds, the `SensorsManager` will call the `fetch` method from all `Sensor` instances with a pointer to a `Broker` instance. Each `Sensor` holds a queue of `Data` to be dispatched. When the fetch method is called, the queue is emptied, and all the data is dispatched to the `Broker` instance.

This polling of the `Sensor` instances is done in a separate thread so that the execution of the rest of the application's logic is not interrupted.

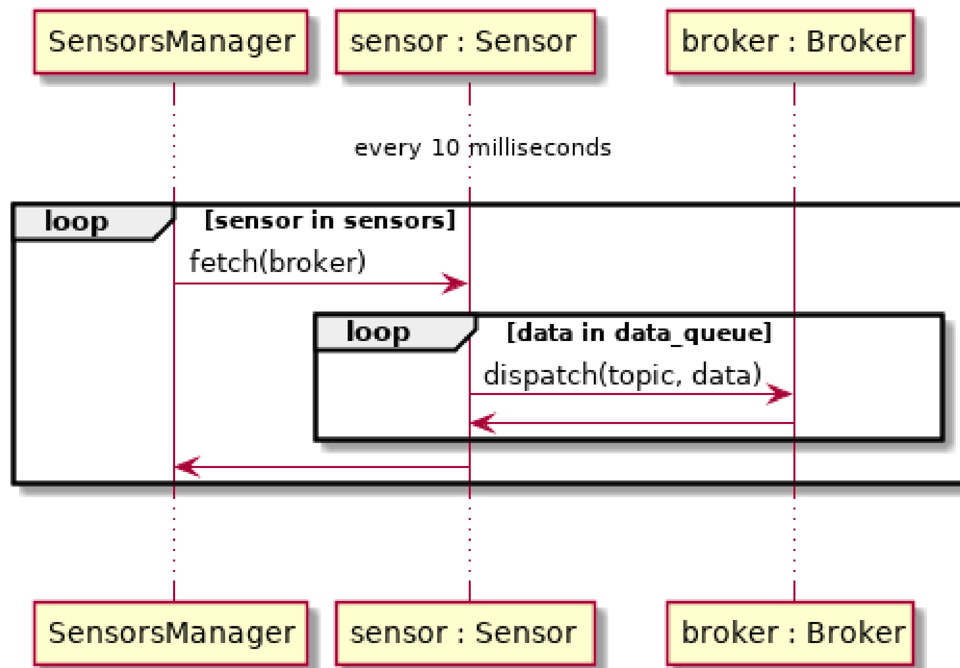
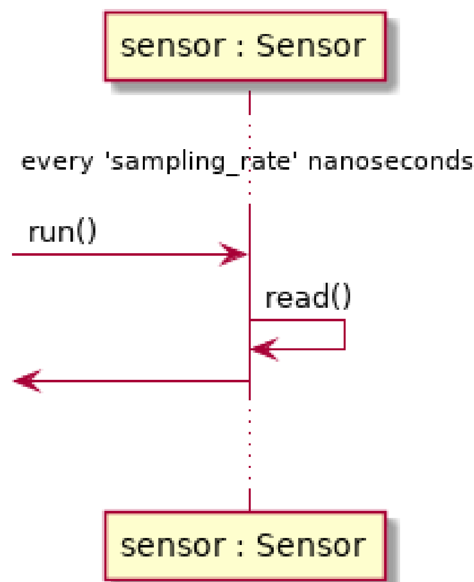


Figure 16 Sequence diagram for the fetch method

For the data acquisition of the sensor, the `Sensor` class has a pure virtual or abstract method called `read`. This method is called periodically in a separate thread owned by the `Sensor` instance. The rate, in nanoseconds, at which the `read` method is called can be configured by setting the `sampling_rate` parameter either in the constructor or by calling the `set_sampling_rate` method.



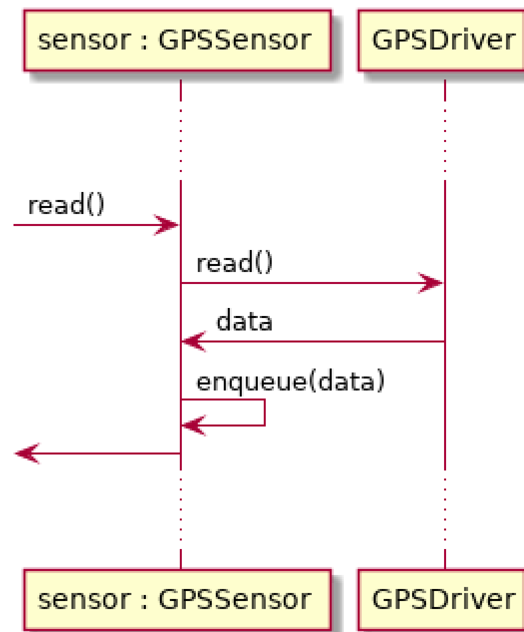


Figure 17 Sequence diagram for the read method and example

Other common configurations for all `Sensor` instances are the name of the sensor, that will be used to populate the origin value from the generated `Data`, and the topic where these data will be dispatched.

Both `Sensor` and `SensorsManager` classes make use of the two-phase initialization explained in the previous section. When calling the `start` method, the internal threads used for reading and fetching the data are started. When calling the `stop` method, the threads are stopped and joined.

8.2.1 GPS sensor

For implementing a GPS sensor, I subclassed the `Sensor` and `Data` classes as `GPSSensor` and `GPSData`. The data that is acquired by this sensor includes,

- The date and time reported by the GPS satellite.
- Coordinates (latitude and longitude).
- Altitude.
- Ground and vertical speeds.
- Other data related to the quality of the GPS reception, and the uncertainty of the data received.

GPS "sensors" (actually GPS receivers) are a special kind of sensors. Unlike most of them, GPS receivers cannot be read. Instead they are constantly sending updates to the computer they are connected to.

To handle this behaviour, I am using an open-source project called gpsd (GPS daemon). It's a separate application that runs as a daemon in the background. It is connected through a serial port to the GPS receiver and parses the data it receives into JSON. Then, any other application can connect to this daemon through TCP and ask for the GPS data.

As part of the gpsd project, there are also C, C++ and Python libraries that can be used to integrate the gpsd daemon into other applications. I'm using the C++ library.

A pseudocode implementation of the `read` method of `GPSSensor` follows,

```
read()
    gpsd_data = gpsd.read(timeout=10s)
    gps_data = GPSData(gpsd_data)
    data_queue.enqueue(gps_data)
end
```

It is important to set a timeout; in case the daemon is not available or if it's not possible to establish a new TCP connection. Although the timeout is set to 10 seconds, it won't block the rest of the application as it is running on a separate thread.

As configuration parameters, the GPSSensor only takes the hostname and the port where the gpsd daemon is running. In most cases, the default values for these two parameters are enough. The default hostname is `localhost`, and the default port is `DEFAULT_GPSD_PORT` (internally set to 2947).

8.2.2. Analog sensor

The `AnalogSensor` is a `Sensor` that reads from the analog-to-digital converter (ADC). For the data acquired by this sensor, I subclassed the `Data` class as `AnalogData`. It only holds a real (`double`) value.

Although it has a single output value, this sensor has a big number of configuration parameters,

- **`file`**. The file setup by udev to access the ADC.
- **`zero_value_voltage`**. The voltage read by the ADC at which the sensor reads a value of 0.
- **`span_value_voltage`**. The voltage read by the ADC at which the sensor reads the maximum.
- **`scale`**. The maximum value that can be read by the sensor.
- **`quantization_bits`**. The resolution of the ADC in bits.
- **`zero_voltage`**. The lowest possible voltage at which the ADC reads a value. Typically, 0 volts.
- `span_voltage`. The maximum possible voltage at which the ADC reads a value. Typically, 3.3 or 5 volts.

The ADC reads a voltage that is then converted to an integer between 0 and $2^{\text{quantization_bits}}$. This number must be then converted to a more meaningful value. This is done using the following formula,

$$value = (ADC\ value - zero)\ \times \frac{scale}{span}$$

Where,

$$zero = \frac{2^{quantization\ bits} * (zero\ value\ voltage - zero\ voltage)}{span\ voltage}$$

$$span = \frac{2^{quantization\ bits} * span\ value\ voltage}{span\ voltage}$$

A pseudocode implementation of the read method follows,

```
read()
    fd = open("/dev/adc")
    adc_value = read(fd)
    value = convert(adc_value)
    data = AnalogData(value)
```

```
    data_queue.enqueue(data)
end
```

As you can see, we are using the standard system calls `open` and `read` for reading from the sensor. As we have seen previously, the operating system internally will call the ADC driver. The actual sensor reading is done at the driver level. Normally when a board has an ADC, an ADC driver is installed by default.

8.3. Control

The control package allows to build control systems taking advantage of the event system. I have built a state machine that responds to the events sent by a `Broker` instance. The architecture of this package is based on two classes,

- `State`. Represents a state of the system. It has a condition that must be met for the system to change to it, and two actions. One is executed when the system's current state changes to this state, and the other when the system changes its current state again.
- `StateMachine`. Holds a set of `States` and serves as a `Listener` of events. When the `Broker` sends a new event, it checks the condition of all states. If one of them is met, the current state is changed to it.

The `State` class is meant to be subclassed for each state to implement. All States must implement the following methods,

- `arrive`. It's called when the state machine's current state is set to this State. It takes no parameters and has no return value.
- `leave`. It's called when the state machine's current state changes from this state to another one. It takes no parameters and has no return value.
- `check_condition`. It's called when the state machine receives a new event. It shall check if the state machine shall change its current state to this state. It takes as parameters a pointer to the current state, the topic of the event sent by the `Broker` and the associated data.

Instead of subclassing `State`, it's also possible to use the `LambdaState` class. This class acts as a wrapper with three lambdas, one for each method from `State` to be implemented.

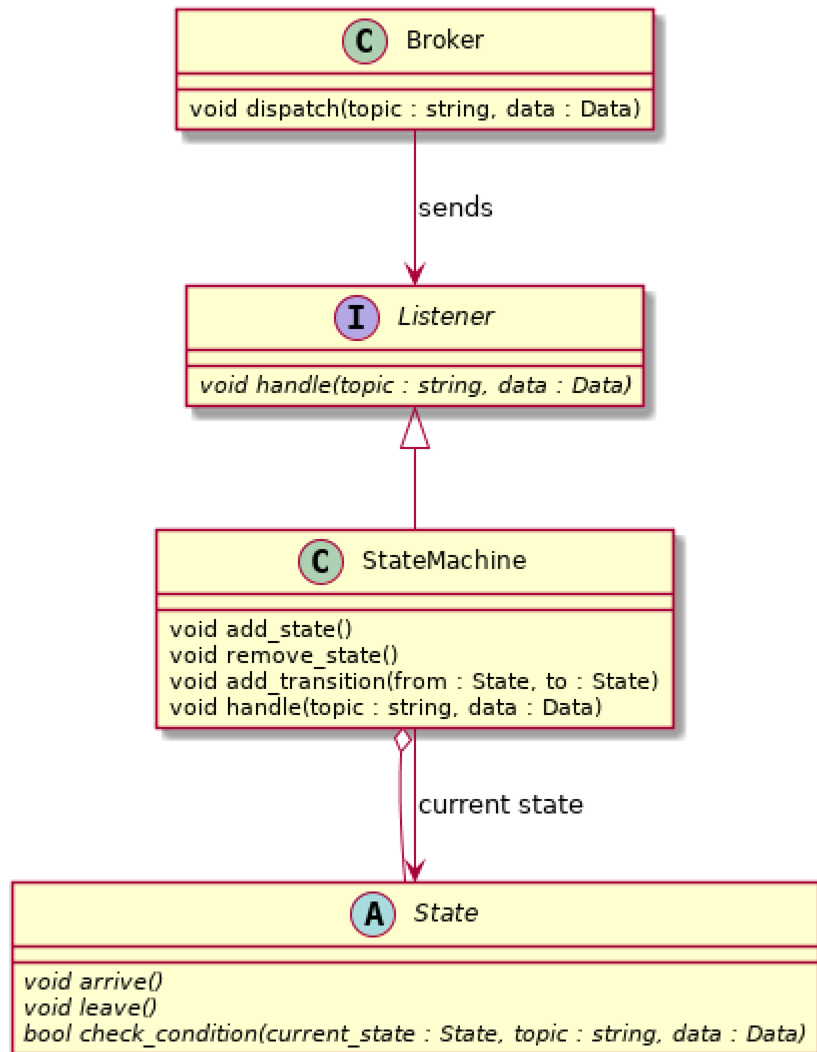


Figure 18 State and StateManager

When a `StateMachine` is built, a start `State` must be provided. Later, more states can be added by calling the `add_state` method and removed with the `remove_state` method.

Not every `State` is reachable from all the other `States`. For this reason, the `StateMachine` holds a map of all the reachable `States` for each `State`. To add a new transition the `add_transition` state must be called with the origin and destination `State`.

As I said previously, the state changes are done when receiving a new event from a Broker instance. The `StateMachine` class implements the `Listener` interface and its `handle` method. A pseudocode implementation of this method follows,

```
void handle(topic, data):
    for state in transitions[current_state]:
        if state.check_condition(current_state, topic, data):
            current_state.leave()
            current_state = state
            current_state.arrive()
        endif
    endfor
end
```

The sequence diagram of this method is the following,

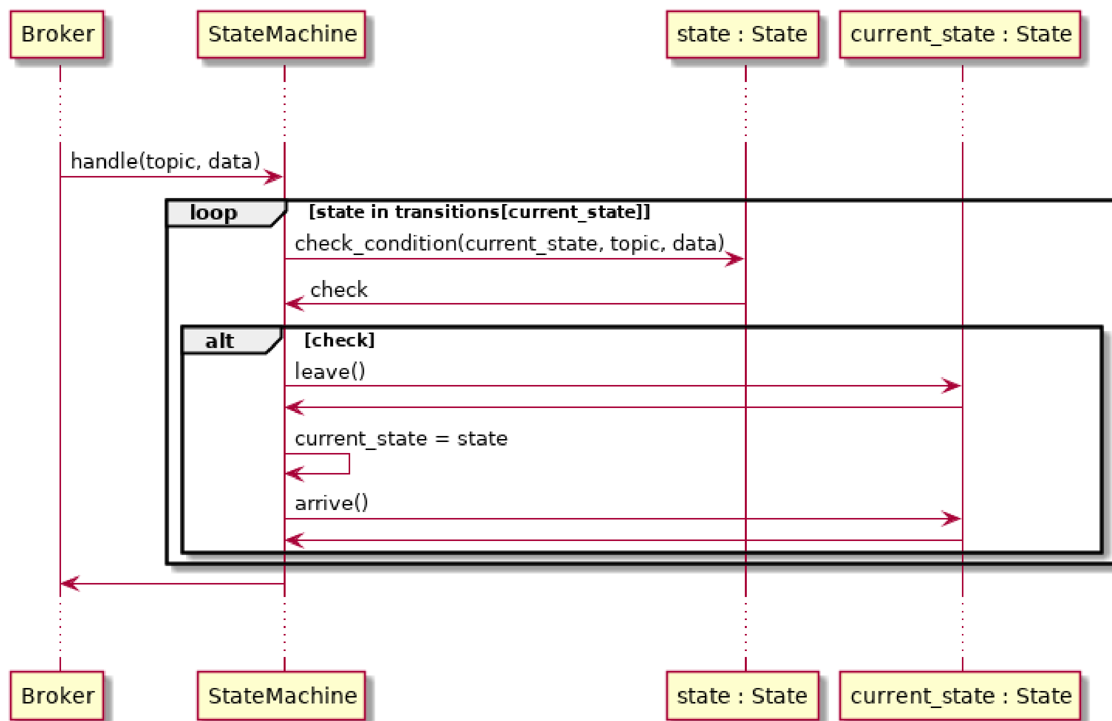


Figure 19 Sequence diagram of the handle method

Internally, the `StateMachine` holds an unordered set (a hash table where the key and value have the same object) of smart pointers of `States`. Here are stored the states added by calling the `add_state` method. The transitions are stored in an unordered

map with a raw pointer to the origin state as the key, and a list of pointers to the destination states as the value. The current state is stored as a raw pointer.

8.4. Serialization

To be able to save the event data to a database or to send it to remote hosts, it's necessary to have a way to change its representation to a format that can be understood by both parties. This process is called serialization.

The architecture of this package is composed by two classes and interfaces,

- `Serializable`. An interface that should be implemented by any class that will be serialized.
- `SerializedObject`. The classes that implement this interface represent the result of serialization in a given format.

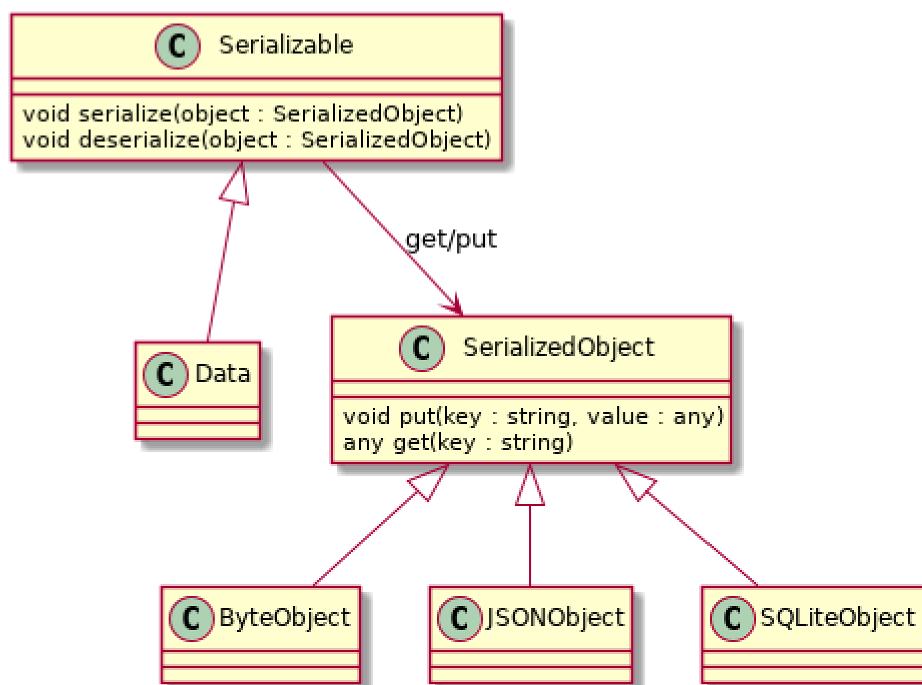


Figure 20 Serializable and SerializedObject

When a `Serializable`'s serialize method is called with a pointer to a `SerializedObject`, it will call the `put` methods from the object. These methods take a string key that identifies a property and a value of most of the standard C++ types (int, float, bool, string, …). Similarly, when the deserialize method is called, the

Serializable will call the get methods from the SerializedObject to reconstruct the serialized object.

As an example, the pseudocode implementation of the serialize and deserialize methods from the Data class follow,

```
void serialize(serialized_object):
    serialized_object.put("timestamp", timestamp)
    serialized_object.put("origin", origin)
end
```

```
void deserialize(serialized_object):
    timestamp = serialized_object.get("timestamp")
    origin = serialized_object.get("origin")
end
```

With this implementation, the serializables don't need to know the specifics of each format. Furthermore, as the interface is based on key-value pairs, it's perfect for implementing the serialization to formats like JSON or even SQL queries.

To obtain the result of the serialization process, all SerializedObjects must implement the method get_bytes that returns an array of bytes.

In the following subsections I will discuss the specifics of each implementation of SerializedObject.

8.4.1. ByteObject

In this implementation, the serialized values are stored with their "raw" bytes and "packed" into an array of bytes. Each packet is prepended by the total size of the packet, and each value is prepended with its size in bytes.

The typical structure of the packets is like this,

Total size 4 bytes	Size of property 1 1 byte	Property 1 256 bytes max	…	Size of property N 1 byte	Property N 256 bytes max

The packet for a serialized Data would look like this,

Total size 4 bytes	Size of timestamp 1 byte	Timestamp 4 bytes	Size of origin 1 byte	origin 256 bytes max

As you can see the string key of the serialized properties is ignored. The reason behind this decision, is that this serialization is meant to use as little space as possible. This behaviour is documented.

8.4.2. JSONObject

This implementation produces a JSON object from a `Serializable`. As the `SerializedObject`'s put and get methods have key-value pairs as parameters, and JSON objects are just a collection of key-value pairs, the implementation should be very easy.

In fact, the JSONObject class is just a wrapper as it uses the JSON library by Niels Lohmann to create the JSON documents.

Besides getting the result as a byte array, it's also possible to get the json object from the library, and the JSON object as a string.

8.4.3. SQLiteObject

This implementation produces the `CREATE TABLE` and `INSERT` statements to be executed in an SQLite database. For the inserts, it produces prepared statements that can be compiled on the database. Then, the "serialized" properties can be bound to the prepared statement by passing it to the `bind_values` method.

The name of the table must be provided when creating a `SQLiteObject` instance. It is needed to build the `CREATE TABLE` and `INSERT` statements.

The two outputs generated by this serialization process are the prepared statements for the inserts and the `CREATE TABLE` statement. As it makes no sense to have a byte representation of this serialization, the `get_bytes` method returns an empty byte array.

8.5. IO

Another important part of the rt-data framework is the I/O, as it allows access to the hardware and the file system. This package has been given two duties, the access to the data buses (serial, SPI, I2C) and the writing of the event data to the file system, databases or to remote hosts.

8.5.1. Writing of event data

When acquiring data, it's important to save it in a non-volatile storage. We might want to analyse it afterwards or, in case of failure we might want to know what happened.

My idea when designing this package was to make the API implementation-agnostic. Following the template pattern, I implemented a common interface `Writer`. A class that uses one writer, can be modified to accept another one with minimal changes in the code.

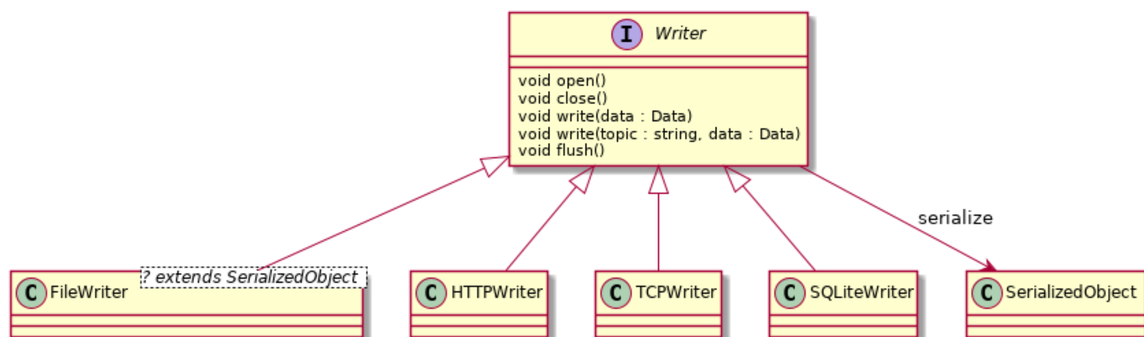


Figure 21 Writer and writers

The writer interface defines two `write` methods. One only with an instance of Data, and one that also includes the topic where the data was sent. Other than that, it is up to the implementation to define what's the difference between the two.

The interface also has a `flush` method. Its usage may vary between implementations, but the idea is that after this method has been called, all the data passed to a `write` method has been effectively written.

The open and `close` methods follow the same two-phase initialization as the rest of the framework. When calling open, all the resources needed by the writer, are initialized. And when calling `close`, all the resources are freed.

Typically, a `Writer` implementation will make use of a `SerializedObject` to change the format of the event data before writing it. The specific method of serialization is specific for each writer.

FileWriter

The `FileWriter` is an implementation of the `Writer` interface, to write event data to a file. Internally, it uses the ostream class from the fstream header (C++ standard library) to handle the actual writing to files. The only parameter needed to build this writer is the name of the file to be written.

This writer is generic to use any type of `SerializedObject`. It will call the `get_bytes` method and write what it returns into the file.

HTTPWriter

This implementation can "write" (send) an event data to an HTTP endpoint. To send the HTTP packet it uses libcurl, a free software library that is widely used. The only parameter needed to build this writer is the URL of the HTTP endpoint.

This writer uses the `JSONObject` class for serialization and sends what the `get_bytes` method returns.

In this implementation, the `flush` method in this implementation does nothing.

TCPWriter

The TCPWriter sends event data through TCP to remote hosts. It uses Linux sockets. The parameters needed to build this writer are the IP address of the host and the port of the destination application.

This writer uses the `ByteObject` class for serialization and sends what the `get_bytes` method returns.

In this implementation, the `flush` method in this implementation does nothing.

SQLiteWriter

The SQLiteWriter is an implementation of the Writer interface that writes event data to a local SQLite database. It uses the SQLiteCpp library by Sébastien Rombauts[9] that is a wrapper around sqlite3. The sqlite3 "library" is the SQLite implementation in C (remember that SQLite is a SQL database that is completely implemented in just a C library that can be included in any application). From sqlite3 I'm using what the call "the amalgamation" that is all its source code in a single header file.

For better performance, the user of the framework can enable two "optimizations". One is disabling the synchronous commit. This way SQLite won't wait for the OS to confirm the writes. The other is storing the journal in memory. Note that these changes break with the ACID principles.

When a data is passed to be written, the writer stores it in a buffer. When the buffer is full, all the data in the buffer are inserted in the database using a transaction. The size of the buffer is set to 500 by default but can be configured by the user.

This writer uses the `SQLiteObject` class for serialization of the event data. When one is passed to be written, it is passed to an instance of `SQLiteObject`. If the table for the data doesn't exist, it is created with the statement returned by the `get_create_table` method. Then, a prepared statement for the insert is created with the statement returned by the `get_insert` method. All prepared statements are cached for better performance, and the next time a data of the same type is passed to be written, the cached prepared statement will be used. Finally, the values from the data are bound to the prepared statement by calling the `bind_values` method from the `SQLiteObject` instance, and the prepared statement is executed.

8.5.2. Data buses

This package includes classes that can be used to manage some of the data buses present in most microcontrollers and microcomputers. As you can see in Figure 22, the classes in this package do not follow any common interface, even when they perform very similar operations. The reason is that each one of them needs a different interface.

[9] Available at https://github.com/SRombauts/SQLiteCpp

I2C needs an address to write to, SPI is capable of full-duplex transfers, and it makes no sense to read or write an array of bytes from a GPIO.

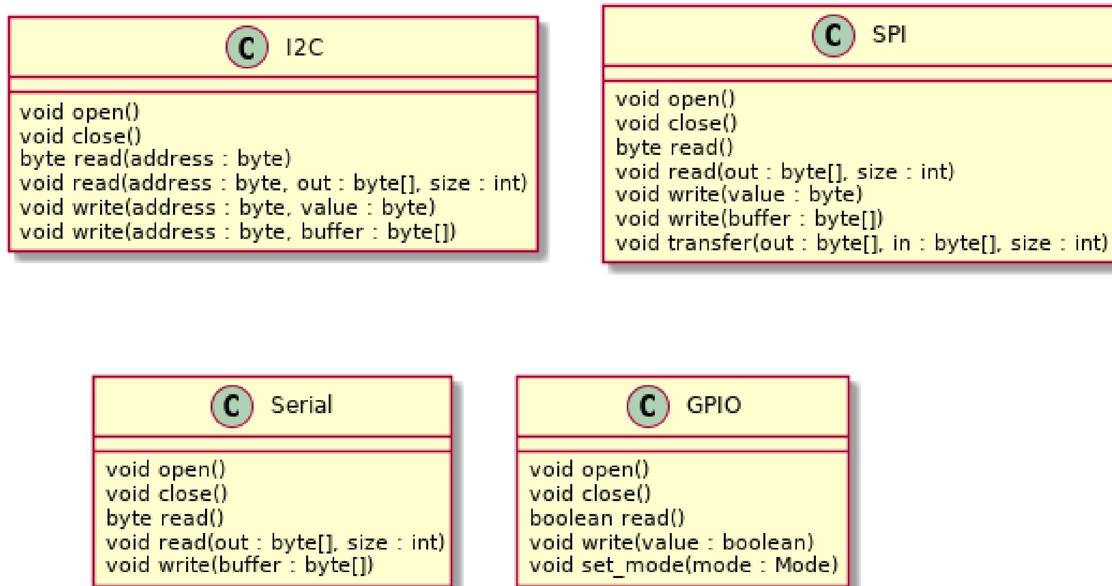


Figure 22 Data buses

I2C

An instance of the `I2C` class can be used to manage a single I2C bus. It uses the i2c-dev library that is part of the Linux kernel. To build an instance, the path to the file created by udev or sysfs that represents the bus must be passed.

With this class it's possible to read and write from a I2C bus a single byte or an array of bytes. To read and write it's necessary to pass to the `read` or `write` methods, the address of the register to read/write. As the I2C addresses are 7 bits long but the smallest amount of memory we can address is 8 bits. This might introduce some issues as some I2C devices expect the padding bit at the start of the address, but others require it at the end.

This class uses the same two-phase initialization we've seen previously in other classes.

SPI

An instance of the `SPI` class can be used to manage a single SPI interface. It uses standard system calls like `ioctl` to read and write from the bus.

When building an instance, the user is capable of selecting the SPI mode, the number of bits per word, and the speed of the communication in Hz. When building the instance, it's also necessary to pass the path to the file that represents the SPI interface, that is created by udev or sysfs.

We can read and write a single byte or an array of bytes by using the `read` and `write` methods. It is also possible to make full-duplex transfers by calling the transfer method with two arrays of bytes, one for the bytes to be sent and one that will hold the received bytes.

It also uses a two-phase initialization.

Serial

An instance of the Serial class can be used to manage a single TTY terminal, primarily UART interfaces. It uses the termios Unix API. Using this API, it's possible to read and write using system calls like `read` and `write`.

When building an instance, the user can configure the baud rate, the size of a character and the type of parity. By default, the baud rate is set to 9600 bits/s, the size of the character is set to 8 bits, and the parity type is set to none. This is the default configuration in some devices like the Arduinos. When building the instance, it's also necessary to pass the path to the file that represents the TTY.

We can read and write a single byte or an array of bytes by using the `read` and `write` methods.

This implementation also used a two-phase initialization.

GPIO

Using an instance of the `GPIO` class it's possible to manage a single general-purpose input output (GPIO) pin. The Linux sysfs exposes the file `/sys/class/gpio/export` to which we can write to ask the kernel the control of a GPIO pin [18]. For example, if we write "5" to this file, it will create the directory `/sys/class/gpio/gpio5/` with two interesting files,

- “direction”. Where we can write “in” to set the pin as input, or “out” to set it as output.
- “value”. If the pin is set as an output, we can write a “1” or a “0” that will set the pin as high or low.

This class uses these sysfs files and directories to manage a GPIO pin. When building an instance, we must pass the number of the pin that will be managed.

Then, we can read the status of the pin by calling the `read` method, that will return either HIGH or LOW (these values are members of the `PinStatus` enum). If set as an output, we can set the status of the pin by calling the `write` method with HIGH or LOW as a parameter. We can set the pin as output or input by calling the `set_status` method with INPUT or OUTPUT (these values are part of the Mode enum).

The GPIO class also uses a two-phase initialization.

8.6. Timestamping

As we’ve seen previously, all `Data` instances are timestamped with the time of its creation. This is very important as it allows for a better traceability of the acquired data. This package has three classes,

- `Timestamp`. Uniquely identifies a point in time.
- `Duration`. Represents the time between two points in time.
- `TimeUnit`. Represents a time unit (seconds, milliseconds, …).

`TimeUnit` instances hold the number of nanoseconds that the time unit it represents has. For example, the instance that represent the microsecond will hold the value 1.000 because a microsecond is equivalent to 1.000 nanoseconds. Because it holds nanoseconds, the smallest time unit that can be represented is the nanosecond.

The `TimeUnit` class has a convenience method that allows to convert an integer from one time unit to another,

```
int64 convert(value, original_timeunit, destination_timeunit):
    return (value*original_timeunit)/destination_timeunit
end
```

For convenience, the C++ operators * and / have been overloaded. A `TimeUnit` can be multiplied and divided by another `TimeUnit` or by a 64-bit integer. We just multiply or divide the number of nanoseconds of the first operand with the number of nanoseconds of the second one.

`Timestamp` instances hold the number of nanoseconds since the Unix epoch (1st January 1970). They can be created in three ways, by calling the constructor with an integer that represent an amount of nanoseconds since epoch, by calling the static method `from_duration` with an integer time and its `TimeUnit`, or by calling the now method. This method takes no arguments and will return a `Timestamp` instance with the number of nanoseconds since epoch until now.

To get the current timestamp I've used the chrono header from the C++ standard library. By calling the `std::chrono::high_resolution_clock::now()` method, we can obtain the current timestamp from the clock with the highest resolution.

For convenience, the Timestamp class overloads the +, -, >, < and == C++ operators,

- Subtracting two timestamps. Will return a `Duration` instance representing the time duration between the two points in time.
- Adding a `Duration` to a `Timestamp`. Will return a `Timestamp` instance with the number of nanoseconds of the `Duration` added to the number of the nanoseconds of the `Timestamp`.
- Subtracting a `Duration` from a `Timestamp`. Will return a `Timestamp` instance with the result of the subtraction of the number of nanoseconds of the `Duration` from the number of the nanoseconds of the `Timestamp`.
- Comparing if two `Timestamps` are equal. Will return a boolean value indicating if both timestamps have the same number of nanoseconds since epoch.
- Comparing if a `Timestamp` is greater than/less than another `Timestamp`. Will return a boolean value indicating if the number of nanoseconds from one `Timestamp` is bigger/smaller than the other one.

A `Timestamp` instance can be converted to an integer representing seconds, milliseconds, microseconds or nanoseconds by calling the methods `to_seconds`, `to_millis`, `to_micros` or `to_nanos`.

`Duration` instances hold two nanoseconds timestamps. There are two ways to build an instance, either by calling the constructor with two nanosecond values or by calling the constructor with an integer value and its `TimeUnit`.

A `Duration` instance can be converted to an integer representing seconds, milliseconds, microseconds or nanoseconds by calling the methods `to_seconds`, `to_millis`, `to_micros` or `to_nanos`.

8.7. Concurrency

The concurrency package includes classes that help with the concurrency management and synchronization. The classes it includes are,

- `Thread`. An extension of the `thread` class from the C++ standard library.
- `ThreadPool`. An implementation of a thread pool.
- `ConcurrentQueue`. A thread-safe queue.

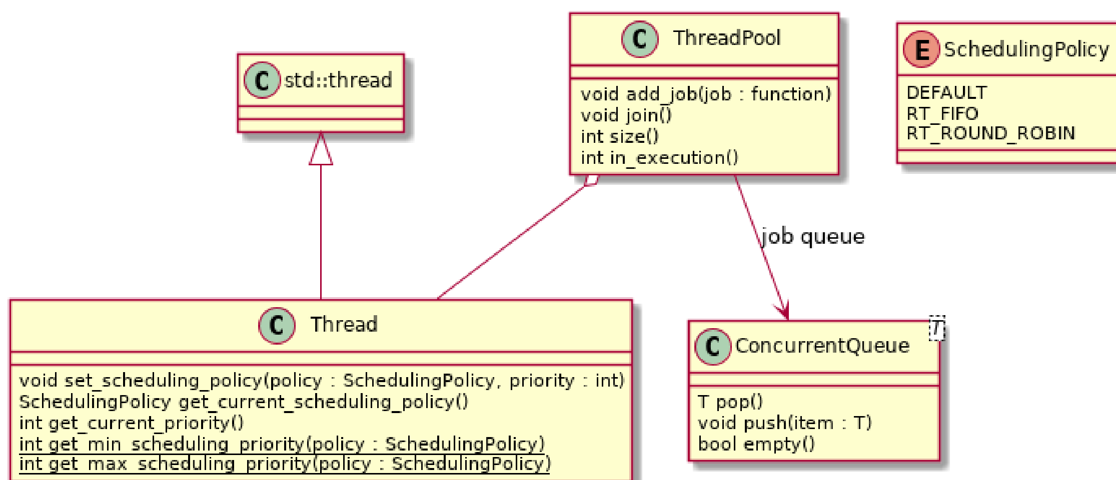


Figure 23 The 'concurrent' package

The `Thread` class adds the ability of configuring the scheduling policy of a thread. The policies are defined by the Linux scheduler. The policies that can be used in an rt-data thread are,

- Default scheduling, SCHED_OTHER.

- First-In, First-Out (FIFO) scheduling [19]. It's a real-time scheduling policy. Each thread is given a priority from 1 to 99. When scheduled for execution, it's put at the end of the queue for its priority. Once the CPU is available, the thread at the head of the highest priority queue is executed.
- Round-Robin (RR) scheduling. It's a real-time scheduling policy. Is an extension of the FIFO scheduling, in which each thread is only allowed to run for a maximum time quantum.

Note that threads with a real-time scheduling policy will always pre-empt the execution of a non-real-time thread. In rt-data the scheduling policies are represented by the `SchedulingPolicy` enumeration.

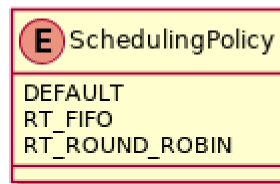


Figure 24 SchedulingPolicy enumeration

The scheduling policy and the priority can be set by calling the `set_scheduling_policy` method with a `SchedulingPolicy` and an integer representing the priority as parameters. The current scheduling policy can be gotten by calling the `get_current_scheduling_policy` method and the priority can be gotten by calling the `get_current_priority` method. The minimum priority for a scheduling policy can be gotten by calling `get_min_scheduling_priority`, and the maximum priority by calling `get_max_scheduling_priority`. For all these operations, rt-data uses the POSIX Threads (pthreads) API.

The `ThreadPool` instances hold a list of `Thread` instances. The number of threads in the pool is configurable, but by default it's 10 threads. The pools execute atomic jobs that have no parameters and no return values and are stored as std::function objects from the C++ standard library. The jobs are put into a `ConcurrentQueue`.

To add a new job, the user must call the `add_job` method with a std::function job as a parameter. This job is put into the job queue. If a thread is available and waiting for a

new job, it will be notified and it will start executing the new job. If no thread is available, the job will wait in the queue until a thread is available.

A pseudocode implementation of what each thread executes follows,

```
void thread_run():
    while !stopped:
        new_job.wait(continue if stopped || !job_queue.empty())
        if !stopped:
            job = job_queue.pop()
            job()
        endif
    endwhile
end
```

To wait for a new job, the threads use a `std::condition_variable`, from the C++ standard library, that will block the thread until it's notified by another thread or the condition passed as a parameter to the `wait` method evaluates as true. The `new_job` condition variable is notified in the `add_job` method,

```
void add_job(job):
    job_queue.push(job)
    new_job.notify()
end
```

To stop the execution of all the threads in the pool, the user must call the `join` method. It will notify all the threads and then it will join them,

```
void join():
    stopped = true
    new_job.notify()
    for thread in threads:
        if thread.joinable():
            thread.join()
        endif
```

```
    endfor
end
```

The `ConcurrentQueue` class is a thread-safe wrapper around the `std::queue` class from the C++ standard library. It uses a mutex for synchronization, so that only one operation of the queue can be performed at a time. Although better thread-safe implementations exist, this solution introduces almost no complexity and has been proven to be good enough.

It provides "just" three operations, `pop`, `push` and `empty`. `pop` removes and returns the item at the head of the queue. `push` adds a new item to the end of the queue. And `empty` returns whether the queue is empty or not.

8.8. Configuration

The configuration package provides classes that can be used to configure the application and its components without needing to recompile the code using configuration files. It includes the following classes,

- `Configuration`. An abstract class that can be subclassed to implement a tree-like structure of pairs of key-values that stores configuration properties. Each instance of `Configuration` is a node of the configuration tree. Each node has a key and a value.
- `JSONConfiguration`. A concrete implementation of `Configuration` that loads configuration properties from a JSON file.
- `Any` and `AnyImpl`. Can be used to store values of any data type.

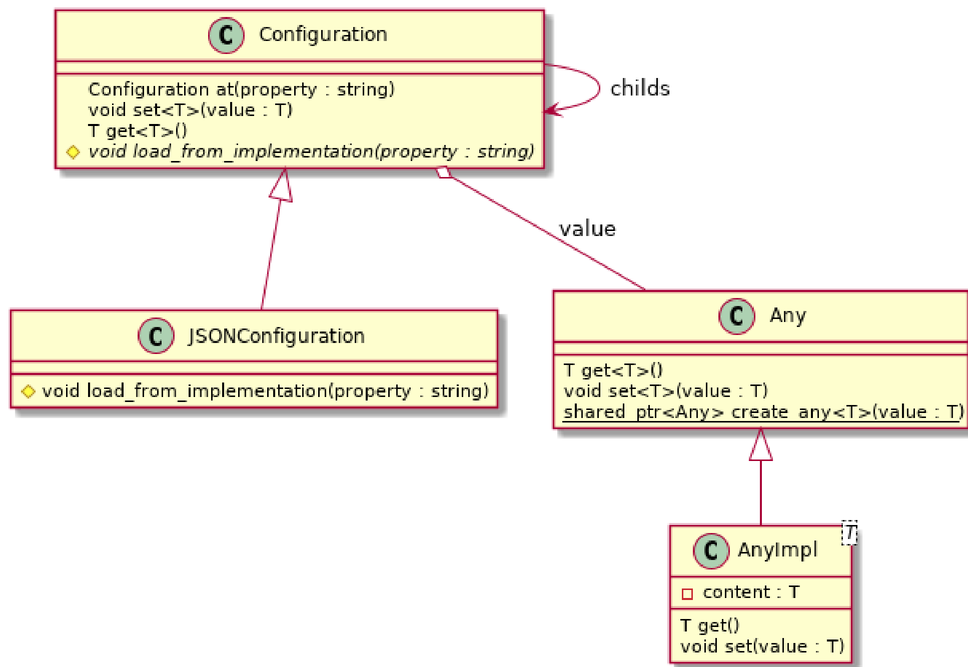


Figure 25 Configuration package

A configuration tree might look like the one in Figure 26.

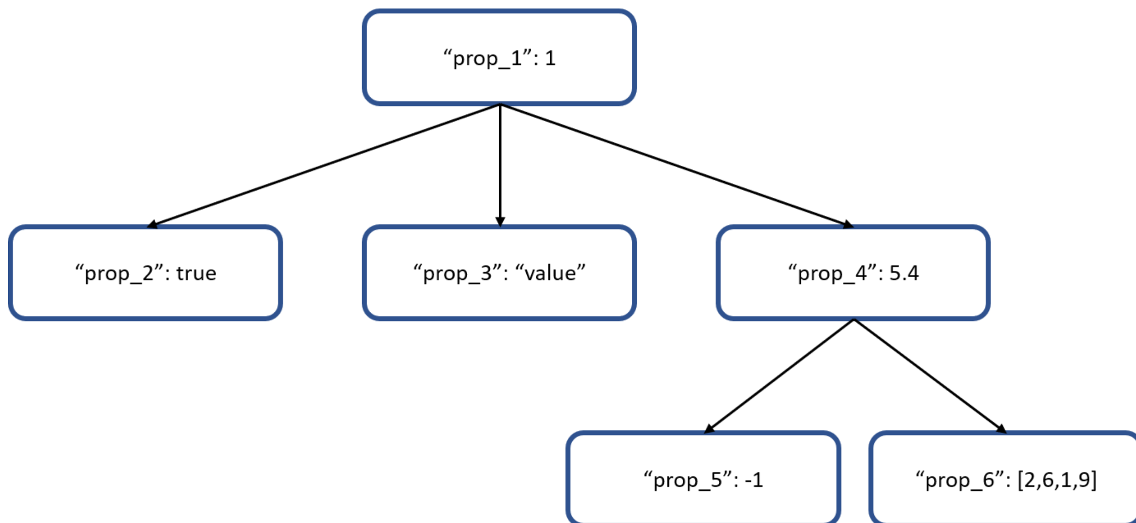


Figure 26 Configuration tree

The configuration tree can be accessed through a `Configuration` instance (a node of the tree). The access is always done from top to bottom, and it's not possible to access a parent node from a child node. To access a child node, the user can invoke the method at on a `Configuration` node with the key of one of its child nodes as a parameter. The keys of the nodes are always strings but the value can be of any data type. To access the

value stored in a node, we can call the `get` method. The value can also be changed by calling the `set` method with the new value as a parameter.

The children of a node are stored in an unordered_map (hash table). It is lazily initialized. When a property is requested for the first time, it is loaded from the underlying configuration file to the configuration tree.

The `Configuration` class does not implement any way of loading the configuration from a configuration file, it's delegated to the subclasses. Following the template method pattern, they must implement the method `load_from_implementation` that takes a key and (potentially, if it exists) stores a `Configuration` node in the tree of child nodes. This method is called when the user calls the `at` method with a key that does not exist in the tree of child nodes.

A pseudocode implementation of the at method follows,

```
Configuration at(property):
    if property not in childs_map:
        load_from_implementation(property)
    endif
    return childs[property]
end
```

The `JSONConfiguration` class has an implementation of the `load_from_implementation` method that loads the `Configuration` node from a JSON file. The parsing of the JSON files is handled by the beforementioned json library by Niels Lohmann. Using JSON files as configuration files will impose two limitations,

1. Only leaf Configuration nodes will store a value. After loading the file any node can be modified to include a value by calling the `set` method.
2. The JSON standard only supports a limited set of data types (c.f. section 5.5. JSON). After loading the file all nodes can be modified to include any value of any data type.

At Figure 27 it can be seen the sequence diagram for the `at` and `load_from_implementation` methods of class JSONConfiguration.

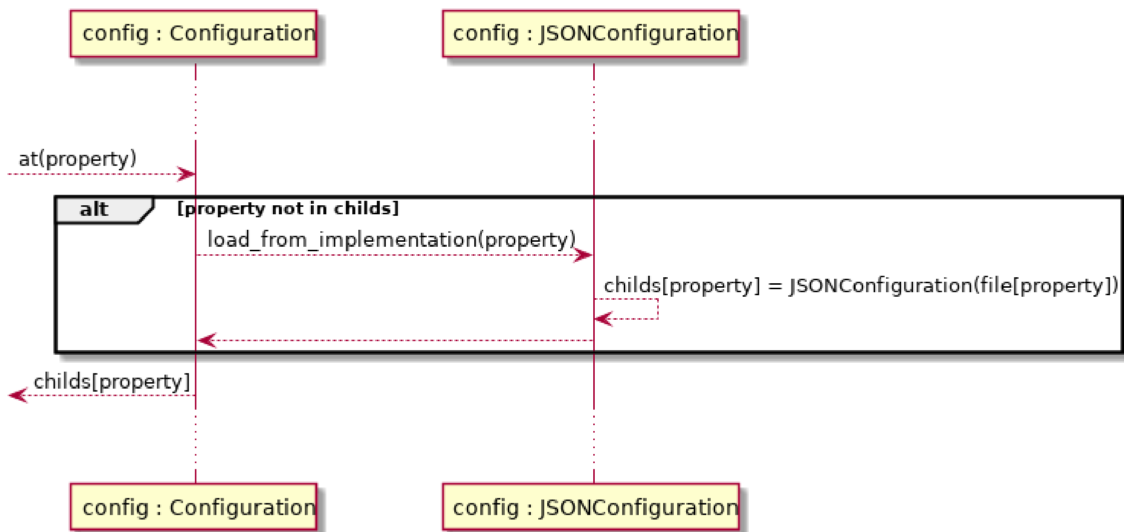


Figure 27 Sequence diagram of the at method from class JSONConfiguration

Note that the `Configuration` class does not provide any way to persist the changes done to a configuration tree back to a configuration file.

To store the node values from the configuration tree, I've implemented the class Any. Although the C++ standard provides a similar class (`std::any`) since C++17, I've chosen to not use it as C++17 is not available for all the platforms I was targeting. The available libstdc++ in the distros for the RoadRunner board, require building the application with GCC version 4.9 which is not fully compatible with C++17.

From the Any class we can get the stored value by calling the `get` method that's templated, so it must be called as, for example, `get<int>` to get an int value. To modify the stored value the `set` method must be called with the new value as a parameter. It is also templated, but in this case the template parameter can be omitted as the C++ compiler will deduct it from the type of the new value.

How this class works is quite interesting. The value is actually hold on the subclass `AnyImpl`. The `AnyImpl` class is generic. The type of the value is template parameter the class receives. For example, an `AnyImpl<bool>` holds a boolean value. But what we don't want to have to specify the template parameter all the time, as this would defeat the purpose of an "any" type. To do this, the `get` and `set` methods from Any are casting the `this` pointer to `AnyImpl` and calling its `get` and `set` methods.

For this reason, the instances of Any cannot be directly created by calling its constructor. Instead, they should be created by calling the static method `create_any` with the value to store as a parameter. This method will create an `AnyImpl` instance and returns a smart pointer to `Any`.

9. Verification and validation

As I explained previously (c.f. 2.2.3. Validation method and software quality assurance), the verification process has consisted of unit and integration testing. Unit testing was automated in a Jenkins instance available at jenking.rt-data.org. After each change it's committed to the GitHub repository, the code is compiled, and the unit tests are executed. Although not part of the verification process, the Jenkins instance was also configured to execute a static code analysis with every commit.

The unit tests are all available under the test directory of the source code. The integration tests are available under the test/integration directory.

The graph in Figure 28 shows the failed and successful builds in Jenkins. For a build to be successful, the compilation must end without errors and all unit tests must execute successfully.

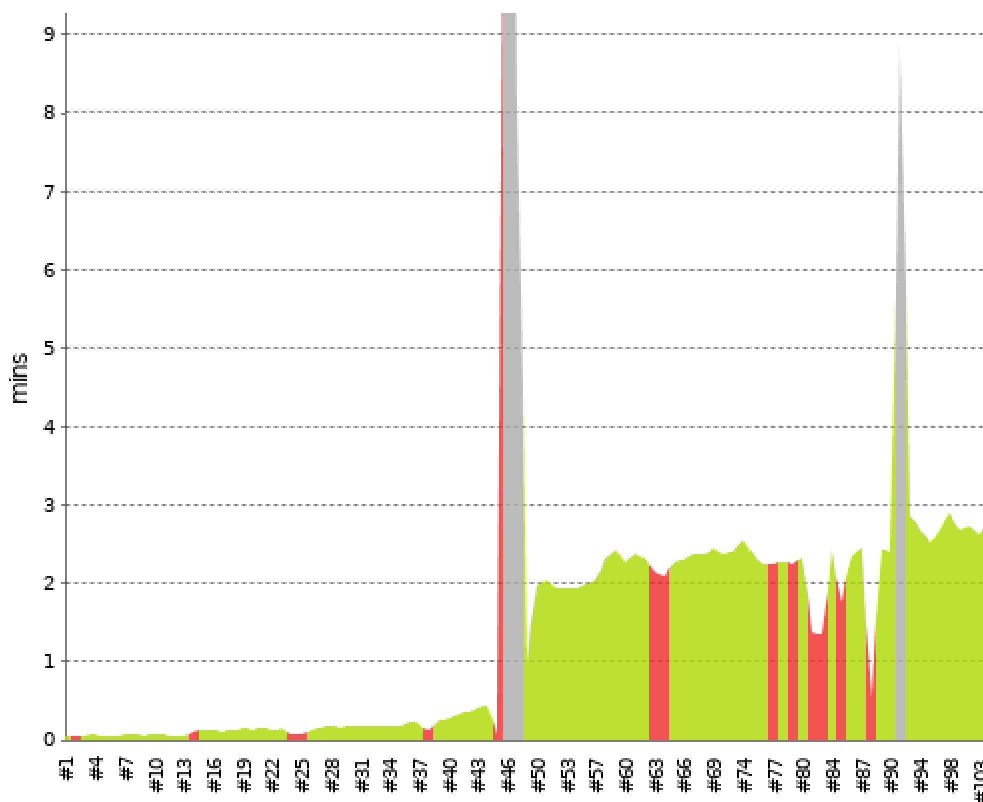


Figure 28 Failed and successful builds

None of the failed builds were due to a failure in the unit tests. The failures were a consequence of either a timeout in the compilation, or due to missing dependencies.

For the validation process, I have performed two acceptance tests, at the end of sprints 5 and 8. A third acceptance test was scheduled for the end of sprint 12, but at the end it was not possible to perform due to the scheduling of Cosmic Research's static tests.

The results for the two acceptance tests follow,

Table 10 Results of the first acceptance test

1ST ACCEPTANCE TEST – SPRINT 5	
FEATURES TESTED	Event management (Broker, Listener, LambdaListener), sensors management (SensorsManager, Sensor), analog sensor, logging, SQLite writer and SQLite serialization.
ISSUES DETECTED	• A segmentation fault produced on the ConcurrentQueue as a result of returning references instead of a copy. • A segmentation fault caused because the lambda that the Broker sends to the ThreadPool was capturing a reference to a shared_ptr. • A race condition on the ThreadPool where a thread was trying to pop a job from the job queue, but the job queue was empty.
RESULTS	Success

Table 11 Results of the second acceptance test

2ND ACCEPTANCE TEST – SPRINT 8	
FEATURES TESTED	HTTP writer, TCP writer, configuration, GPS sensor
ISSUES DETECTED	None
RESULTS	Success

For the first acceptance test, I developed a stub Linux driver that simulates a 10-bit ADC connected to an analog sensor.

For the second acceptance test, I used the gpsfake software from the GPSD project. This software allows to simulate a GPS receiver that feeds the gpsd daemon with the contents of a text file. For this test, I also implemented an HTTP and a TCP server in Python, that received the data written by the HTTPWriter and the TCPWriter.

10. Laws, regulations and licenses

10.1. Laws and regulations

The systems implemented using the software that is being developed in this project might be subject to regulation under ***The Wassenaar Arrangement on Export Controls for Conventional Arms and Dual-Use Goods and Technologies***.

This multilateral agreement is not a treaty, and therefor it's not legally binding. It does also not limit the export of technologies under the control of the arrangement, it is just an agreement for the transparency of the export of the controlled technologies.

According to the list approved on the 5-6 December 2018 Plenary Meeting [20], the developed systems could be under an export control if they are used for flight management and/or flight control systems.

Under the category 7, Navigation and Avionics, section D, subsection 4, it states:

> "Source code" incorporating "development" "technology" specified by 7.E.4.a.2., 7.E.4.a.3., 7.E.4.a.5., 7.E.4.a.6. or 7.E.4.b., for any of the following:
>
> a. Digital flight management systems for "total control of flight";
>
> b. Integrated propulsion and flight control systems;
>
> c. "Fly-by-wire systems" or "fly-by-light systems";
>
> d. Fault-tolerant or self-reconfiguring "active flight control systems";
>
> e. Not used since 2012
>
> f. Air data systems based on surface static data; or g. Three dimensional displays.
>
> Note 7.D.4. does not apply to "source code" associated with common computer elements and utilities (e.g., input signal acquisition, output signal transmission, computer program and data loading, built-in test, task scheduling mechanisms) not providing a specific flight control system function

As stated in the note, this does not apply to systems not used for flight control. For the same reason, it does not apply for rt-data itself. It applies, for example, for the flight control system for Bondar that will make use of rt-data.

Although not technically binding, the Council of the European Union has established the Wassenaar Arrangement as law in all member states of the European Union under the Council Regulation No 428/2009 [21]. Council Regulations are enforceable laws in all member states, including Spain.

10.2. Licenses

As for licensing, rt-data is licensed under the 3rd version of the GNU Lesser General Public License, abbreviated to LGPLv3 or just LGPL. This license allows everyone to use this software for free, even for commercial use.

The license states that any derivate software can be licensed under any license, and it's not mandatory to distribute the source code. But, if there are modifications in the rt-data code, these modifications must be licensed under the LGPL and the source code must be distributed.

Also, the authors and contributors of rt-data are not liable for any damages that the software might cause.

As for rt-data's dependencies, SQLiteCpp is licensed under the MIT license. The MIT license is very permissive and only requires that the copyright and license notices are preserved. The SQLite code is in the public domain. The json library by Niels Lohmann is also under the MIT license. The MIT license is LGPL-compatible.

For the rest of dependencies that are not statically linked or compiled against rt-data, gpsd is under the 3-clause BSD license which is LGPL-compatible. libcurl (and curl) is distributed under the curl license (inspired by the MIT license), which is LGPL-compatible. Finally, g3log is licensed under The Unlicense license, which is also LGPL-compatible.

11. Sustainability and social commitment

In this section I am going to analyse the sustainability of the project from three points of view, the environmental, the social and the economic point of view.

11.1. Environmental sustainability

As previously said the project will be developed in the Cosmic Research's office in Terrassa. I have estimated that in the duration of the project, ~4 months, the electrical consumption will be around 2.400kWh. This is equivalent to 888kg of CO_2[10]. This might seem like a high amount of energy, but we must take into consideration that the office is equipped with air conditioning, a fridge, servers, and a workstation. If we only take into consideration the main tool used to develop the project, my laptop, the electrical consumption would be of around 66,5kWh that is equivalent to 24,61kg of CO_2.

Obviously, the office will not be only used by me, as there are other Cosmic Research's teams using it. So, in fact, a great part of the electrical consumption is shared with other teams. Also, we have put in place strict recycling and power-saving policies.

If we compare my proposed solution to the ones I explained in the state-of-the-art, ROS and EPICS, there's not a big difference in the environmental impact. Both ROS and rt-data are designed to be used in embedded low-power platforms. EPICS on the other hand although it was designed to be used with large industrial systems with high power consumption, it can also be used in low-power platforms for some use cases.

I think we cannot deny that this project will be used for purposes that might not be environmentally sustainable. For example, in Cosmic Research it will be used in the static-tests of Bondar's motor. The propellant that it uses is not, by any means, environmentally sustainable. One of its main components is aluminium, that is known to cause acid rain and can harm plant growth.

Obviously, the contrary is also true. It will also be used for purposes that are environmentally sustainable, as in renewable energy or IoT projects (for example).

---

[10] Calculated using https://www.ceroco2.org/calculadoras/electrico

When using this framework in Bondar's project, the risks are also very high. If used for the parachute control system of Bondar and failed to deploy the chutes at the correct time, the consequences could be fatal. Also, if used for the

11.2. Economical sustainability

A detailed analysis of the project costs can be found on this document. I have estimated that the total cost of the project will be 12.406,77€, including human, hardware and software resources. In Table 8 there is a summary of the budget of the project.

At the end, there have been no deviations from the budget. The total cost of the project has been 11.278,88€. In Table 9 there is a summary of the cost of the project.

If we compare rt-data to the solutions explained in the state-of-the-art, there is not a big difference with the ROS framework. Both can be used in low-cost embedded platforms with low-cost hardware. EPICS on the other hand was designed to be used in (usually) expensive industrial-grade hardware, although it can be used in embedded platforms for some use cases. All three are free (as in freedom) software and can be used for free.

As free software (as in freedom and as in cost) rt-data is not auto-sustainable economically. All present and future support and development will have to be done by volunteers or by an organization like Cosmic Research. This last possibility is not uncommon, companies like Microsoft and Google support open-source projects that do not generate any revenue.

During operation, the only expected cost is the cost of the maintenance of the system. It is expected that rt-data receives updates during its life, which might need the use of both human and hardware resources.

11.3. Social sustainability

This project will not have a direct impact on our quality of life. Instead the projects that will use this framework might have a direct impact on the quality of life of the people. Some possible users are academic and scientific research groups, industries, transport systems, etc.

Unlike ROS or EPICS, rt-data will provide an easy-to-use interface for non-proficient users. This might expand the userbase, that could cause a bigger social impact of rt-data.

Furthermore, as rt-data is free software, anyone can add functionality and even make their own versions of the software. The license selected, the Lesser GNU Public License, allows people to create derived works both open-source and proprietary.

11.4. Sustainability matrix

Taking into consideration all what has been said I've assigned points to the sustainability. I have divided each dimension in three points of view, Project Put into Production (PPP), useful life and risks. The points have been assigned in the following manner,

- PPP. From 0 to 10. 10 means that the project is completely sustainable.
- Useful life. From 0 to 20. 20 means that the project is completely sustainable.
- Risks. From 0 to 10. 10 means that a scenario with a bad outcome has a high probability.

Table 12 Sustainability matrix

	PPP	USEFUL LIFE	RISKS
ENVIRONMENT	7	10	7
ECONOMIC	8	2	5
SOCIAL	10	7	0
TOTAL	25/30	19/60	12/30

The reasoning I've followed is the following,

- **Environment** – **PPP**: The electrical consumption is relatively low, as not much resources are needed for the project. No other environmental impact is expected during the project development.
- **Environment** – **Useful life**: This project is not a final product so we cannot predict its environmental impact. That being said, some of the known applications of this project are known to be environmentally unsustainable.
- **Environment** – **Risks**: The project itself has no almost no direct environmental risk. But, some of its applications might

- **Economic** – **PPP**: The project cost has been estimated and can be assumed by the supporting organization, Cosmic Research.
- **Economic** – **Useful life**: The project will not be economically auto-sustainable as it is free software, as in freedom and as in cost. During operation, only costs related to maintenance are expected.
- **Economic** – **Risks**: The project is financed by Cosmic Research, that is an association with very limited funding. The cost of maintaining the project is low, but in the (not that improbable) case were Cosmic had an even more limited budget, the development of rt-data would be stopped.
- **Social** – **PPP**: On a personal level, this project will give me experience in some fields that I expect will be useful in my career.
- **Social** – **Useful life**: The project does not target to offer new functionalities that do not exist in the present, but to make them more accessible to non-proficient users.
- **Social** – **Risks**: This software nor its known applications won't have any negative impact on the population.

12. Conclusions

As a general conclusion, I am glad to say that I've met all the project objectives and that the software complies with the requirements negotiated with the stakeholders. It has been hard as there was a lot of features to be implemented and it has been very challenging to finish every sprint in just one week.

This project has allowed me to acquire more knowledge in a field that is very interesting for me. I think that the world of IoT and automation has a big potential, and I want to be part of its future.

I'm quite proud of the results I've achieved on this project. It's the first time that I build something this size all by myself, but I think I've managed to deliver a usable software and with enough quality to be used by hobbyists and research groups or companies that are developing proof of concepts of their projects. I've followed standard processes in the industry that I've been taught during the bachelor's degree.

Furthermore, it will also be very helpful for implementing different data acquisition systems for Cosmic Research. From the data acquisition system that will be integrated in the future Bondar rocket, to the system that will serve to test and analyse the performance of Bondar's motor, Nebula.

12.1. Future work

The project had a very tight scheduling and I had to exclude from it some features and works that'd be interesting to work on in the future.

1. User documentation and tutorials. This software is meant to be usable by starters and shall have detailed documentation on how each component work and how to configure it. It would be interesting to have tutorials to help them develop their firsts applications with rt-data.
2. Support for more hardware. Now rt-data only has support for GPS receivers and analog sensors.
3. Support for other OSs. Now rt-data only supports Linux, but embedded systems also use other OSs like VxWorks or RTLinux. It might also be interesting to port it to be usable with FreeRTOS.

4. Support for more persistence methods. Now rt-data only supports persisting data to a file or to an SQLite database. It would be interesting to port it to work with other DBMSs.
5. Being able to retrieve written data. Now rt-data allows to write (persist) data but not to recover it.

13. Bibliography

[1] “Global Internet of Things Market - Global Industry Analysis, Size, Share, Growth, Trends and Forecast 2016 - 2024,” 2017.

[2] P. Pascual Vázquez, “Integración de EPICS en una distribucion Linux para la plataforma Raspberry-Pi,” 2017.

[3] M. Haizad, R. Ibrahim, A. Adnan, T. D. Chung and S. M. Hassan, “Development of low-cost real-time data acquisition system for process automation and control,” in *2016 2nd IEEE International Symposium on Robotics and Manufacturing Automation (ROMA)*, 2016.

[4] D. Selisteanu, M. Roman, D. Sendrescu, E. Petre and B. Popa, “A distributed control system for processes in food industry: Architecture and implementation,” in *2018 19th International Carpathian Control Conference (ICCC)*, 2018.

[5] Argonne National Laboratory, “EPICS Overview,” [Online]. Available: https://epics.anl.gov/EpicsDocumentation/AppDevManuals/AppDevGuide/3.12 BookFiles/chapter1.html. [Accessed 22 February 2019].

[6] ROS, “ROS Core Components,” [Online]. Available: http://www.ros.org/core-components/. [Accessed 22 February 2019].

[7] Glassdoor, “Salaries in the Barcelona area,” [Online]. Available: https://www.glassdoor.com/. [Accessed 10 March 2019].

[8] Red Eléctrica de España, “Términode facturación de energía activa del PVPC,” Red Eléctrica de España, [Online]. Available: https://www.esios.ree.es/es/pvpc. [Accessed 10 March 2019].

[9] GNU, “The GNU/Linux operating system,” [Online]. Available: https://www.gnu.org/. [Accessed 28 04 2019].

[10] A. Rubini and C. Jonathan, Linux Device Drivers, Second ed., O'Reilly, 2001, p. 586.

[11] FreeDesktop, “udev(7) - Linux Programmer's Manual,” [Online]. Available: http://man7.org/linux/man-pages/man7/udev.7.html. [Accessed 29 05 2019].

[12] University of California, “ioctl(2) - Linux Programmer's Manual,” [Online]. Available: http://man7.org/linux/man-pages/man2/ioctl.2.html. [Accessed 29 05 2019].

[13] B. Stroustrup, The C++ programming language, Fourth ed., Addison-Wesley, 2013.

[14] S. Cass and P. Bulusu, “The Top Programming Languages 2018,” IEEE Spectrum, 2018 07 31. [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. [Accessed 29 05 2019].

[15] SQLite, “About SQLite,” [Online]. Available: https://www.sqlite.org/about.html. [Accessed 30 05 2019].

[16] SQLite, “Well-Known Users of SQLite,” [Online]. Available: https://sqlite.org/famous.html. [Accessed 30 05 2019].

[17] Ecma International, “Standard ECMA-404: The JSON Data Interchange Syntax,” 2017.

[18] Kernel.org, “GPIO Sysfs Interface for Userspace,” [Online]. Available: https://www.kernel.org/doc/Documentation/gpio/sysfs.txt. [Accessed 02 06 2019].

[19] Kernel.org, “sched(7) - The Linux Programmer's Manual,” [Online]. Available: http://man7.org/linux/man-pages/man7/sched.7.html. [Accessed 14 06 2019].

[20] Wassenaar Arrangement Secretariat, “List of Dual-Use Goods and Technologies,” 01 12 2018. [Online]. Available: https://www.wassenaar.org/app/uploads/2018/12/WA-DOC-18-PUB-001-Public-Docs-Vol-II-2018-List-of-DU-Goods-and-Technologies-and-Munitions-List-Dec-18.pdf. [Accessed 10 05 2019].

[21] Council of the European Union, Council Regulation No 428/2009 of 5 May 2009, setting up a Community regime for the control of exports, transfer, brokering and transit of dual-use items, 2009.

[22] “Data acquisition,” [Online]. Available: https://en.wikipedia.org/wiki/Data_acquisition. [Accessed 21 February 2019].

[23] “Control system,” [Online]. Available: https://en.wikipedia.org/wiki/Control_system. [Accessed 21 February 2019].