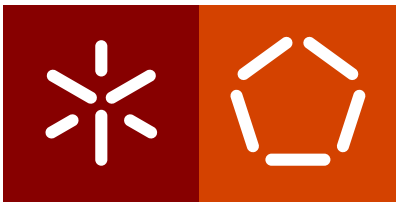

Universidade do Minho
Escola de Engenharia
Departamento de Engenharia Eletrónica
Laboratórios e Práticas Integradas 2

Integrator Project: Final Report

Radio Frequency Camera Assisted Rover (RFCAR)

Group 7

Nuno Rodrigues	A85207
Hugo Carvalho	A85156
Hugo Ferreira	A80665
João Faria	A85632
João de Carvalho	A83564
José Mendes	A85951
José Pires	A50178

Supervised by:
Professor Doutor Vitor Silva

July 16, 2020

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

Notation	Description	Page List
ACL	Asynchronous Connection-Oriented Logical	12, 14
API	Application Programming Interface	14, 20, 132, 133, 164
AT	ATtention	18
BLE	Bluetooth Low—Energy	7
CSI	Camera Serial Interface	130
DL	downlink	17
GGSN	Gateway GPRS Support Node	16, 17
GPRS	General Packet Radio Service	16, 17, 65, 66
GPS	Global Positioning System	206
GSM	Global System for Mobile Communications	16
HCI	Host Controller Interface	7, 14
I/O	Input/Output	20
I2C	Inter-Integrated Circuit	66
IDE	Integrated Development Environment	137
IOT	Internet of Things	7
IP	Internet Protocol	15, 17
IPC	Inter-Process Communication	18, 77
L2CAP	Logical Link Control and Adaptation Protocol	12–14

Notation	Description	Page List
LAN	Local Area Network	15, 16
MAC		
	Machine Address Code	10
	Media Access Control	15
ME	Mobile Equipment	16, 18
MS	Mobile Station	16, 17
NVS	Navigation Virtual Subsystem	65–67, 69, 70, 72, 73, 136, 147, 171, 195, 200
OMT	Object-Modeling Technique	7
OOSE	Object Oriented Software Engineering	7
OS	Operating System	21, 131, 137, 177
OSI	Open Systems Interconnection	15, 16
PC	Personal Computer	147
PDP	Packet Data Protocol	17
PSM	Protocol Service Multiplexers	13
QFD	Quality Function Deployment	24
RFCAR	Radio Frequency Camera Assisted Rover	1, 74, 77, 137, 170, 205
RFCOMM	Radio Frequency Communications	12–14
RVVS	Remote Vision Virtual Subsystem	65–70, 72–74, 132, 136, 164, 171, 177, 186, 187

Notation	Description	Page List
SCO	Synchronous Connection-Oriented	12, 14
SCP	Secure Copy	177
SD	Storage Disk	177
SDP	Service Device Protocol	13
SFTP	Secure File Transfer Protocol	177
SGSN	Serving GPRS Support Node	16, 17
SIG	Bluetooth Special Interest Group	7
SIM	Subscriber Identity Module	16, 18
SMS	Short Message Service	16
SSH	Secure Shell	177
STL	Standard Template Language	88
TCP	Transmission Control Protocol	11, 15, 136
TCP/IP	Transmission Control Protocol/Internet Protocol	8, 136
UDP	User Datagram Protocol	11, 12, 15
UI	User Interface	73, 187, 188
UML	Unified Modeling Language	7, 66, 70, 73
USB	Universal Serial Bus	177, 178
V4L	Video4Linux	132
V4L2	Video4Linux2	132, 133
VM	Virtual Machine	132, 177
WLAN	Wireless local Area Network	15

Symbols

Symbol	Description	Unit
ℓ	wheelbase length of car	m
ω	angular velocity	rad/s
ϕ	direction car is facing	rad
π	ratio of circumference of circle to its diameter	
ψ	steering angle	rad
θ	direction to face	rad

1. Introduction

The present work, within the scope of the curricular unit of Laboratórios e Práticas Integradas II, consists in the project of the development of a product with strong digital basis, namely, a remote controlled vehicle used to assist exploration and maintenance in critical or unaccessible areas to human operators.

In this chapter are presented the project's motivation and goals, the product concept, the project planning and the document organisation.

1.1. Motivation and Goals

The main goal of this project is to develop a remote controlled vehicle with the following characteristics:

1. Remotely operated: the vehicle must be remotely operated to enable its usage in critical or unaccessible areas to human operators;
2. Provide visual feedback to the user: to be a valuable asset in the exploration and maintenance domains, the vehicle must provide visual feedback to the user of its surroundings.
3. Safe: the vehicle must be safe to use and prevent its damage and of its surroundings
4. Robust: the vehicle must be able to sustain harsh environmental conditions and provide redundant mechanisms to avoid control loss.
5. Affordable: so it can be an economically viable product.

1.2. Product concept

The envisioned product consists of a remote controlled vehicle used to assist exploration and maintenance domains, hereby, denominated as Radio Frequency Camera Assisted Rover (RFCAR). To satisfy such requirements, the vehicle must contain a remotely operated camera that provides a live video feed to the user.

Additionally, the vehicle must include an odometric system that assists the driving and avoids unintentional collisions when remote control is compromised, e.g., when connection is lost. The vehicle provides means for exploration and conditions assessment in critical or unaccessible areas to human operators, such as fluid pipelines and other hazardous locations.

1.3. Planning

In Appendix A is illustrated the Gantt chart for the project (Fig. A.1), containing the tasks' descriptions. It should be noted that the project tasks of Analysis, Design, Implementation and Tests are performed in two distinct iterations as corresponding to the Waterfall project methodology.

Due to unpredictable circumstances, limiting the mobility of team staff and goods, the implementation stage will not be done at full extent, but rather at a simulation stage. Thus, to overcome these constraints, the project focus is shifted to the simulation stage, where an extensive framework as to built to model the system operation, test it, and providing valuable feedback for the dependents modules. As an example, the modules previously connected just by an RS232 link, must now include upstream a web module (TCP/IP) — the data is now effectively sent through the internet, and must be unpacked and delivery serially as expected if only the RS232 link was used.

The tasks are described as follows:

- Project Kick-off: in the project kick-off, the group is formed and the tutor is chosen. A brainstorming about conceivable devices takes place, whose viability is then assessed, resulting in the product concept definition (Milestone 0).

- State of the Art: in this stage, the working principle of the device is studied based on similar products and the system components and its characteristics are identified.

- Analysis: In the first stage — Analysis 1 — contains the analysis results of the state of the art. It should yield the specifications document, containing the requisites and restrictions to the project/product, on a quantifiable basis as required to initiate the design; for example, the vehicle's desired speed should be, at maximum, $2\ \mathrm{m/s}$. The second stage — Analysis 2 — contains the analysis of the first iteration of the development cycle.

- Design: it is done in two segments: modules design — where the modules are designed; integration design — where the interconnections between modules is designed. It can be subdivided into conceptual design and solution design.

  - In the conceptual design, several problem solutions are identified, quantifying its relevance for the project through a measuring scale, inserted into an evaluation matrix, for example, QFD.
  - In the solution design, the selected solution is developed. It must include the solution modelling, e.g.:
    * Control system: analytically and using simulation;
    * Transducer design: circuit design and simulation;
    * Power system: power supply, motors actuation and respective circuitry design and simulation;
    * Communications middleware: communication protocols evaluation and selection;
    * Software layers: for all required modules, and considering its interconnections, at distinct levels:
      · front end layer: user interface software, providing a easy and convenient way for the user to control and manage the system.
      · framework layer: software required to emulate/simulate and test the required system behaviour, providing seamless interfaces for the dependents modules
      · back end layer: software running behind the scenes, handling user commands received, system monitoring and control.

- Implementation: product implementation which is done by modular integration. In the first stage, the implementation is done in a prototyping environment — the assisting framework developed, yielding version alpha; in the second stage it must include the coding on the final target modules, yielding prototype beta.

- Tests: modular tests and integrated tests are performed. Tests are generally considered as those performed over any physical component or prototype. Here, it is used as a broader term, to reflect the tests conducted into the system and the several prototypes.

- Functional Verification/Validation: System verification may be performed to validate overall function, but not for quantifiable measurement, due to the latencies involved. Regarding validation, specially for an external agent, thus, it should be limited to user interface validation.

- Delivery: — project closure encompassing:
  1. Final prototype

2. Support documentation: how to replicate, instruction manual.
3. Final report
4. Public presentation

1.4. Report organisation

This report is organised as follows:

- In Chapter 2 lays out the theoretical foundations for project development, namely the project development methodologies and associated tools, and the communications technologies.
- In Chapter 3 are identified the key requirements and constraints the system being developed must meet from the end-user perspective (requirements) and, by defining well-established boundaries within the project resources (time, budget, technologies and know-how), the list of spefications is obtained.
- After defining the product specifications, the solutions space is explored in Chapter 4, providing the rationale for viable solutions and guiding the designer towards a best-compromise solution, yielding the preliminary design and the foreseen tests to the specifications.
- The preliminary design is further refined in Chapter 5 and decomposed into tractable blocks (subsystems) which can be designed independently and assigned to different design teams, allowing the transition to the implementation phase.
- Next, in Chapter 6, the design solution is implemented into the target platforms.
- Then, in Chapter 7, the implementation is tested at the subsystem level (unit testing) and system level (integrated testing), analysing and comparing the attained performance with the expected one.
- After product testing, in Chapter 8, the specifications must be verified and validated by an external agent. subsystem level (unit testing) and system level (integrated testing), analysing and comparing the attained performance with the expected one.
- Chapter 9 gives a summary of this report as well as prospect for future work.
- Lastly, the appendices (see Section 9.2) contain detailed information about project planning and development.

2. Theoretical foundations

In this chapter some background is provided for the main subjects. The fundamental technical concepts are presented as they proved its usefulness along the project, namely the project development methodologies and associated tools, and communications in detail.

2.1. Project methodologies

The methodologies used for the project development are briefly described next.

2.1.1. Waterfall

For the domain-specific design of software the waterfall methodology is used. The waterfall model (fig. 2.1) represents the first effort to conveniently tackle the increasing complexity in the software development process, being credited to Royce, in 1970, the first formal description of the model, even though he did not coin the term [1]. It envisions the optimal method as a linear sequence of phases, starting from requirement elicitation to system testing and product shipment [2] with the process flowing from the top to the bottom, like a cascading waterfall.

In general, the phase sequence is as follows: analysis, design, implementation, verification and maintenance.

1. Firstly, the project requirements are elicited, identifying the key requirements and constraints the system being developed must meet from the end-user perspective, captured in natural language in a product requirements document.

2. In the analysis phase, the developer should convert the application level knowledge, enlisted as requirements, to the solution domain knowledge resulting in analysis models, schema and business rules.

3. In the design phase, a thorough specification is written allowing the transition to the implementation phase, yielding the decomposition in subsystems and the software architecture of the system.

4. In the implementation stage, the system is developed, following the specification, resulting in the source code.

5. Next, after system assembly and integration, a verification phase occurs and system tests are performed, with the systematic discovery and debugging of defects.

6. Lastly, the system becomes a product and, after deployment, the maintenance phase start, during the product life time.

While this cycle occurs, several transitions between multiple phases might happen, since an incomplete specification or new knowledge about the system, might result in the need to rethink the document.

The advantages of the waterfall model are: it is simple and easy to understand and use and the phases do not overlap; they are completed sequentially. However, it presents some drawbacks namely: difficulty to tackle change and high complexity and the high amounts of risk and uncertainty. However, in the present work, due to its simplicity, the waterfall model proves its usefulness and will be used along the project.

As a reference in the sequence of phases and the expected outcomes from each one, it will be used the chain of development activities and their products depicted in fig. 2.2 (withdrawn from [3]).

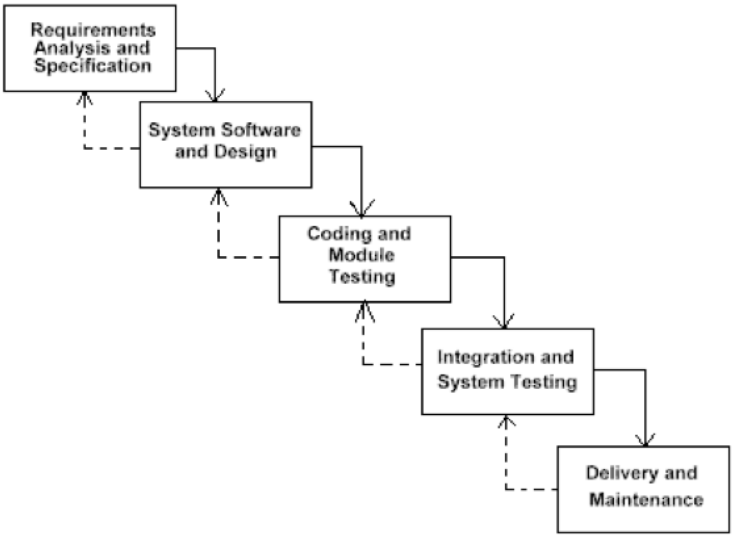

Figure 2.1.: Waterfall model diagram

2.1.2. Unified Modeling Language (UML)

To aid the software development process, a notation is required, to articulate complex ideas succinctly and precisely. The notation chosen was the Unified Modeling Language (UML), as it provides a spectrum of notations for representing different aspects of a system and has been accepted as a standard notation in the software industry [3].

The goal of UML is to provide a standard notation that can be used by all object- oriented methods and to select and integrate the best elements of precursor software notations, namely Object-Modeling Technique (OMT), Booch, and Object Oriented Software Engineering (OOSE) [3]. It provides constructs for a broad range of systems and activities (e.g., distributed systems, analysis, system design, deployment). System development focuses on three different models of the system (fig. 2.2) [3]:

1. **The functional model**: represented in UML with use case diagrams, describes the functionality of the system from the user's point of view.

2. **The object model**: represented in UML with class diagrams, describes the structure of the system in terms of objects, attributes, associations, and operations.

3. **The dynamic model**: represented in UML with interaction diagrams, state-machine diagrams, and activity diagrams, describes the internal behaviour of the system.

2.2. Communications

The communications technologies and the associated tools used for the project development are briefly described next.

2.2.1. Bluetooth

Bluetooth is a ubiquituous radio-frequency technology (2.4 GHz) for wireless communication, generally at short distances. Bluetooth is managed by the Bluetooth Special Interest Group (SIG) and the current version of the standard is 5.0. Starting from version 4.0, also known as Bluetooth Low—Energy (BLE), power consumption was minimised, making it suitable for embedded and Internet of Things (IOT) applications. Bluetooth is a full protocol stack, illustrated in Fig. 2.3, comprising the controller, the host device and the applications interacting with the device. The Host Controller Interface (HCI) layer enables the host device to interface the controller, required for low-level operations such as asynchronous device discovery or reading

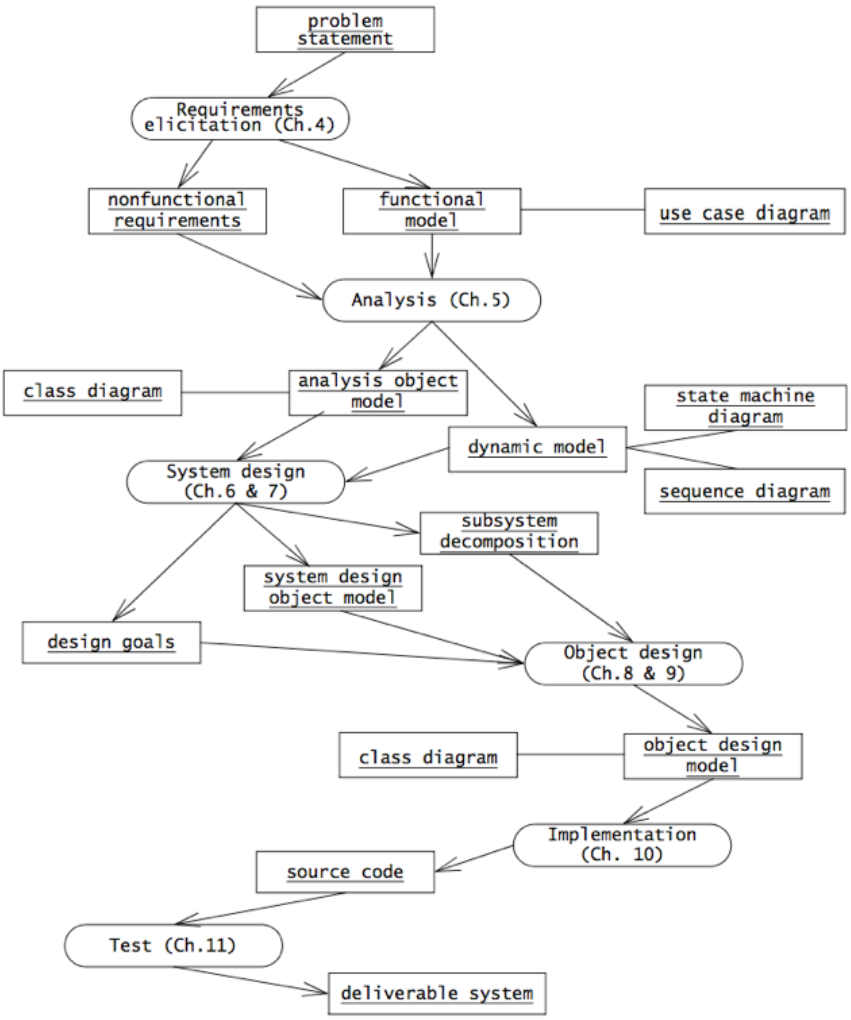

Figure 2.2.: An overview of the object-oriented software engineering development and their products. This diagram depicts only logical dependencies among work products (withdrawn from [3])

radio signal intensity. The host provides profiles to the external applications, easing the interaction betwen device and applications. Conceptually, Bluetooth is very similar to Transmission Control Protocol/Internet Protocol (TCP/IP) stack. Both are part of the network programming class and share the same principles of communication and data exchange between devices. The difference is that Bluetooth was designed for short distance communication, whereas the internet programming does not share this concern. This affects how two devices detect each other initially and how they establish the initial connection. From that moment on, the procedure is similar to the TCP/IP stack (see Fig. 2.4 and Fig. 2.5).

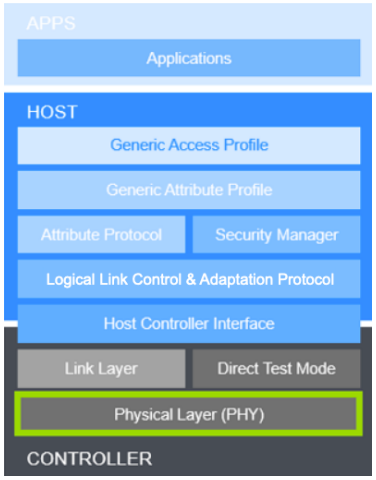

Figure 2.3.: Bluetooth 5.0 protocol stack

2.2.1.1. Establishing a connection

Establishing a connection depends if the device in analysis is trying to establish an outgoing or incoming connection. Basically, for the former case the device sends the first data packet to start the communication, and for the latter the device receives the first data packet. The devices that initiate outgoing connections must choose a target device and a transport protocol, before establishing the connection and exchange data. The devices that initiate incoming connections must select a transport protocol and then listen for incoming connections, before establishing the connection and exchange data [4].

Figures 2.4 and 2.5 illustrate this concept, comparing network, internet and bluetooth programming for outgoing and incoming connections, respectively. For outgoing connections, only the two first steps (choosing a target device, transport protocol and port number) are different; after the connection is established, the process is similar. For incoing connections the procedures are even more similar, with the major difference that port numbers are dynamically assigned for Bluetooth programming. The procedures for programming outgoing and incoming connections are presented next.

Outgoing connection programming procedure:

1. Choose a target device
2. Choose a transport protocol and port number
3. Establish a connection
4. Exchange data
5. Disconnect

Incoming connection programming procedure:

1. Choose a transport protocol and port number
2. Reserve local resources and go into listening mode
3. Wait and accept incoming connections
4. Exchange data
5. Disconnect

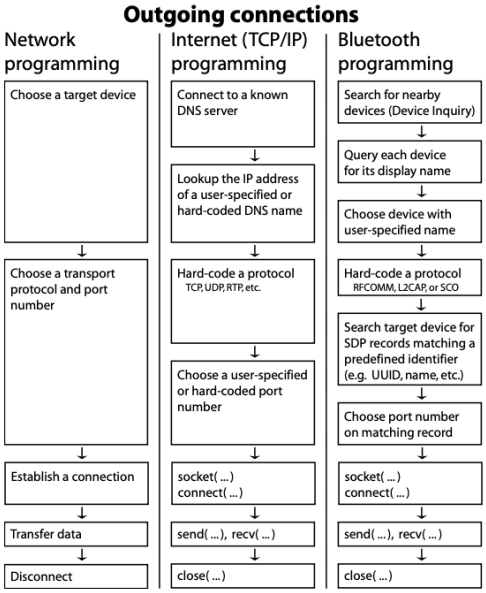

Figure 2.4.: Comparison of Network, Internet and Bluetooth programming for outgoing connections (withdrawn from [4])

2.2.1.2. Selecting a target device

Every Bluetooth chip manufactured has a 48-bit unique address — BT address — identical to the Machine Address Code (MAC) for Ethernet protocol, acting as the basic addressing unit for Bluetooth programming.

The Bluetooth device must know the target device address to communicate. However, the client application does not need to know a priori the target address: the end-user provides a user-friendly name — device name — and the client translate this to the physical address when searching for nearby devices. The device name may not be unique, but the address must be.

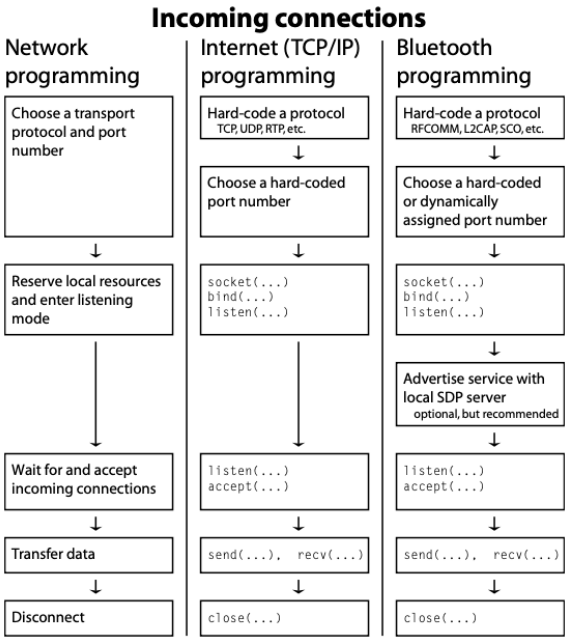

Figure 2.5.: Comparison of Network, Internet and Bluetooth programming for incoming connections (withdrawn from [4])

In the device name lookup process, the device searches for the nearby devices by inquiring each device individually and compiling a list with their addresses, which is generally slow. A Bluetooth device does not announce its presence to other devices; it must start a device discovery process — Device Inquiry — to detect them. In the Device Inquiry process the device broadcast a discovery message and waits for replies. Each reply consists of the physical address of the device and of an integer identifier the class of the device (e.g., smartphone, headset, etc.). More detailed information, such as the device name, may be obtained by contacting each discovered device individually. Additionally, for privacy and power consumption reasons, the device may choose not to respond to device inquiries or connections attempts.

2.2.1.3. Transport protocols

Different applications have different needs, thus the need for different transport protocols. The two factors that distinguish these protocols are guarantees and semantics. The guarantees of a protocol state how hard it tries to deliver a packet sent by the application, yielding: robust protocols, like Transmission Control Protocol (TCP), which ensures that all sent packets are delivered or terminates the connection; best-effort protocols, like User Datagram Protocol (UDP), which makes a reasonable attempt at delivering transmitted packets, but ignores its failure. The semantics of a protocol concerns if it distinguishes between datagrams beginning and end, and can be either packet-based (UDP) or streams-based (TCP).

Bluetooth contains four essential transport protocols, namely, by relevance order:

1. Radio Frequency Communications (RFCOMM): reliable and stream-based protocol, which emulates well serail ports. It allows only 30 open ports per device and it is the most frequently used protocol.

2. Logical Link Control and Adaptation Protocol (L2CAP): packet-based protocol which can be configured for several levels of reliability, imposing delivery order, unlike UDP. It encapsulates the RFCOMM connection.

3. Asynchronous Connection-Oriented Logical (ACL): It is not explicitly used, but it encapsulates L2CAP connections. Two Bluetooth devices may have, at most, one ACL connection between them, which is used to transport all RFCOMM and L2CAP traffic.

4. Synchronous Connection-Oriented (SCO): best-effort and packet-based protocol, which is used exclusively to transmit voice-quality audio at precisely 64 Kb/s.

The summary of Bluetooth transport protocols is illustrated in Fig. 2.6, where is visibly the connection encapsulation. Two Bluetooth devices may have, at most, one ACL and SCO connection between them, whereas the number of RFCOMM and L2CAP active connection is limited only by the number of available ports.

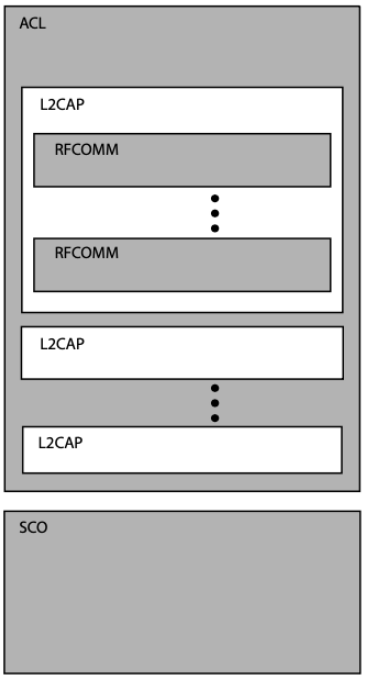

Figure 2.6.: Most relevant Bluetooth transport protocols (withdrawn from [4])

2.2.1.4. Port Numbers

A port is used to enable multiple applications on the same device to use the same transport protocol. Bluetooth uses a slightly different terminology: for L2CAP, ports are called Protocol Service Multiplexers (PSM) (range of 1–32767); for RFCOMM, ports are called channels (range of 1–30).

Some protocols have a set of reserved/well-known ports, intended for specific usage. L2CAP reserves ports 1–1023 for standardized usage (e.g., Service Device Protocol (SDP) uses port 1), whereas RFCOMM does not have reserved ports, given its small number.

2.2.1.5. Service discovery

For a client application to initiate a outgoing connection it must know in which port the server application is listening. If the application is standard, then a reserved port is used. The traditional approach for Internet programming is hardcoding the same port number at both client and server: when the connection is established, the server listens on that part and the client connects to it. However, the static port definition is cumbersome and prevents two server applications from using the same port.

Bluetooth tries to solve this problem by introducing SDP, as illustrated in Fig. 2.7:

1. Each device keeps an SDP server listening on a well-known port.

2. When the server application is executed, it stores a description of itself and a port number in the SDP server on the local device.

3. Then, when a remote cliente application connects for the first time to the device, it provides a description of the service it is searching for, and the SDP server returns a list of all corresponding services with the respective associated port numbers.

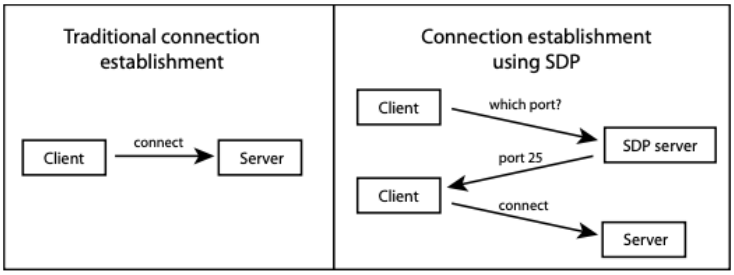

Figure 2.7.: Comparison between traditional connection establishment (internet) and using SDP (Bluetooth) (withdrawn from [4])

2.2.1.6. Host Controller Interface — HCI

The HCI defines how a host (e.g., a computer) interacts and communicates with the local Bluetooth adapter (the controller). All communications between these two agents are encapsulated in HCI packets, of four different kinds:

1. Command: sent from the host to the local adapter to control it, which can be used for starting a device discovery, connecting to a remote device, adjusting the connection parameters, amongst others.

2. Event: generated by the local adapter and sent to the host when an event of interest occurs, for example, device detected, connection established, etc.

3. ACL data: encapsulates data to send ou received from a remote Bluetooth device. In this sense, the HCI is a transport protocol for all transport protocols (ACL,L2CAP, and RFCOMM). When packets arrive to the local adapter, the HCI headers are removed and the pure ACL packet is transmitted through air.

4. SCO

2.2.1.7. Development Stacks for Bluetooth

A development stack refers to a collection of device drivers, development libraries and tools provided to enable software developers to create Bluetooth applications. In most operating systems there is a Bluetooth dominant stack, easing development, as there is a high level of incompatibility between Bluetooth development stacks. Since Bluetooth is a communications technology, the transport protocols supported by each stack are of particular interest. Occasionaly, wrappers around the developed libraries for a stack are created to provide a higher abstraction programming interface for Bluetooth, in languages like Python or Java. The most relevant development stacks and wrappers are depicted in Fig. 2.8.

The Bluetooth development stack selected was **BlueZ**, since its a powerful open source stack, included in all major GNU/Linux distributions and with extensive Application Programming Interface (API) s, supporting a extensive set of protocols which enables to fully explore the Bluetooth local resources. The RFCOMM, L2CAP, and SCO protocols are acessed using the standard sockets interface and the HCI is more conveniently used through the provided wrappers around the several HCI commands and events.

PyBlueZ (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
PyBlueZ (Windows XP)	RFCOMM	L2CAP	SCO	HCI
BlueZ (GNU/Linux)	RFCOMM	L2CAP	SCO	HCI
Microsoft (Windows XP)	RFCOMM	L2CAP	SCO	HCI
Widcomm (Windows XP)	RFCOMM	L2CAP	SCO	HCI
Java – JSR82 (cross-platform)	RFCOMM	L2CAP	SCO	HCI
Series 60 Python (Symbian OS)	RFCOMM	L2CAP	SCO	HCI
Series 60 C++ (Symbian OS)	RFCOMM	L2CAP	SCO	HCI
OS X	RFCOMM	L2CAP	SCO	HCI

Figure 2.8.: Most relevant development Bluetooth stacks and wrappers and its supported protocols (withdrawn from [4])

2.2.2. IEEE 802.11 — Wi-Fi

IEEE 802.11, commonly known as Wi-Fi, is part of the IEEE 802 set of Local Area Network (LAN) protocols, and specifies the set of Media Access Control (MAC) and physical layer protocols for implementing Wireless local Area Network (WLAN) communication in a wide sprectrum of frequencies, ranging from 2.4–60 GHz.

2.2.2.1. TCP/IP

The most commonly used protocols for Internet communications, including Wi-Fi, are TCP and Internet Protocol (IP), usually associated together, being part of the OSI model (Fig. 2.9), which characterises and standardises the communication functions of a telecommunication or computing system, being agnostic to their underlying internal structure and technology.

A computer protocol is a standardised procedure for the exchange and transmission of data between devices, as requested for the application processes. The TCP provides services at the Transport layer, handling the reliable, unduplicated and sequenced delivery of data [5], while the UDP provides data transportation without guaranteed data delivery or acknowledgments. The TCP can be thought of a reliable version of UDP, generalizing. The IP part of the TCP/IP suite, providing services at the Network layer, is used to make origin and destination addresses available to route data across networks.

These protocols are applied in sequence to the user's data to create a frame that can be transmitted from the sending application to the receiving application. The receiver reverses the procedure to obtain the original user's data and pass them to the receiving application [5].

Another interesting fact, due to the technology agnostic aspect of the OSI Model, is that IP and the higher-level protocols may be implemented on several kinds of physical nets.

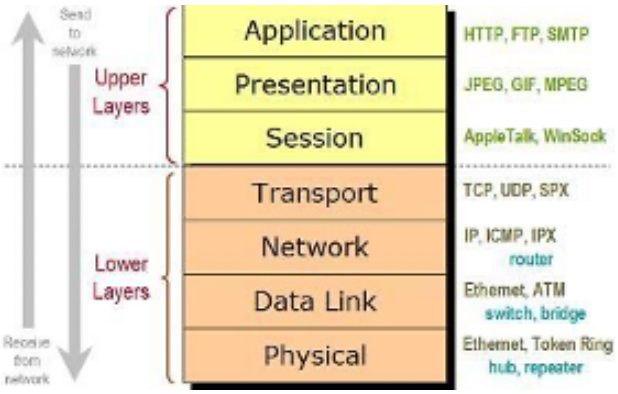

Figure 2.9.: OSI model

2.2.3. General Packet Radio Service — GPRS

General Packet Radio Service (GPRS) is a packet based wireless communication service that offers data rates from 9.05 up to 171.2 Kbps and continuous connection to the Internet for mobile phone and computer users [6]. GPRS is based on Global System for Mobile Communications (GSM) communications and complements existing services such as circuit switched cellular phone connections and the Short Message Service (SMS). However, GPRS is packet oriented (like the Internet), enabling packet data to be sent to or from a mobile device, closing many of the gaps in the GSM standard, namely [6]:

- enable access to company LAN and the Internet;
- provide reasonably high data transmission rates;
- enable the subscriber to be reachable at all times — not only for telephone calls but also for information such as new emails or latest news:
- offer flexible access, either for many subscribers at low data rates or few subscribers at high data rates, optimizing network usage;
- offer low cost access to new services

A GSM network is not able to transmit data in packet switched mode, as required by GPRS. Thus, two additional modules were required: Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN), yielding the GSM/GPRS network depicted in Fig. 2.10. On the user side there is a mobile device known as the Mobile Station (MS), consisting of a Mobile Equipment (ME) and the Subscriber Identity Module (SIM). For GPRS, the ME needs to be equipped with packet transmission capabilities, constituting three different classes of GPRS equipment [6]:

- Class A: equipment that can handle voice calls and packet data transfers at the same time;
- Class B: equipment that can handle voice or packet data traffic (but not simultaneously) and can put a packet transfer on hold to receive a phone call;
- Class C: equipment that can handle both voice and data, but has to disconnect from one mode explicitly in order to enable the other.

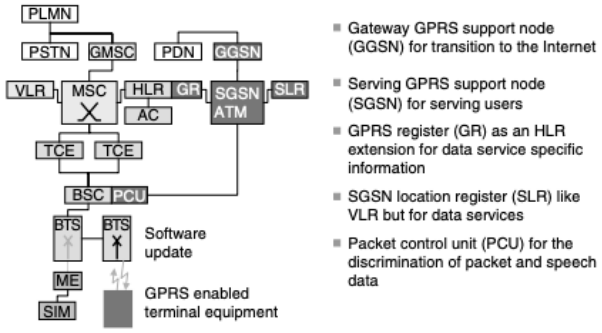

Figure 2.10.: GSM/GPRS network (withdrawn from [6])

The predominant protocol in the world of data networks is the IP, enabling the networking of different network architectures and the standardization of applications. In the mobile network GPRS the subscriber has a logical IP connection with an external data network, representing an actual member of this IP network. Packets can then be transferred between the MS and a server in the IP network, with the GPRS standard describing how they are transmitted on the radio interface and through the whole GPRS network.

Prior to the data transfer three important procedures must be performed [6] (see Fig. 2.11):

1. glsgprs Attach: the MS must be attached in the GPRS network. This is a logical procedure between the MS and the SGSN which takes note of the position, i.e. the 'routing area', of the MS. Storing and updating the position of the MS is particularly important for downlink (DL) transmissions because this information enables the GPRS network to locate the MS.
2. Activation of a Packet Data Protocol (PDP) context: A connection between the MS and the GGSN must be set up, so each node in the GPRS network knows how it has to forward the IP packets of this MS.
3. Data transfer: the path between the MS and the external data network is prepared, so IP packets can be sent through the GPRS network towards the destination address.

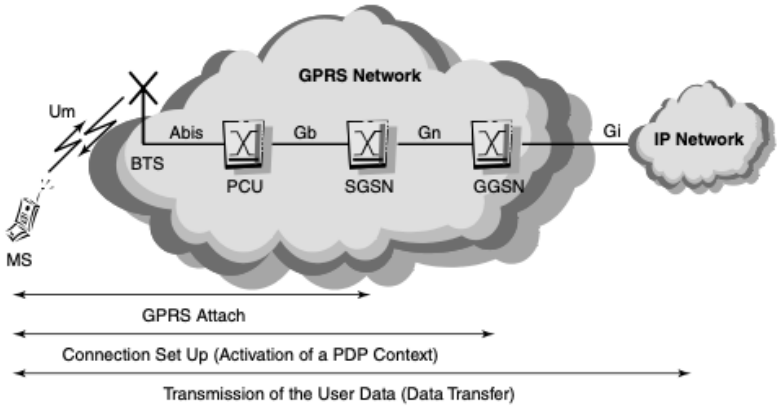

Figure 2.11.: GPRS procedures (withdrawn from [6])

Thus, the user required hardware for implementation of a GSM/GPRS network is the ME, namely, a GSM/GPRS modem and a SIM card with subscription to a mobile operator. A GSM/GPRS modem is generally driven through the RS-232 bus, using the Hayes command set (ATtention (AT) commands) to interface it.

2.2.4. Network programming — sockets

Computer systems implement multiple processes which require an identifier. As such, the IP address is not enough to uniquely identify the origin/destination of data to be transmitted, and the port number is added. This combination of an IP address and port number is sometimes called a network socket [7], allowing data to be delivered to multiple processes in the same machine — same IP address. It is the socket pair (the 4-tuple consisting of the client IP address, client port number, server IP address, and server port number) that specifies the two end points that uniquely identifies each TCP connection in an internet [7]. Bluetooth, as aforementioned, also uses sockets for multiprocess communication, given the device address, transport protocol and port number [4].

In a broader sense, a socket can be described as a method of Inter-Process Communication (IPC) that allows data to be exchanged between applications, either on the same host (computer) or on different hosts connected by a network [8], as a local interface to a system, created by the applications and controlled by the operating system, allowing an application process to simultaneously send and receive messages from other processes.

The Socket API was created in UNIX BSD 4.1 in 1981, with widespread implementation in UNIX BSD 4.2 [8]. It implements the Client-Server paradigm and implement several (standard) functions to access the operating system network resources, through system calls, in Linux [8].

There are two generic ways to use sockets: for outgoing connections — client socket — and for incoming connections — server socket. Fig. 2.12 illustrates the required steps to obtain a connected socket:

1. When a socket is initially created is mostly unuseful.
2. Binding the server socket associates it to an unique network tuple (address and port number), enabling it to be uniquely addressed.
3. When a socket server goes into listening mode, the remote devices can initiate the connection procedure, referring to its unique network tuple.
4. When the socket server accepts a connection, it spawns a new socket which is connected to the remote device, and the endpoints can effectively communicate. The server socket is ready to accept new incomming connections.

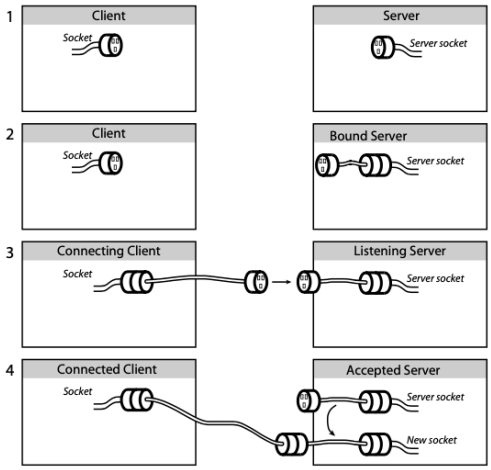

Figure 2.12.: Steps to obtain a connected socket (withdrawn from [4])

2.2.5. Client/server model

The client/server model is the most common form of network architecture used in data communications today [9]. A client is a system or application that request the activity of a service provider system or application, called servers, to accomplish specific tasks. The client/server concept functionally divides the execution of a unit of work between activities initiated by the end user (client) and resource responses (services) to the activity request as a cooperative environment [9]. The client, typically handling user interactions and data exchange/modification in the user's behalf, makes a request for a service, and a server,

often requiring some resource management (synchronization and access to the resource), performs that service, responding to the client requests with either data or status information [10].

An example of a simple client-server model using the Socket API, through system calls, is presented in Fig. 2.13. The operation of sockets can be explained as follows [8]:

- The **socket()** system call creates a new socket, establishing the protocols under which they should communicate. For both client and server to communicate, each of them must create a socket.
- Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place. Two sockets are connected as follows:
  1. One application, assuming the role of server, calls **bind()** to bind the socket to a well-known address, and then calls **listen()** to notify the kernel it is ready to accept incoming connections.
  2. The other application, assuming the role of client, establishes the connection by calling **connect()**, specifying the address of the socket to which the connection is to be made.
  3. The server then accepts the connection using **accept()**. If the **accept()** is performed before the client application calls **connect()**, then the **accept()** blocks.
- Once a connection has been established, data can be transmitted in both directions between the applications (analogous to a bidirectional telephone conversation) until one of them closes the connection using **close()**.
- Communication is performed using the conventional **read()** and **write()** system calls or via a number of socket-specific system calls (such as **send()** and **recv()**) that provide additional functionality. By default, these system calls block if the Input/Output (I/O) operation can't be completed immediately. However, nonblocking I/O is also possible.

2.3. Concurrency

Concurrency is used to refer to things that appear to happen at the same time, but which may occur serially [11], like the case of a multithreaded execution in a single processor system. Two concurrent tasks may start, execute and finish in overlapping instants of time, without the two being executed at the same time. As defined in POSIX, a concurrent execution requires that a function that suspends the calling thread shall not suspend other threads, indefinitely.

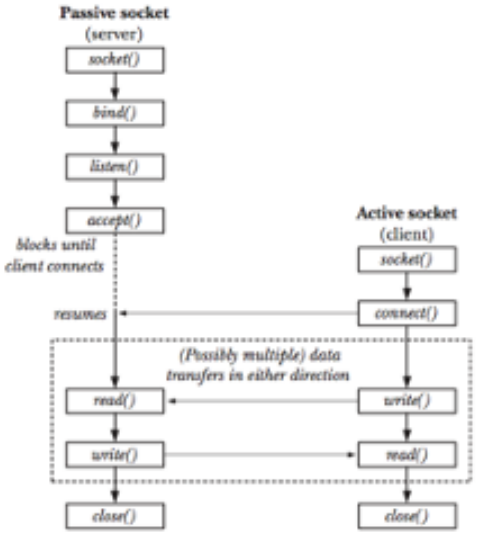

Figure 2.13.: Overview of UNIX system calls with sockets implementing a server/client paradigm (withdrawn from [8])

This concept is different from parallelism. Parallelism refers to the simultaneous execution of tasks, like the one of a multithreaded program in a multiprocessor system. Two parallel tasks are executed at the same time and, as such, they require the execution in exclusivity in independent processors.

Every concurrent system provides three important facilities [11]:

- **Execution Context**: refers to the concurrent entity state. It allows the context switch and it must maintain the entities states, independently.
- **Scheduling**: in a concurrent system, the scheduling decides what context should execute at any given time.
- **Synchronisation**: this allows the management of shared resources between the concurrent execution contexts.

2.4. Android

The mobile Operating System (OS) chosen for this project was Android. This section gives an introduction to some android concepts to take into account in section 5.4.

2.4.1. Activity

An Android activity is a view that allows user interaction. Generally, an app can have multiple activities but always starts with the main activity. Each activity can start another to perform various tasks.

2.4.1.1. Activity Lifecycle

The activities in the system are managed as activity stacks. The activities are organized in a stack, the back stack, in the order by which they were opened, meaning that when a new activity is started it is placed onto the top of the stack. The prior activity won't come into the foreground unless the current activity exits, e.g., when the user presses a back, home or similar return button. Usually, the lifecycle of an android activity has this four key stages:

- **Created**: In this stage the activity is beaing created.
- **Resumed**:The activity is in the foreground thus now visible.
- **Paused**: Another activity is in foreground but this one is still visible.
- **Stopped**: The activity is running in the background and is no longer visible.

To control this lifecycle, one needs to implement the callback methods that refer to each specific stage by overwriting the pre-existing ones. The lifecycle of an activity is depicted in Fig. 2.14.

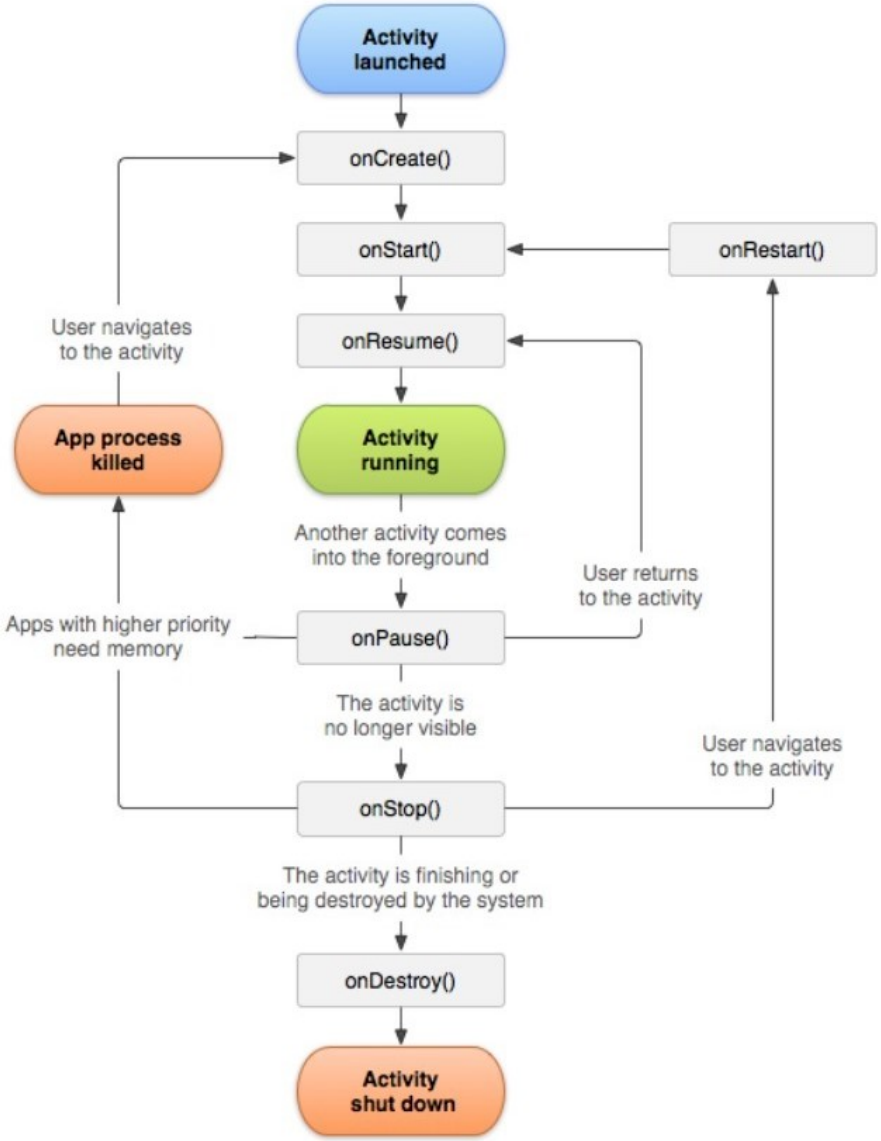

Figure 2.14.: Android activity lifecycle view

3. Requirements Elicitation and Specifications Definition

In this stage the project requirements are elicited, identifying the key requirements and constraints the system being developed must meet from the end-user perspective, captured in natural language in a product requirements document. The end-user perspective is generally abstract, thus requiring a methodic approach to obtain well-defined product requirements, i.e., product specifications. The product specifications are the result of a compromise between end-user requirements and its feasibility within the available project resources (time, budget, and technologies available). As the specifications are well-defined, they serve as design guidelines for the development team and can be tested later on to assess its feasibility and, ultimately, the quality of the product.

3.1. Foreseen specifications

In this section the foreseen product specifications of the system to be developed are provided. Such specifications were obtained through the intersection of customer, functional requirements and project restrictions.

3.1.1. Quality Function Deployment — QFD

The customer requirements are usually abstract and can collide with the functional requirements, compromising the fulfilment of the project. Thus, it raises the need of a methodology which converts abstract requirements into a series of concrete engineering specifications.

An efficient quality assessment methodology is the use of a Quality House (Quality Function Deployment (QFD)). In this method, the desired requirements are laid out as rows and the engineering specifications/restrictions as columns. In the intersections lies a symbol representing the strength (weak, moderate or strong — Figure 3.2) of the relationship requirement-specification. This symbol is one of the many tools that allow the quantification of relations existing between the customer requirements and engineering specifications.

For instance, the 'engine power' specification and the 'fast' requirement have a very strong correlation (9) since the power of the engine is directly responsible for the speed of the car.

Along with the requirements, the importance given to each is also specified, ranging from 1 (lowest importance) to 5 (highest importance) these, along with the number at each intersection, will be used to calculate the importance of each specification and thus assign priorities for the Design Team.

Lastly, the triangle shape (the 'roof' within the house metaphor) serves as another way of measuring relationships, this time between each specification: such is achieved by placing a symbol (ranging from very negative to very positive, see Figure 3.1) in the diagonal intersection of two specifications. I.e., the battery life will have a very negative correlation with the battery temperature, due to the fact that the increase of the temperature will cause a decrease in life time. As such a 'very negative' correlation was placed in the diagonal intersection betwixt 'Battery Life' and 'Battery Temperature'.

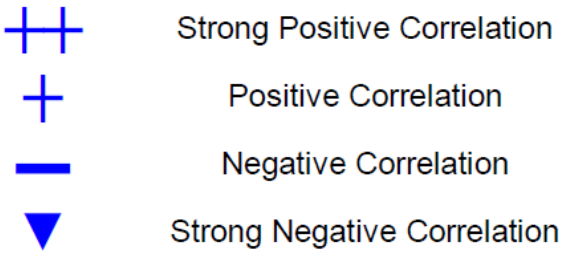

Figure 3.1.: Quality House — Specification Correlation Strength Symbols

Figure 3.3 shows the 'Quality House' for the RF CAR containing:

- **Customer Requirements**: Vehicle Integrity; Obstacle Avoidance; Reliable Feedback; Fast Response; Fast; Budget Friendly; Low Consumption; Small.
- **Functional Requirements or Restrictions**: Autonomy; Battery Temperature; Minimum Distance to Obstacle; Maximum Velocity; Motor Expectancy; Cost of Production; Motor Power; Ramp-Up Speed Time; Frame Rate; Camera Range; Resolution; Communication Range; Communication Speed; Dimensions; Mass.
- **Intersection Values** (referencing the strength of the requirement-specification correlation) — see Figure 3.2.
- **Analytical Data**, depicting, in a quantifiable manner, the aims of the project and the relevance of each entity:
  - Target or Limit Value: The metrics the design team will be based on, white spaces are left for either further discussion and refinement.

- Difficulty: Allows a subjective input to be added so that 'importance' can be changed to balance unforeseen circumstances.
- Importance and Relative Weight: The main conclusion for which the QFD was used, it assigns the priorities for the design team in an objective manner.

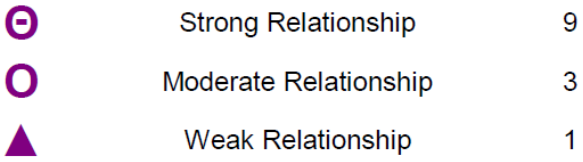

Figure 3.2.: Quality House – Relationship Strength Symbols

With the QFD, the prioritized ranks and specification targets were obtained and diffused within the Design Team with a straightforward guideline. For instance, the low cost requirement should be prioritized over all other specifications, followed by the maximum speed, Ramp-Up Speed Time and so on. On the other hand, the engine expectancy is of little to no consequence (note that the importance added up to a mere 3%), followed by the camera-related specifications. This could be regarded as a point of discussion, which should be prioritized? The functionality of the car or the the feedback provided by the camera?

With the last point in mind, the QFD has the advantage of promoting further discussion, simply by changing the importance of a requirement the priority ranking will change, ergo the priorities can be altered, easily and efficiently, if deemed appropriate.

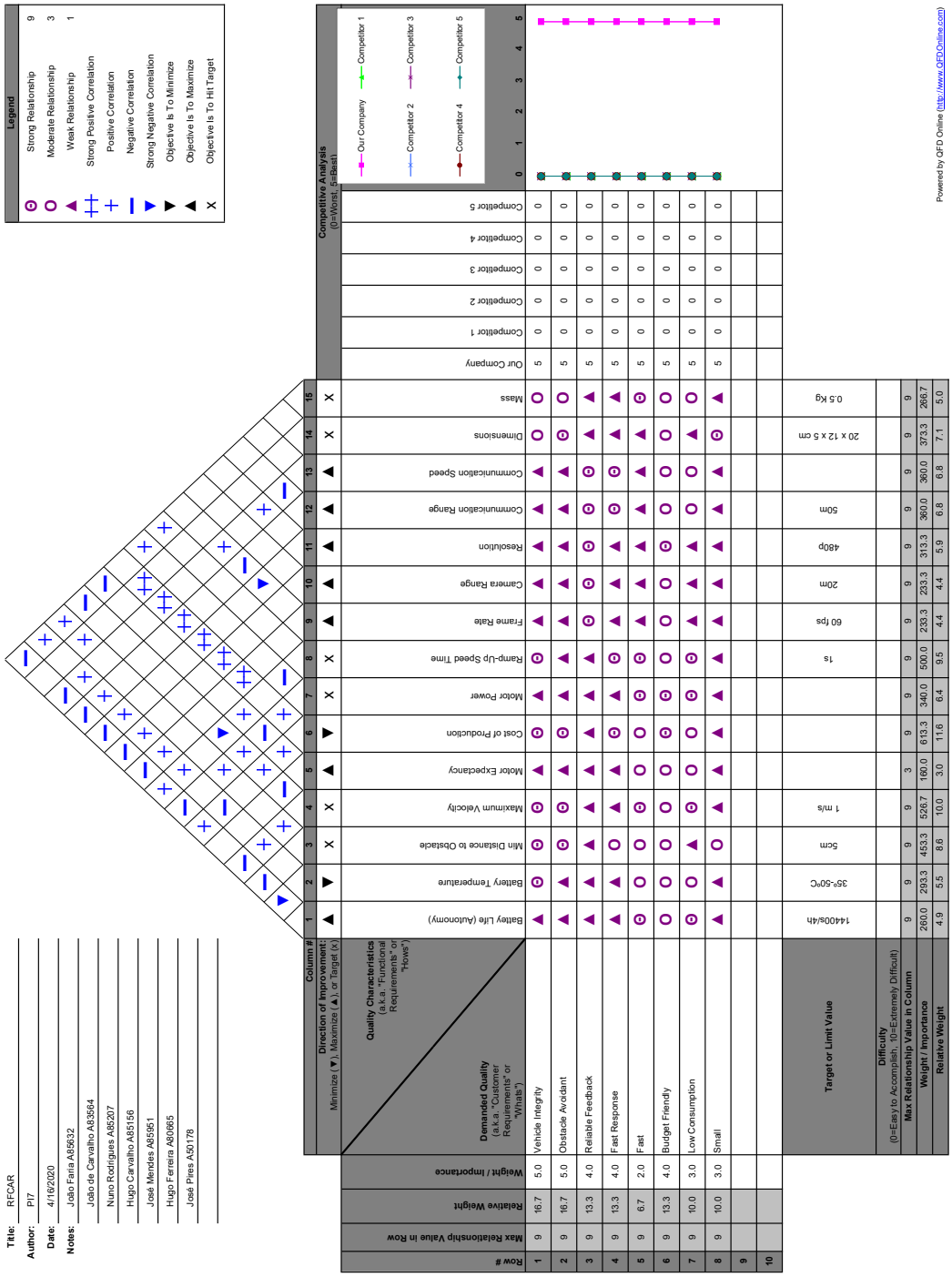

Figure 3.3.: Project Study – RFCar Quality House

3.1.2. Vehicle Autonomy

The vehicle is operated in wireless mode, thus, a portable power source must be included. The autonomy refers to the vehicle operating hours since the battery is fully charged and safely discharged and should be observed for the following scenarios:

- No load and vehicle operating at maximum speed;
- No load and vehicle operating at mean speed;
- Maximum load and vehicle operating at maximum speed;
- Maximum load and vehicle operating at mean speed.

3.1.3. Speed

The vehicle must be operated within a safe range of speed, while also not increasing excessively the power consumption. Thus, these speed boundaries should be tested in the absence of an external load and in the presence of the maximum load.

3.1.4. Safety

Vehicle self integrity protection is a requirement in product design, especially considering the vehicle is to be remotely operated. The safety in the operation can be analysed in two ways, and considers the preservation of people and goods. For the former, it is important to assure safe interaction as well as user operation — the vehicle may encounter several obstacles along its path, but it must not inflict any damage. For the latter, the vehicle under operating conditions must not inflict any damage to goods. Thus, in the presence of conflicting user commands violating the safety of people and goods, the local system should override them, taking corrective measures to prevent it. The same holds true if the communication between user and system is lost.

3.1.5. Image acquisition

The vehicle is equipped with a camera to assist in its navigation, thus, requiring it to be fed to the user's platform appropriately. To do so, several functionalities details need to be addressed efficiently. It was selected the most relevant three and these include the frame rate, the resolution and the image range.

3.1.5.1. Frame rate

Frame rate refers to the frequency at which independent still images appear on the screen. A better image motion is the result of a higher frame rate but the processing overhead increases as well, so a compromise must be achieved between the quality of the image and the increased processing overhead required. The minimum frame rate defined must be such that allows a clear view of the navigation.

3.1.5.2. Range

How far can the camera capture images without being distorted or unseen by the user. The range must be such that allows the user to see the obstacles when the car is heading to them and provide enough time to change the direction.

3.1.5.3. Resolution

The amount of detail that the camera can capture. It is measured in pixels. The quality of the acquired image is proportional to the number of pixels but a increased resolution requires a greater data transfer and processing overhead, thus, a compromise must be achieved. The minimum resolution must be such that provides the least amount of information required for the user.

3.1.6. Communication

For the implementation of the communication, several stages must be considered: Reliability, redundancy and communication range.

3.1.6.1. Reliability

A communication is reliable if it guarantees measures to deliver the data conveyed in the communication link. As reliability imposes these measures, it also increases memory footprint, which must be considered depending on the case. For the devised product, an user command must be acknowledged to be processed, otherwise, the user must be informed; on the other hand, loosing frames from the video feed is not so critical — user can still observe conveniently the eld of vision if the frame rate is within acceptable boundaries.

3.1.6.2. Redundancy

The communication protocols are not flawless and the car relies on them to be controlled. If the communication is lost, the car cannot be controlled. A possible solution for this issue is using redundancy in the communication protocols (e.g Wi-Fi and GPRS), so if one protocol fails, the car will still be controlled using the other.

3.1.6.3. Range

The communication protocols have a limited range of operation, and, as such, regarding the environment on which the car is used the range can be changed. The range established the maximum distance allowed between user and system for communication purposes.

3.1.7. Responsiveness

The movement of the car will be determined by the tilt movement of the smartphone. Sensibility refers to the responsiveness of the car on the minimum smartphone tilt movement. The sensibility must be in an range of values in which small unintentional movements will not be enough to change the state of the car and it does not take big smartphone tilts for the car to move.

3.1.8. Closed loop error

The speed, direction and safe distance to avoid collisions must be continuously monitored to ensure proper vehicle operation. The closed loop error must then be checked mainly in three situations as a response to an user command:

- speed: the user issued an command with a given mean speed, which should be compared with the steady-state mean speed of the vehicle.
- direction: the user issued an command with a given direction, which should be compared to the vehicle direction.
- safe distance to avoid collisions: the user issued an command with a given direction and speed which can cause it to crash. The local control must influence, to prevent collision, and the final distance to the obstacles must be assessed and compared to the defined one.

3.1.9. Summary

Table 3.1 lists the foreseen product specifications.

Table 3.1.: Specifications

	Values	Explanation
Autonomy	4 h	Time interval between battery fully charged and safely discharged
Speed Range	0.1 to 1 m/s	Speed at which the car can operate
Frame Rate	60 fps	Frequency at which independent still images appear on the screen
Camera Range	20 m	How far can the camera capture images without loosing resolution
Camera resolution	480p	Amount of detail that the camera can capture
Communication Range	50 m	Maximum distance between the car and the smarphone without losing connection
Speed Error	5 %	Maximum difference between desired and real speed
Direction Error	5%	Maximum difference between desired and real direction
Distance Error	5 %	Maximum difference between desired and real distance to the obstacle
Dimensions	20x12x5 cm	Dimensions of the car
Weight	0.5 kg	Weight of the car

4. Analysis

After defining the product specifications, it is possible to start exploring the solutions space within the project's scope, providing the rationale for viable solutions and guiding the designer towards a best-compromise solution. In this chapter are presented the preliminary design and the foreseen tests to the specifications.

4.1. Initial design

Following an analysis family tree of the products(remote controlled cars), the state of the art and the QFD matrix in fig. 3.3, an initial design of the product itself can be produced (fig. 4.1). The selected approach was top-down, in the sense that the requirements and specifications were addressed and that resulted in a general diagram of the product concept. Some macro-level decisions were made in this stage to narrow the solutions pool of the problems, as follows:

- The car itself should be battery-powered, as it is a free-moving object that is intended to work in environments where trailing cables could interfere with its regular movement.
- The device used to control the car should ideally be one already owned by the user, with an integrated screen (e.g. smartphone), as it would make it more affordable and have a more straightforward interface.
- The protocols for communication between the controlling device and the Rover should be chosen from within the pool of those readily available to smartphones (e.g. Wi-Fi, GPRS, Bluetooth) to keep the price of the overall product down and make it as practical as possible.
- The control and communication unit for the Rover should be divided into two modules: one which can measure and process sensor inputs and control the actuators in real-time, as well as communicate with the user-operated device through a low-latency connection. And another one which can interface directly with the camera module and manage data transmission to the user at the applicational level of the TCP/IP protocol stack, with enough throughput for the specified video resolution and frame

rate. The latter should also exchange sensor reading values and commands with the user-controlled device, introducing redundancy to the controls and thus allowing for more reliable operation.

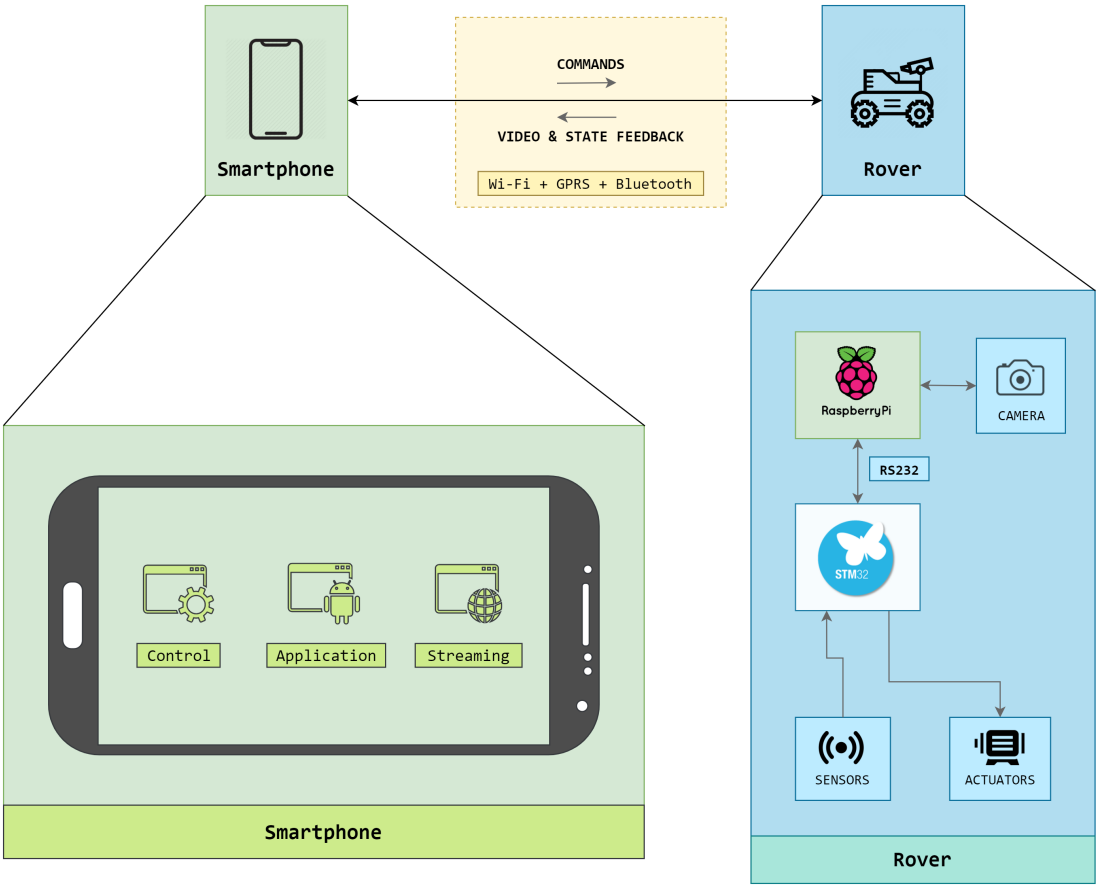

Figure 4.1.: Initial design: Block diagram view

Thus, summarising, the initial design yields the system illustrated in fig. 4.1, comprised of:

- **Raspberry Pi**: Interfaces with the camera directly, streaming that information to the smartphone. It also receives user commands and sends sensorial information it receives from the STM32 module back to the Smartphone, all through redundant Wi-Fi and GPRS connections;
- **STM32**: Receives user commands and sends back sensorial information through a Bluetooth connection, information it also uses to control the actuators;
- **Actuators**: DC Motors that control the movement of the Rover and headlights for nocturnal or low light conditions;
- **Sensors**: Odometric sensors that support the detection of obstacles and luminosity sensors;

- **Camera**: Device connected to the Raspberry Pi that allows the live stream of the car´s surrounding environment;
- **Smartphone**: Grant visual feedback from the live feed of the camera also allowing the user to control the movement of the vehicle intuitively;

For simulation purposes and in conformance with the extraordinary conditions imposed by the recently enacted confinement measures, the need rose to create a virtual environment to simulate the various subsystems of the Rover. This solution allows for integrated testing without the need to deploy it to the hardware, as illustrated in fig. 4.2.

The first of said subsystems is the Physical Environment Virtual Subsystem, which simulates the Rover and the physical environment, also receiving actuator inputs from the Navigation Virtual Subsystem, for which it generates the sensor readings. The latter also exchanges that information, as well as the status of the Rover and the commands from the user with the Smartphone module through a Bluetooth connection, which separates it from the non-simulated environment.

The Remote Vision Virtual Subsystem interfaces with the computer's Web Camera, which is meant to simulate the onboard camera of the Rover. Moreover, it can also communicate through Wi-Fi and GPRS, thus ensuring that communication is kept despite any failure from the Navigation Virtual Subsystem, as well as having shared access to the sensor reading values for monitoring of certain key parameters, depending on their refresh rate.

The RS232 connection between both controlling subsystems ensures proper synchronism and cooperation between them.

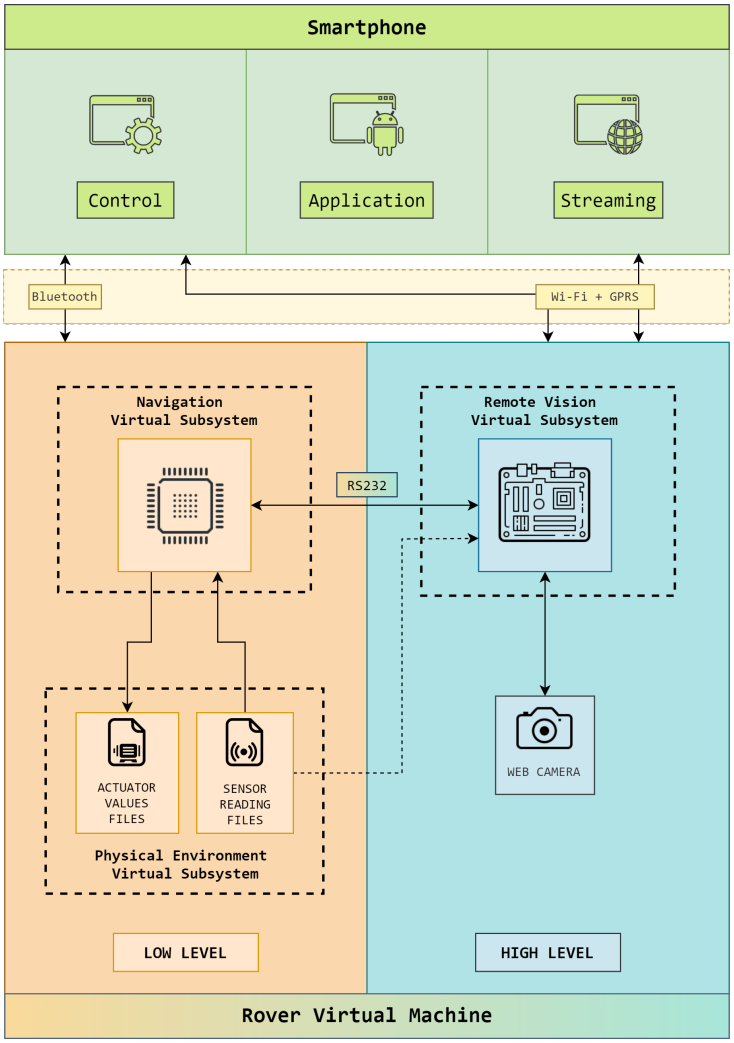

Figure 4.2.: Initial design: Virtual environment block diagram view

4.2. Foreseen specifications tests

The functional testing is generally regarded as those performed over any physical component or prototype. Here, however, a broader sense is used, to reflect the tests conducted into the system and the several prototypes, under the abnormal present circumstances. Moreover, as indicated in the design, the current development strategy encompasses the virtualization of all hardware components, enclosed in a single virtual environment.

Thus, it does not make sense to perform hardware related tests such as velocity measurements, autonomy, safety, etc. As such, the focus is shifted towards software and control verification, encompassing the

following tests: functionality, image acquisition, communication, and control algorithms correctness.

The tests are divided into verification and validation tests.

4.2.1. Verification tests

The verification tests are tests performed internally by the design team to check the compliance of the foreseen specifications. These tests are done after the prototype alpha is concluded.

4.2.1.1. Functionality

The remotely operated vehicle is composed of several modules distributed along several different platforms, some of which distanced from each other. In doing so, the proposed sets of functionalities should be tested in the integrated system, by tracking and analysing the user commands issued along the way until it finally reaches the vehicle (in the virtual environment), assessing if it is correctly processed. For example, if the user issues the vehicle to move to a given place (via smartphone interaction), the message sent to the vehicle must be signalled in each endpoint hit, and the vehicle should move to that place, symbolically detected by the modification of its virtual coordinates.

4.2.1.2. Image acquisition

The vehicle is equipped with remote vision to assist the user in its navigation, thus, requiring the following variables to be tested: frame rate, range, and resolution. In the current scenario, the virtual environment should provide access to a integrated camera, being fairly common nowadays in every modular component, thus, enabling easy testing.

4.2.1.2.1. Frame rate

To test frame rate, the user screen must be updated with the number of frames received from the camera per second and checked against the defined boundaries.

4.2.1.2.2. Range

To test camera's range, an object must be captured at increasing distances, until the object resolution fades or is unclear.

4.2.1.2.3. Resolution

The minimum resolution should be tested as providing the least amount of information required for the user, while minimizing data transfer and processing overhead.

4.2.1.3. Communication

The communication tests are performed in compliance to the specifications provided in Section 3.1.6, namely reliability and redundancy.

4.2.1.3.1. Reliability

To test communication reliability, the most critical communication link is chosen, namely the wireless communication between user's platform (smartphone) and vehicle's platform (virtual environment). Then, the communication link and protocol selected are tested by monitoring the ratio of dropped packets versus total packets, using an appropriate tool on both directions for transmission and reception.

4.2.1.3.2. Redundancy

To test communication redundancy, one communication channel should be turned off, verifying if the communication between nodes is still possible through another communication channel. For example, in the communication between host (user's platform) and remote (virtual environment) guest systems, the priority communication is performed between host and high-level subsystem. However, if this is turned off, the host must also be able to communicate with the remote system via low-level subsystem.

4.2.1.4. Correctness of the control algorithms

As previously mentioned, the speed and position must be continuously monitored to ensure proper vehicle operation. Under the current circumstances, where the sensors and actuators are virtualized, the control loops can be externally stimulated through input files containing the relevant data. Then, the behaviour of the system can be analysed and verified for some cornercase situations, assessing the control algorithms correctness.

4.2.2. Validation tests

The validation tests should be performed by the client using the product's manual, so it is expected that a user without prior experience with the product should be able to use it correctly and safely. On the current circumstances, validation tests should be limited to user interface validation.

For this purpose, an external agent will be provided with the software application and the respective installation and usage manuals, and the feedback will be collected and processed to further improve the product.

5. Design

In the design phase, the solution is developed in the various domains and a thorough specification is written, paving the way for implementation. In this chapter is presented the design for the various domains and subsystems identified.

5.1. Navigation Virtual Subsystem

The Navigation subsystem hosts the core of the system functionality-wise, which is the control routine. This means that it should strive to not only make accurate readings and calculations but also be as efficient as possible in managing those processes in order to introduce very little delay and meet timing requirements.

To meet these requirements as best as possible it should be capable of :

- Gathering information from the physical domain at equally distant instants kT_s and output an electrical representation of the command variable at equally distant instants kT_o;
- Acquiring commands from the Smartphone and Remote Vision Subsystems, identifying matches that will allow it to validate those commands and feeding them into the control rule in a useful format;
- Providing real-time feedback to the user about its status.

The first task should be idealizing the control system itself, understanding what inputs are needed to control the machine and then how it could be used to manipulate the wheels of the car. After that, the rest of system should be designed to fit the needs of the control rules and algorithms and use them to react as fast and consistently as possible within its own constraints and those of the other subsystems.

5.1.1. Control

The objective of the control module is to be capable of, with an input reference velocity and steering angle, outputting the appropriate command variables to achieve said references. The first step in its design is to

conceptualize the car's model; since the interest lies in the motion of the car, the kinematic model will be the one to be studied.

5.1.1.1. Conception Of Car Model

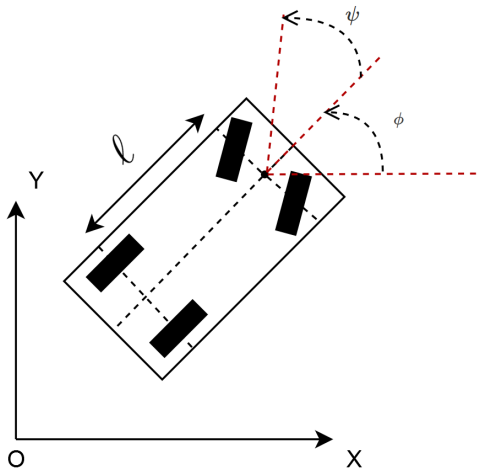

Figure 5.1.: Kinematic Model of Car

Considering the kinematic model of a four-wheel vehicle with a wheelbase length ℓ, a linear velocity $v(t)$ and an angular velocity $\omega(t)$ the following situation will occur: since the rear wheels will remain in the same position no matter where the car is facing towards, they will be facing whatever orientation, ϕ, the car is facing, however, in order for the car to be capable of moving into wherever the user tells it to, the front wheels must turn, ergo a steering angle ψ, must be considered. The resulting direction the car will be going in is $\phi + \psi$. With these considerations, a desired angle of tilt θ and considering that $\phi = 0$, in other words, that whatever direction the car is told to face, it will be relative to its current direction, the following model is obtained:

$$\dot{x} = v(t)cos(\psi) \tag{5.1}$$

$$\dot{y} = v(t)sin(\psi) \tag{5.2}$$

$$\dot{\psi} = \omega(t) = \frac{v(t)}{\ell}\theta \tag{5.3}$$

However this model is not enough for simulation purposes. The simulation has the objective of granting the designers clear ideas of the response of the systems towards given inputs, therefore, for implementation purposes, the simulation must give feedback on the position of the car, its heading and the linear velocity

of the right rear wheel (v_r), and the left rear wheel (v_l) therefore the model will have to changed with these details in mind, which can be achieved considering that:

$$\omega(t) = \frac{v_r(t) - v_l(t)}{\ell} \tag{5.4}$$

$$v(t) = \frac{1}{2}(v_r(t) + v_l(t)) \tag{5.5}$$

Solving the system above for v_r and v_l:

$$v_r(t) = v(t) + \frac{\omega(t)\ell}{2} \tag{5.6}$$

$$v_l(t) = v(t) - \frac{\omega(t)\ell}{2} \tag{5.7}$$

Returning the model in order to $\dot{x}$ and $\dot{y}$:

$$\dot{x} = v(t)cos(\psi) - \frac{\ell}{2}\omega(t)sin(\psi) \tag{5.8}$$

$$\dot{y} = v(t)sin(\psi) + \frac{\ell}{2}\omega(t)cos(\psi) \tag{5.9}$$

$$\dot{\psi} = \omega(t) = \frac{v(t)}{\ell}\theta \tag{5.10}$$

5.1.1.2. Simulation Model

The mathematical model determined in Section 5.1.1.1 was simulated in the Matlab/Simulink environment. Converting the mathematical model into a simulink subsystem yields (Fig. 5.2): The objective of the control module is making sure that the linear velocities of the right and left wheels achieve the desired values, consequently controlling the linear velocity of the rover and its steering angle. Therefore that is what the simulation model takes into account, two controllers, one for each, wheel will make sure that both the steering angle and the linear velocity reach the desired values. This results in the schematic obtained in Fig. 5.3.

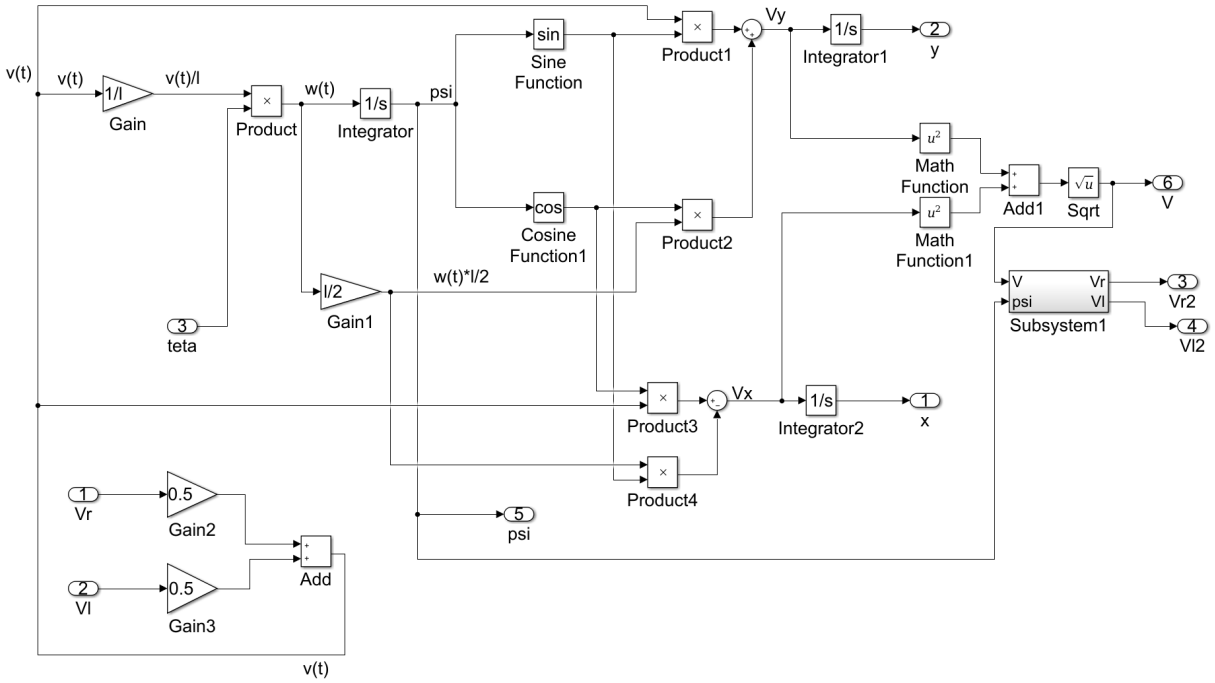

Figure 5.2.: Car Model in a Simulink Subsystem

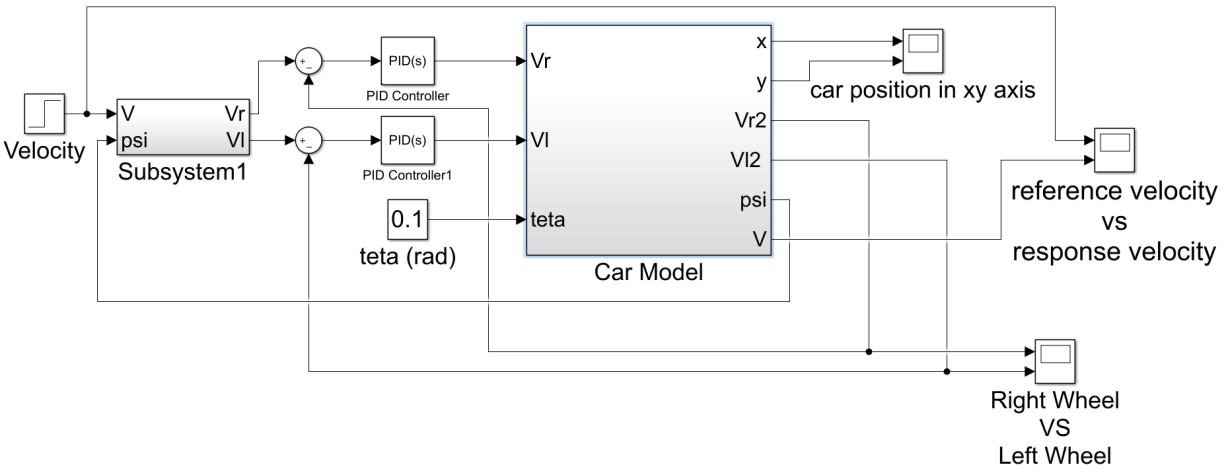

Figure 5.3.: Simulation Schematic

5.1.1.3. Discrete PID

To control the system, a PID controller was used for the reason that it is the most used of all controllers in this type of cases and also leads to better system behavior because of the following statements : the proportional action defines the velocity of approach to the value in steady state; the integral action reduces the error in steady state by integrating the error; the derivative action anticipates the response, deriving the error, allowing to reduce the oscillatory behavior of the system response and its integration in the controller should always be analysed, principally in systems with a lot of noise, to avoid instability in the system.
The PID controller can be implemented either in analog or digital, however, the implementation it is usually digital using a system with a microprocessor associated. This method can also improve the performance by allowing control of actuator saturation(wind-up), even in systems with a lot of noise in the measured variables. Thus, the digital control system performs the sampling of inputs of interest, calculates the value of the control variable and then, if necessary, convert it to an analog value. The sampling introduces a delay in the system and keep the entry value between consecutive samples, leading to the concept of retainer. Normally, the Zero Order Holder (ZOH) is selected to model this effect in the control.

The analog PID controller equation is given by Eq. 5.11:

$$C(t) = C_{est} + K_c\left(e(t) + \frac{1}{\tau_i}\int e(t')dt' + \tau_d\frac{de(t)}{dt}\right) \tag{5.11}$$

- C(t): control action
- C_{est}: control action in steady state;
- K_c: proportional controller gain;
- τ_i: integral time constant;
- τ_d: derivative time constant;
- $r(t) = y_r(t) - y(t)$: error, given by the difference between reference and the output of the system.

The discrete PID controller equation is obtained by discretizing the Eq. 5.11 term to term in moments n and $n-1$, resulting in Eq. 5.12, called position algorithm:

$$C[n] = C_{est} + K_c\left(e[n] + \frac{T}{\tau_i}\sum_{i=0}^{n} e[i] + \tau_d\frac{e[n]-e[n-1]}{T}\right) \tag{5.12}$$

- C[n]: control action
- C_{est}: control action in steady state;
- K_c: proportional controller gain;
- τ_i: integral time constant [s];
- τ_d: derivative time constant [s];
- $e[n] = y_r[n] - y[n]$: error, given by the difference between reference and the output of the system for instant n.
- $e[n-1]$: error for instant n-1.
- T: sampling period [s]

Determining the equivalences for the earnings of the controller in the most conventional one:

$$K_p = K_c;\ K_i = K_c\frac{T}{\tau_i};\ K_d = K_c\frac{\tau_d}{T} \tag{5.13}$$

There are other algorithms for PID control, namely the speed algorithm and the modified ones (position and speed). The speed algorithm calculates the variation of the controller output signal in relation to the immediately previous moment, presenting advantages over position algorithm as it does not need initialization and protects against eventual saturation of the controller (wind-up) and against computational failures. The modified algorithms aim to mitigate the derivative jump when there is an abrupt change in the error, by applying the derivative action to the measured variable instead of the error.

To the modified speed algorithm will then come:

$$\begin{aligned} C[n] = C[n-1] + K_c\Bigg(& (y_m[n-1] - y_m[n]) + \frac{T}{\tau_i}e[n] \\ & + \tau_d\frac{-y_m[n]+2y_m[n-1]-y_m[n-2]}{T}\Bigg) \end{aligned} \tag{5.14}$$

5.1.1.4. Optimal control parameters determination

As the controller in use is a discrete PID, the first step is to find the optimal parameters. For the first set of simulations, the aim is to find the values for the PID gains. As purpose of the car is to explore hazardous

areas, the parameter K_d will be set as 0 due to the noise.

Resorting the matlab functionality to automatically tune the PID gains for the system, the outcome is the

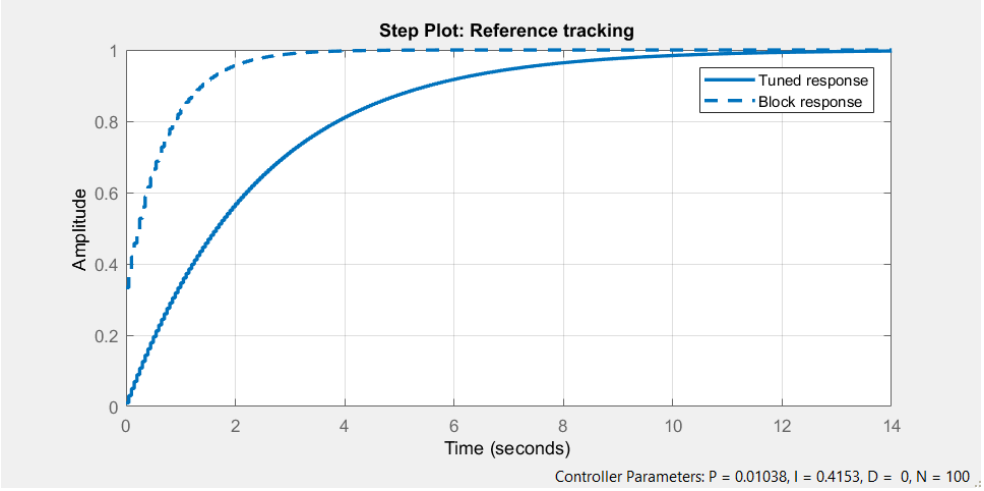

Figure 5.4.: Automatic matlab tune

present in the figure 5.4. It is possible to see that the velocity of the car reaches the reference value in approximately 11 seconds which is too long. The aim is for the car to get to the speed reference as fast as possible, so, in order for that to happen, new values of K_p and K_i have to be found in simulations.

For the first simulation, $K_p = 1$, $K_i = 1$, Linear speed = 1 m/s, $\theta = 0$ rad: With the figure 5.5, it

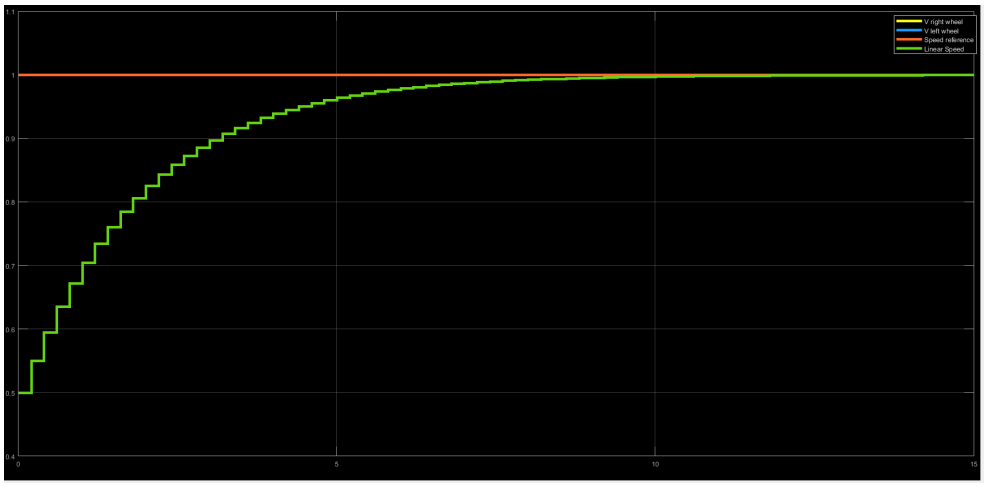

Figure 5.5.: $K_p = 1$, $K_i = 1$

is possible to see that the initial value of the velocity of the car is 0.5 m/s and it take about 7 seconds to reach steady state. 0.5m/s as an initial value for the linear velocity is a considerably high value and can cause the car to slide, therefore, the K_p value needs to get lower, for the initial value to get lower as well.

In the next simulation 5.6, K_p will be set as 0.5 and K_i as 1. Changing the value of K_p to half of the initial

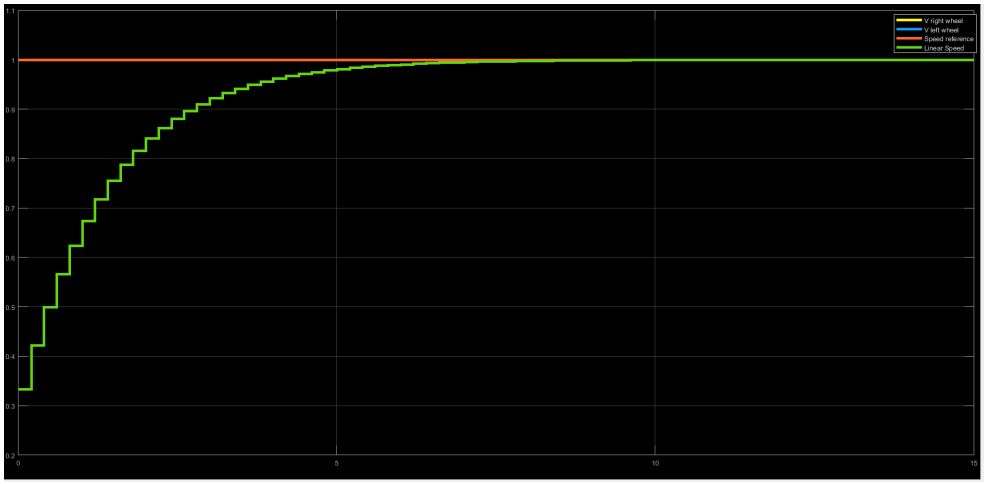

Figure 5.6.: $K_p = 0.5$, $K_i = 1$

value, it is possible to see that the initial linear speed is now 0.3 m/s, making it harder for the car to slide. It takes nearly 6 seconds for the car to reach steady state, thus the K_i value must be increased. In the next simulation 5.7, K_p will be set as 0.5 and K_i as 2. As the K_p value was not changed, the initial linear

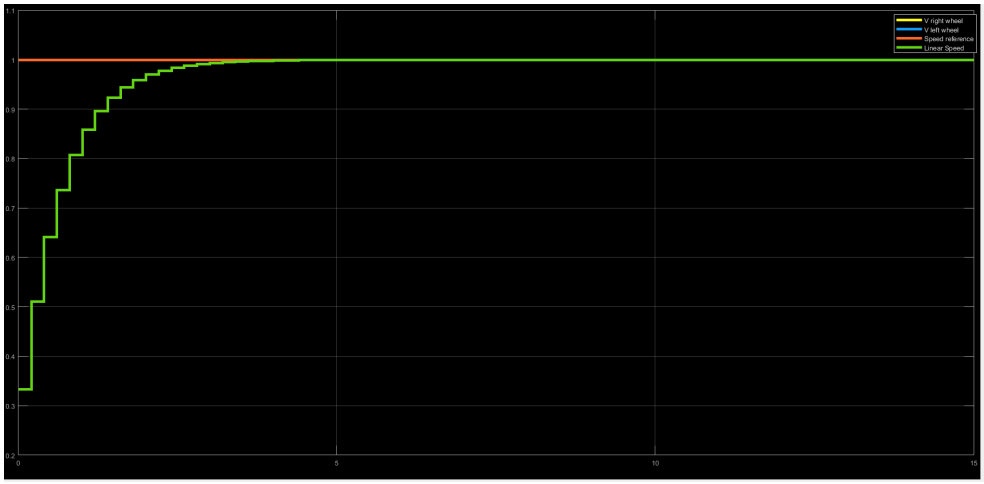

Figure 5.7.: $K_p = 0.5$, $K_i = 2$

speed value remains the same. As for the time it takes the car to reach steady state, it was reduced to 3 seconds due to the increase of the K_i parameter.

In ideal conditions, the aim would be for the car to reach the desired speed instantly, but due to the real limitations, such as wheel sliding, 3 seconds is a safe value.

Parameters	Values
K_p	0.5
K_i	2
K_d	0
Sampling time	50 ms

5.1.1.5. Sampling time determination

In order to choose the sampling time, one needs to take in consideration that with the decrease of the sampling time, the processing overhead will increase and with the increase of the sampling time, the system response will have abrupt changes affecting the performance of the car. So, in order to accommodate both necessities, the sampling time will be set to 50ms.

5.1.1.6. System response

Having determined the parameters of the controller, the next step is to simulate the response of the system. In order to predict the behavior of the car to a linear speed reference and angle of tilt (θ), some simulations were necessary. The output of the simulation is the plot of the linear speed of the car, the linear speed of both wheels (left and right) and the position of the car. In order to plot the position of the car the Cartesian referential is used.
For the first simulation, the parameters were: Speed reference= 1m/s and $\theta = 0$ rad.

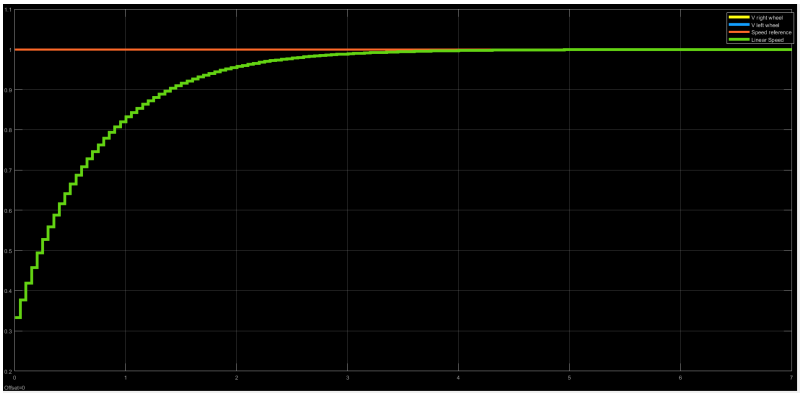

Figure 5.8.: Linear speed with reference v=1m/s,$\theta = 0$ rad

As expected,the figure 5.8 shows that the car linear velocity reached 1m/s. The angle of tilt is equal to 0 which means the car will be moving in a straight line, and as such, both wheels will are moving at the same speed of the car.

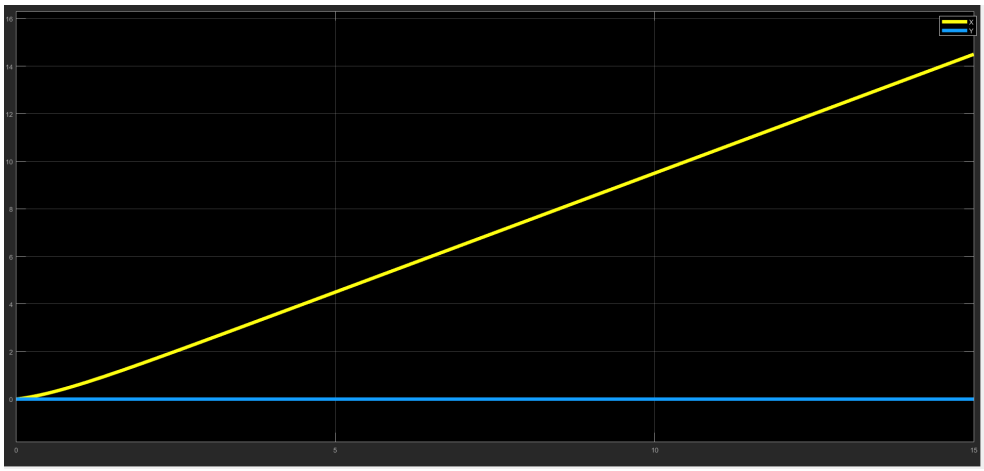

Figure 5.9.: Car position(xy) with reference v=1m/s, $\theta = 0$ rad

As the θ is equal to 0 rad, only 1 coordinate of the car is moving, as the figure 5.9 demonstrates. The x coordinate is equal to 0 the entire simulation time, and the y coordinate is increasing with a linear scope equal to the linear velocity of the car. This implies that the car is indeed moving in a straight line.
Changing the θ to 0.1 rad to simulate constant tilt of the smart phone to the right, and maintaining the value of the speed reference in 1 m/s :

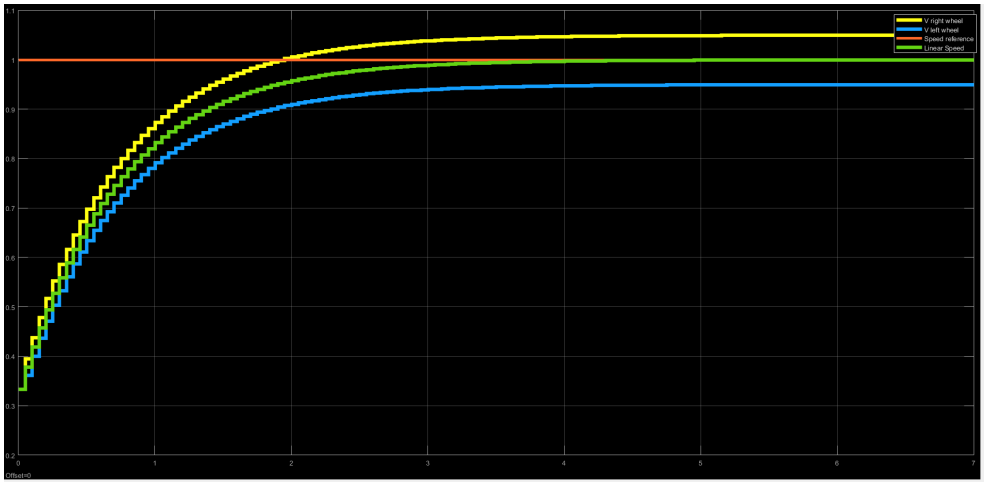

Figure 5.10.: Linear speed with reference v=1m/s,$\theta = 0.1$ rad

In the simulation figure 5.10 it can be observed that having a angle of tilt not equal to 0, causes the left and right wheels to have different velocities, in order to make the car turn. Running more simulations with different values of θ, the outcome shows that the bigger the module of the value of θ, the bigger the difference between the linear velocities of the wheels.

With this figure 5.11 it is possible to observe that both the position of x and y of the car change with time.

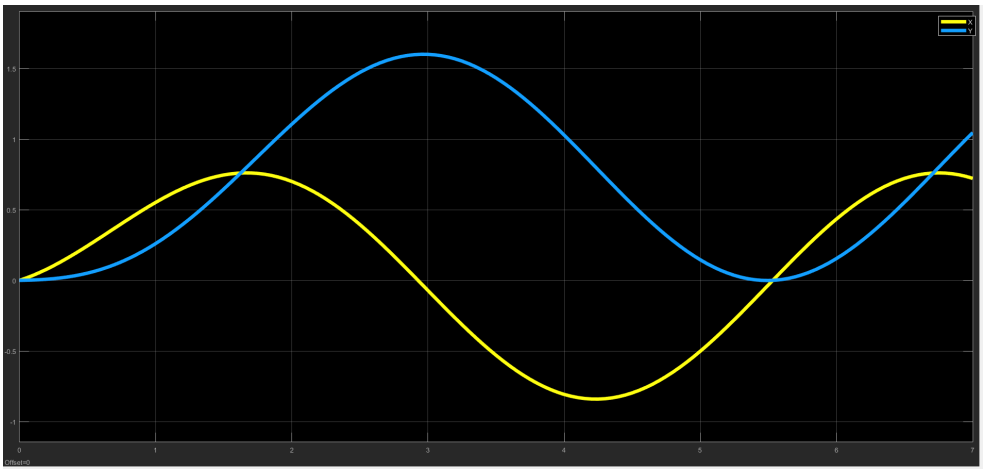

Figure 5.11.: Car position(xy) with reference v=1m/s, $\theta = 0.1$ rad

With a constant angle of tilt, the car will turn constantly in the same direction, eventually making a 360 degrees turn and as the car as small dimensions, it takes a very small time for it to do so, which is what is observed is this simulation.

5.1.1.7. Obstacle Avoidance Through Odometric Sensors

Wireless communication between devices is always prospect to errors and obstructions, as such a way had to be devised so that the rover will function safely when communications fail whilst also avoiding a significant decline in the system's life expectancy.
In order to achieve this, several odometric sensors will be placed on the rover, covering the 360° radius, and whenever it judges an obstacle is too close (through the feedback of the sensors) it will not follow commands that would force it to get any closer to said obstacle.
The implementation of an autonomous obstacle avoidance algorithm was considered, however implementing such an algorithm would, in the best scenario, remove control from the user and, in the worst scenario, enter into direct conflict with the latter. Therefore, since neither of the aforementioned scenarios was in the best interests of either the system or the user, it was opted to simply force the system to stop if a command would take it too close to an obstacle and to not allow it to move towards it any further.

5.1.1.8. Obstacle Avoidance Simulations

In order to plot the rover's path on a 2D diagram, the Mobile Robotics Simulation Toolbox, made available by MATLAB, will be used. A series of waypoints (these will simulate the user's commands) will be placed on a 2D map with borders, the latter of which will be used as obstacles. The waypoints shall direct the

rover towards one of the borders in a turn and a straight line. Ideally the rover will stop before reaching the obstacle whilst also keeping enough space to turn away from it.

The initial position of the rover shall be at coordinates (1,1) and an angle of 0 rads, as depicted in Fig. 5.12. From this standpoint, two simulations will be considered: a straight line that will require a $\frac{\pi}{2}$ rads turn upward (this turn serves to test the algorithm, to make sure it does not stop changes in direction whilst away from obstacles) and head straight towards one of the walls of the map; and a second simulation, involving a turn towards a wall. In order either simulation to be a success the rover must stop before crashing against the wall with enough space to steer away from the obstacle.

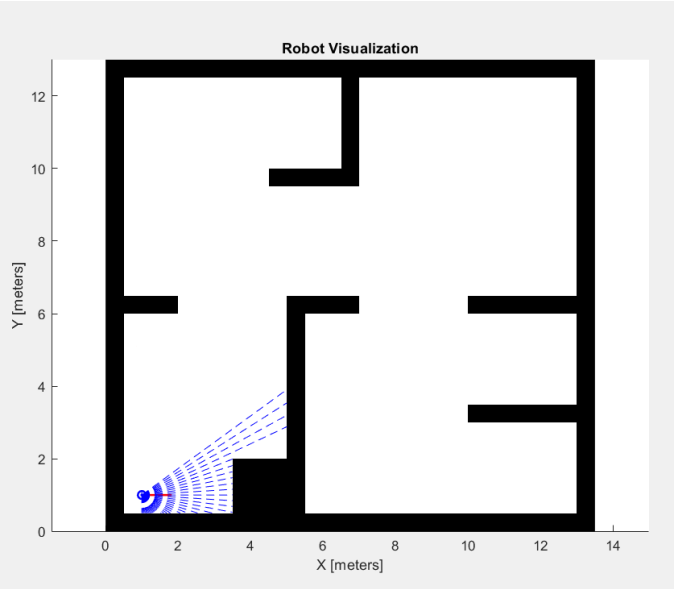

Figure 5.12.: Initial State of simulation

Firstly it must turn towards the wall, Fig 5.13.

Afterwards it must head in a straight line and stop before the obstacle, Fig 5.15

In the second simulation, which shall start, once again from the initial position depicted in Fig. 5.12, the rover must turn towards a path that avoids the right wall, Fig. 5.15 and afterwards turn again and stop before reaching the wall with enough space to turn away from the latter, Fig. 5.16.

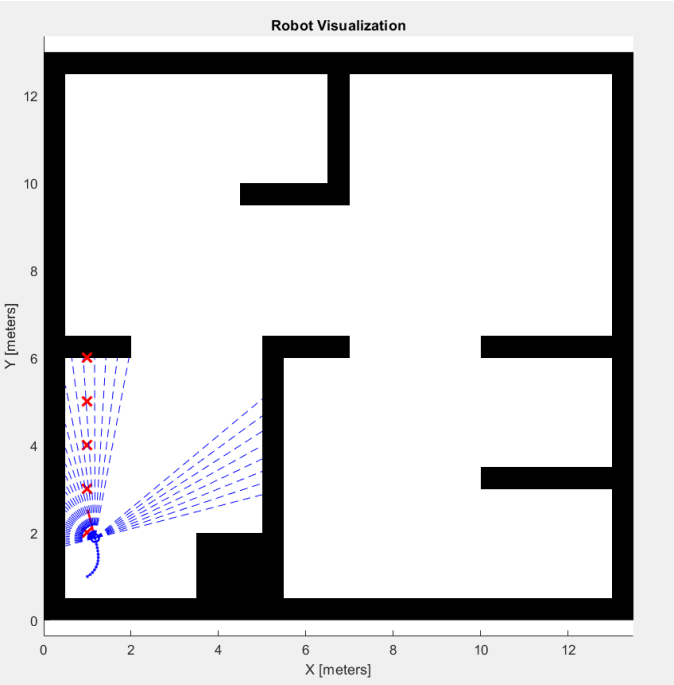

Figure 5.13.: After Rover Turns Towards Wall

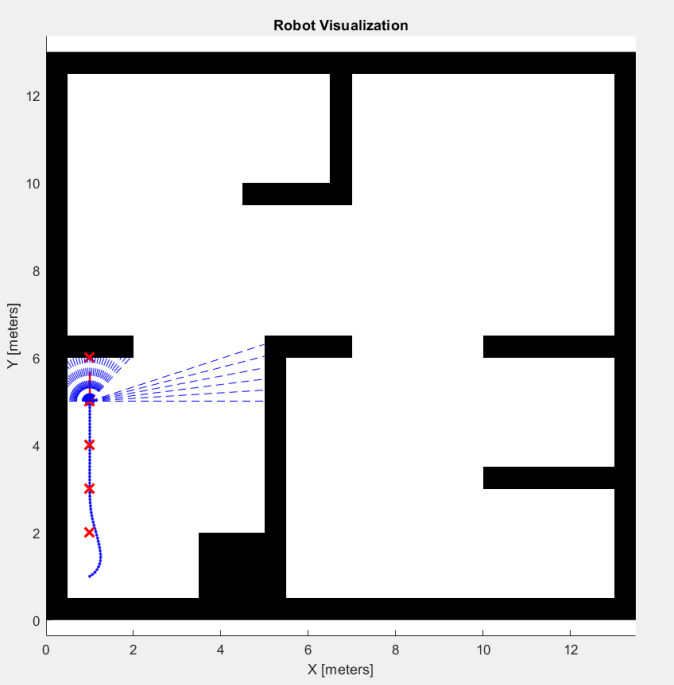

Figure 5.14.: Rover Stopped Before Reaching Wall

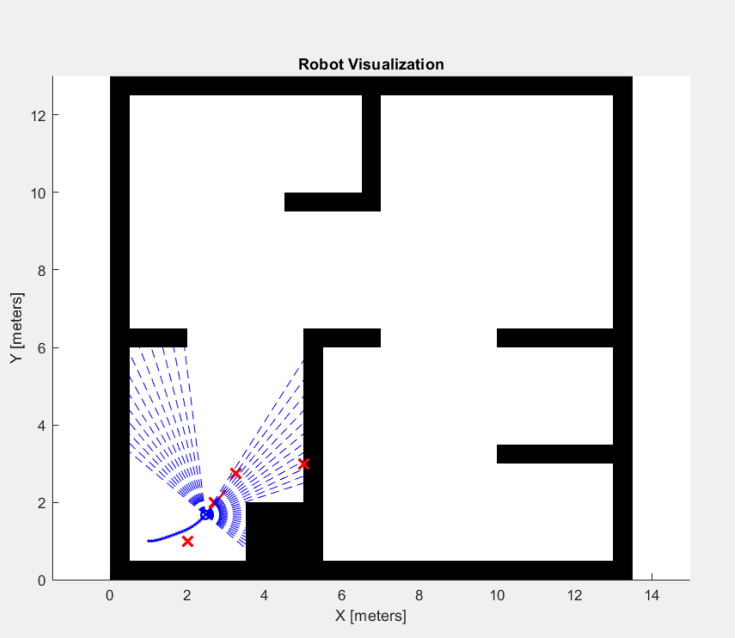

Figure 5.15.: Rover Starts the Turn

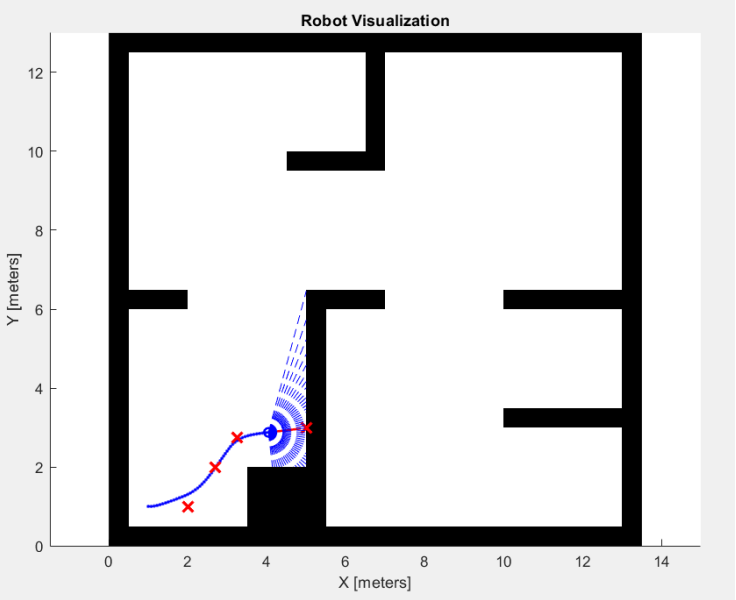

Figure 5.16.: Rover Turned and Stopped Before Reaching Wall

5.1.1.9. Odometric Sensor

For the obstacle avoidance, several of GP2Y0A21YK Infrared Sensors (Fig 5.17) will be placed on the rover, since it has an allowable field angle of up to 40°, this would mean at least 9 sensors would be required to cover 360°, this is assuming that they are placed in such a way that there is no overlap, however since any sort of "blind spot" will place the rover in jeopardy, some overlap is desired. Considering the rover's dimensions, it is unlikely that 9 sensors can be placed upon it with enough overlap to completely cover the 360°, therefore it is likely that a few more will have to placed to guarantee safety of the rover.

Figure 5.17.: Odometric Infrared Sensor GP2Y0A21YK

5.1.2. System design

Tackling the objectives laid out in Section 5.1 requires an organized and well thought-out plan of work because there are a lot of smaller systems at play that need to work cohesively and synchronously.

The best way to achieve a good plan is by first separating the problem into packages, and in each package have subpackages, each with a collection of modules dedicated to serve a collective purpose in the data treatment and organization chain.

The system is divided into such packages as follows:

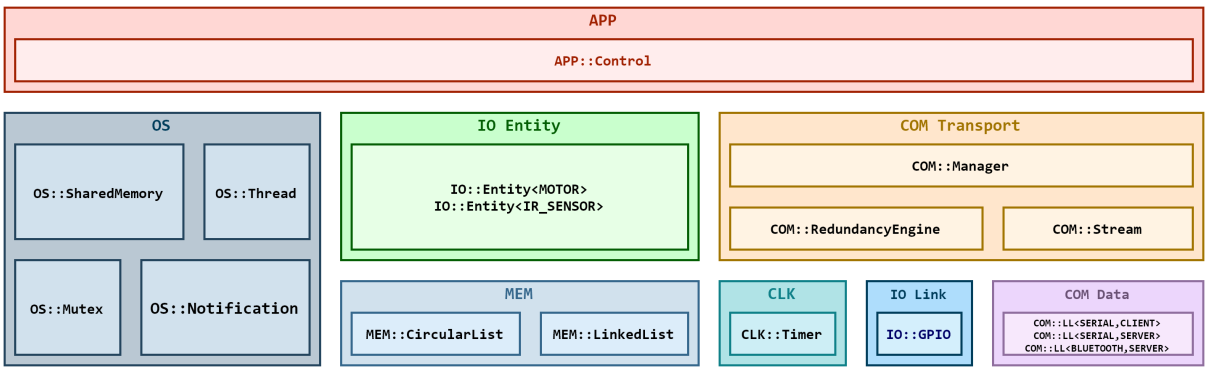

Figure 5.18.: Full Stack Overview

Each subpackage also belongs to a certain layer of software, characterized not by the resources it is associated to but by how close the modules within it are to the hardware. Futhermore, the packages should be distributed between layers in such a way that one seldomly need to use another that doesn't belong to the same layer or the layer directly below.

These layers are, from the bottom to the top:

- The High-level Hardware Abstraction Layer, which consists of the **OS** (partly), **MEM**, **CLK**, **IO Link** and **COM Data** packages/subpackages. Modules within this layer are responsible for the lower-level interaction with the system's resources so they are made to be thread-safe. Their implementation is platform-dependent and their interface is platform-agnostic. The main goal for this layer of software is to standardize the hardware, making for clean, maintainable and easily portable code.

- The High-level Software Abstraction Layer, consisting of the **OS** (partly), **IO Entity** and **COM Transport** packages. The modules within these packages should serve as an interface with the lower level layer, creating a more intuitive interaction process and mechanisms for processing information asynchronously. This way, when other modules make use of those interfaces, the information is already parsed and ready to be retrieved.
  Modules in the layers above ara also highly dependent on both abstraction layers to carry out their tasks in time so the ones in this layer should provide robust mechanisms for exception-handling and timing.

- The Main Application Layer, comprised only of the **APP** package, where the core functionality lies. As stated earlier, modules in this layer are protected from having to access to most lower-level layers but can use those modules and must use when no other abstraction is provided.

5.1.2.1. Static Model: Package diagram

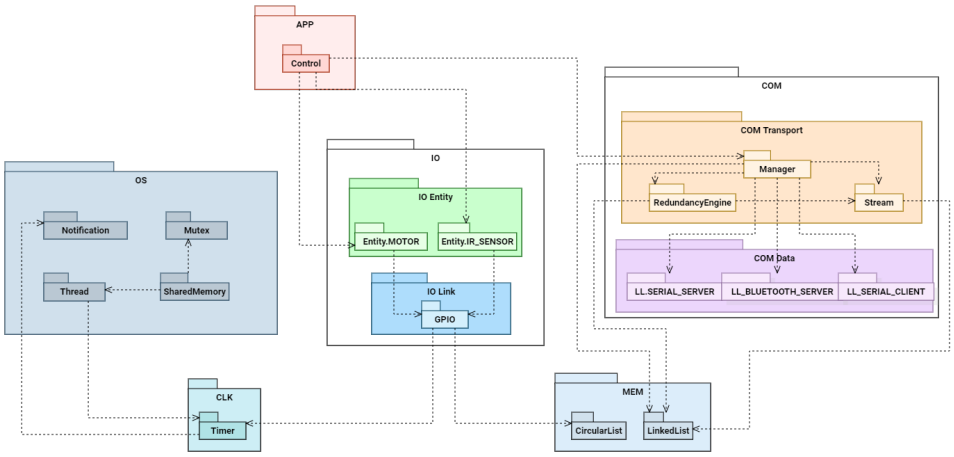

Figure 5.19.: Navigation subsystem package diagram

5.1.2.2. Static Model: Class diagram

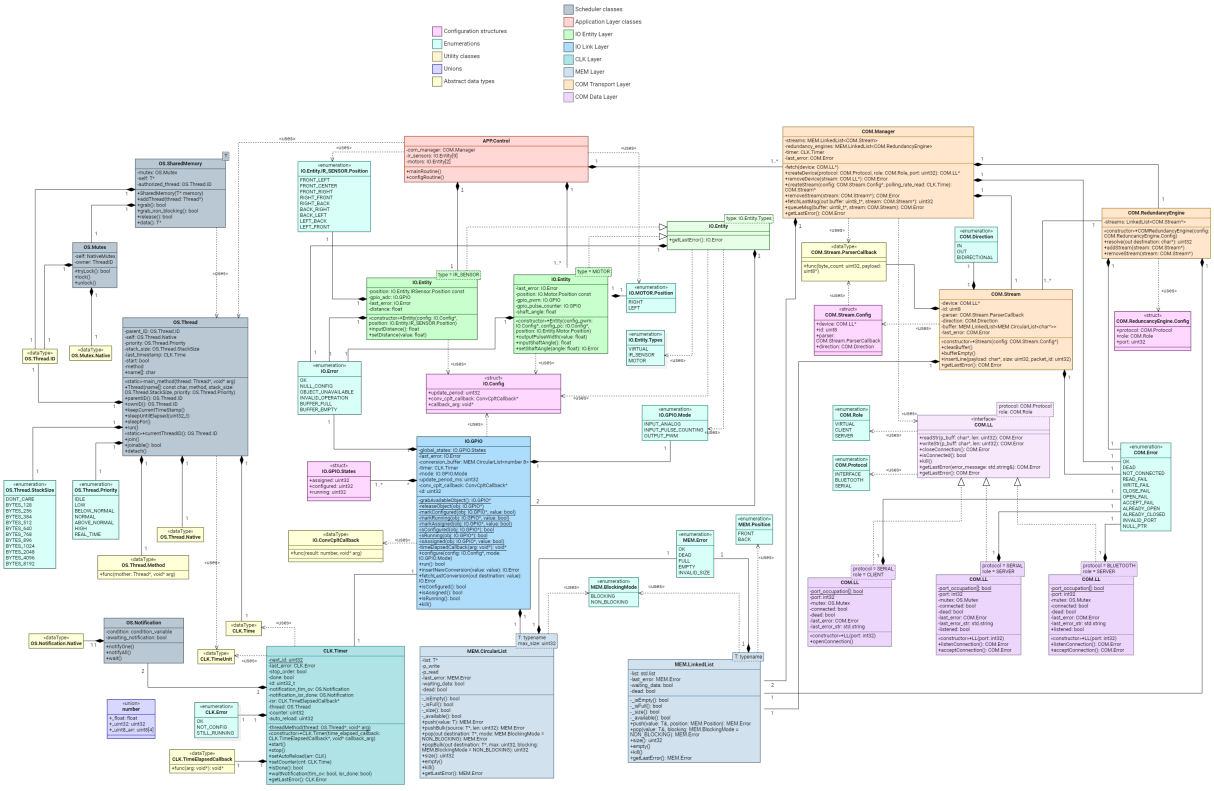

Figure 5.20.: Navigation subsystem class diagram (augmented in Appendix B.1)

5.1.2.3. IO: Input/Output Package

The **IO** package is comprised of the **IO Entity** and **IO Link** subpackages. The modules within these subpackages are the parts of the hardware abstraction layers responsible for standardizing the General Purpose Input/Output resources of the machine.

The **IO Link** subpackage is the interface provides the most generic yet complete package for interacting with the machine. Its only module **IO::GPIO** provides such functionality as automatic resource assignment upon creation, automatic buffering of read and output values and the option of executing a specific piece of code when a conversion is completed for better effort distribution between layers.

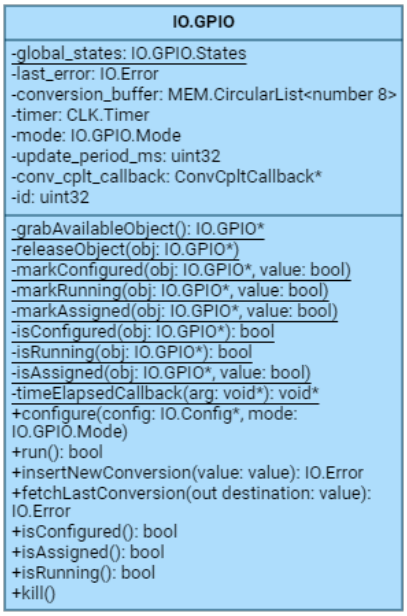

Figure 5.21.: IO::GPIO interface and members

The **IO::Entity** module from the **IO Entity** subpackage serves for specialization of functionality present in **IO Link** to serve a certain purpose attached to a physical meaning. This means it also extends the aforementioned callback mechanisms from **IO::GPIO** for automatic calculation of real-world values based on the measurements taken or otherwise that can be adapted and tailored further to serve the purpose of the specific application. Both provide a modest interface for storing important values calculated in the context of a IO::ConversionCpltCallback and a way to retrieve them.

Its MOTOR specialization uses an instance of **IO::GPIO** as an INPUT_PULSE_COUNTING to be able to count the number of pulses from the motor's encoder and output and one as an OUTPUT_PWM to be able

to output a pulse width modulated signal with a duty-cycle D such that

$$V_{outaverage} = \frac{D \times V_{max}}{100}$$

The **IR_SENSOR** specialization uses an instance of **IO::GPIO** as an **INPUT_ANALOG** for application in IR sensing applications. It provides a way to change the last calculated distance specifically for use in the corresponding callback function and an interface for retrieving that same value.

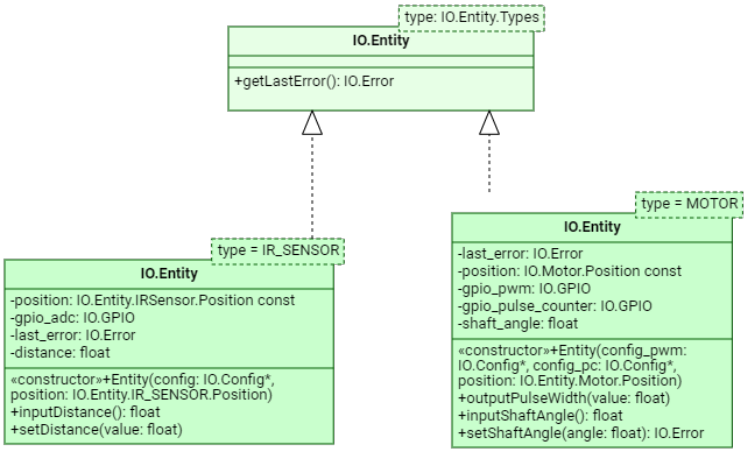

Figure 5.22.: IO::Entity interface and specializations' methods and members

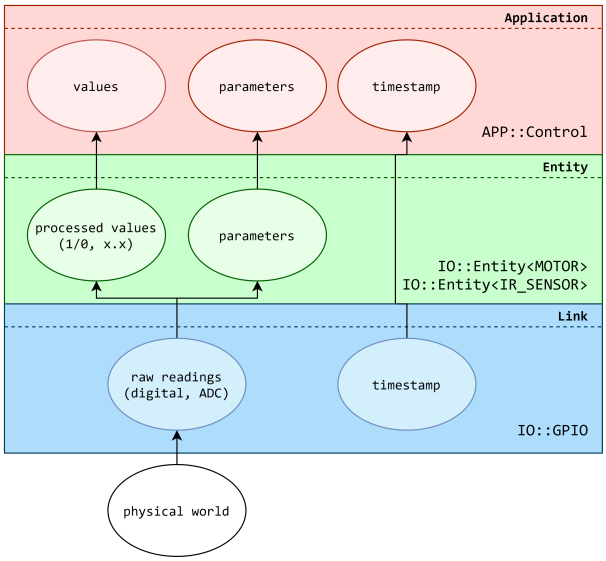

Figure 5.23.: IO subpackage interaction and information propagation diagram

5.1.2.4. COM: Communications Package

The COM package is the sum of the COM Data and COM Transport subpackages. These modules are responsible for standardizing the access to the inter-device communication resources of each machine.

The **COM Data** subpackage is aimed at providing a platform-agnostic interface for communicating over a serial or Bluetooth connection, while also providing specific functionality for different protocols/roles. The **COM Transport** subpackage provides tools for managing multiple simultaneous, redundant, multiprotocol and multi-stream connections as well as automatically parsing of data for usage in time-constrained scenarios.

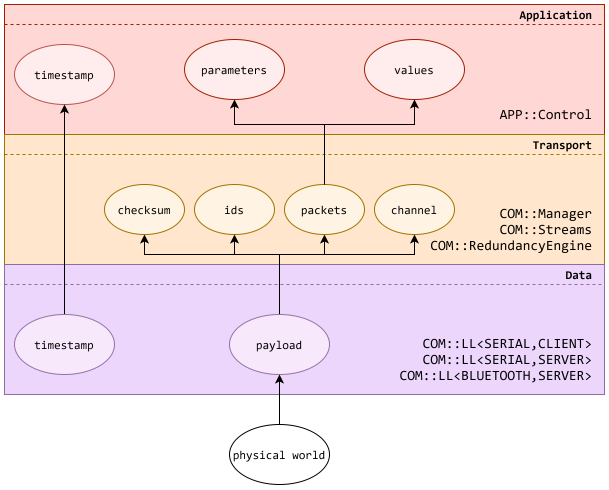

Figure 5.24.: COM subpackage interaction and information propagation diagram

The **COM::LL** module makes up the lower layer of the COM package. It should then have very well defined and error-tolerant behaviour and a somewhat familiar interface. With that in mind, it had an error-reporting system that would give feedback over a variery of errors such as invalid ports or non existing connections. It has been envisioned to be capable of connecting devices through serial and bluetooth protocols using a typical client and server socket model.

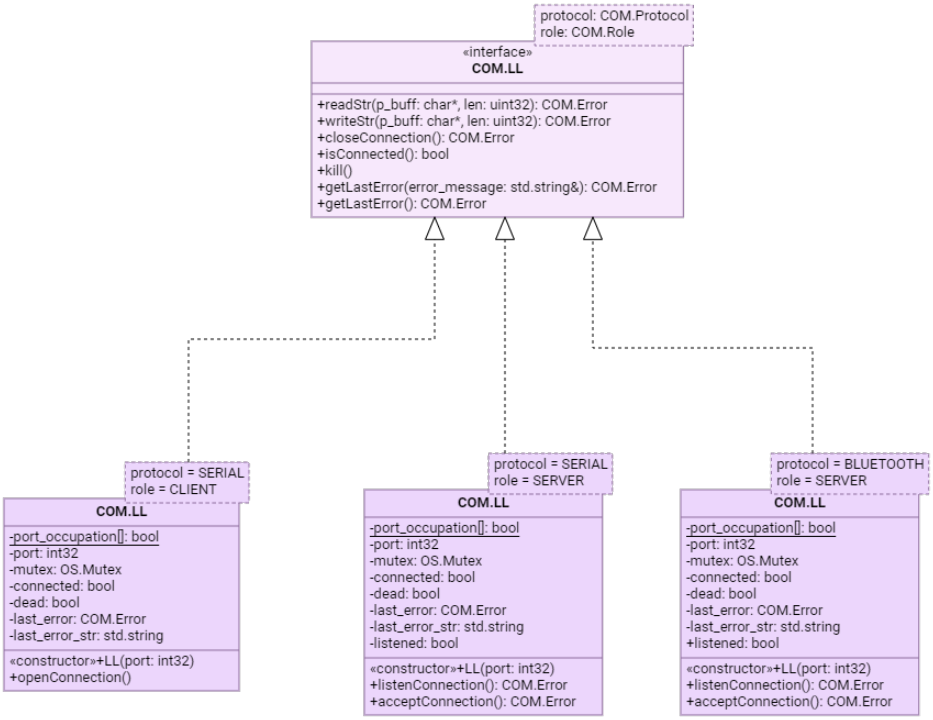

Figure 5.25.: COM::LL interface and specializations' methods and members

The **COM::Stream** module allows the partitioning of communications effected through the same medium into streams, resulting in well established boundaries which simplify any information processing required. It allows the formatting of any message into a recognizable packet complete with all the required metadata, for packet and stream identifications and simple error detection. It also permits the establishment of callbacks for effort distributing in parsing the information.

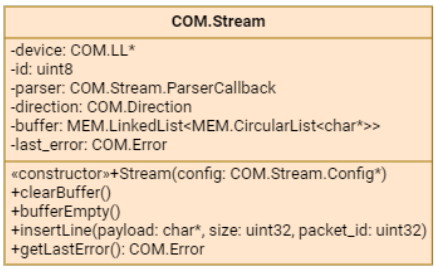

Figure 5.26.: COM::Stream interface and members

The **COM::RedundancyEngine** module allows several related streams "connect" and "disconnect" to and from it and compares the information between them resolving conflicts and missing information. The

latter of which can be accessed through a call **resolve()**.

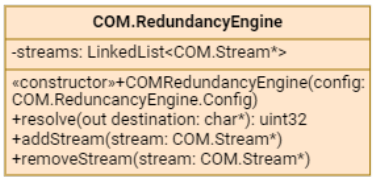

Figure 5.27.: COM::RedundancyEngine interface and members

The **COM::Manager** module is a higher level when compared to the others, it connects the tools of the other modules to create an interface that allows quick and intuitive communication setup and automatization of processes such as redundancy management.

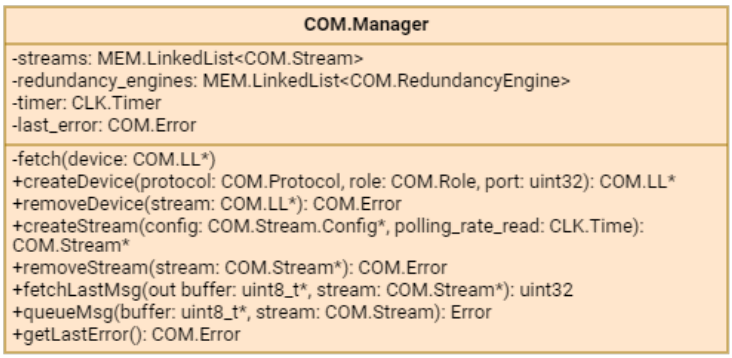

Figure 5.28.: COM::Manager interface and members

5.1.2.5. OS: Scheduler Package

The modules in the **OS** package mainly serve the purposes of thread creation and management management, inter-thread synchronization and thread-safe access to memory. Therefore, they all revolve around threaded execution although each fills in the need for a special bit of functionality.

The **OS::Thread** module provides a platform-agnostic interface for creating an object that will use native types and interfaces that allow each object to behave like a separate execution thread. The provided controls for it include execution and sleep mechanisms as well as options for waiting for the end of their execution and detaching threads from the parent thread. Scheduling can be managed with the options concerning thread priority and for the possibility of creating static threads it also provides the chance of delimiting the stack size.

Figure 5.29.: OS::Thread interface and members

The **OS::Mutex** module provides a way for controlling access to shared resources that are not defined as thread-safe, which is any object that the user might create that is not of any of the modules in the High-level Hardware Abstraction Layer. It can be locked (**lock()**) in one thread in order to block access to a specific excerpt of code and it can only be locked again by the same thread or any other after it is unlocked (**unlock()**) by whatever thread locked it. It is thus non-recursive, which means that any thread, including the thread that locked in blocked in the call to **lock()** it until it is unlocked.

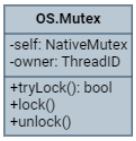

Figure 5.30.: OS::Mutex interface and members

The **OS::SharedMemory** module, one the other hand, provides a platform for sharing resources between specific threads. It may be considered an expansion upon **OS::Mutex** that is specifically tailored to be used in memory protection and can reduce the roster of threads that get access to a certain memory area.

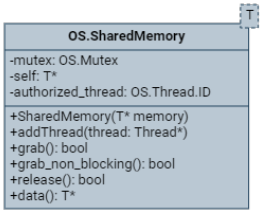

Figure 5.31.: OS::SharedMemory interface and members

OS::Notification is a small module that provides the inter-thread synchronization method known as a signal or a notification. A call to **wait()** from one thread will make it halt until a call for **notifyOne()** that targets it is given from another thread. A call to **notifyAll()** will notify all threads.

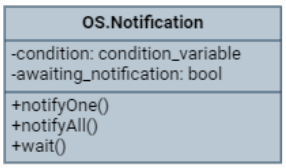

Figure 5.32.: OS::Notification interface and members

5.1.2.6. MEM: Memory Structures Package

The **MEM** package includes modules that offer typical containers such as linked lists and circular lists that are tightly contained and thread-safe. These are very practical ways of implementing those types of structures in heavily threaded applications as they provide easy-to-use interfaces, low-complexity operations and guarantee safe access to the resources when shared by multiple threads.

Both **MEM::CircularList** and **MEM::LinkedList** provide interfaces for inserting and removing objects, retrieving basic information about their state, such as their size and a means to empty the list in an efficient fashion. They both also implement the universal error reporting interface with a specific type for such errors, **MEM::Error**. When making specific operations the user can also choose how the container should react when another thread of execution is accessing the structure: in **MEM::BlockingMode BLOCKING** mode, the list will wait until the resources are freed and in **NON_BLOCKING** mode, in the same situation it will return with an error code.

The **MEM::CircularList** provides a medium for creating a fixed-size First-In-First-Out (FIFO) container to which the user can push a single or multiple objects and retrieve them in a similar fashion.

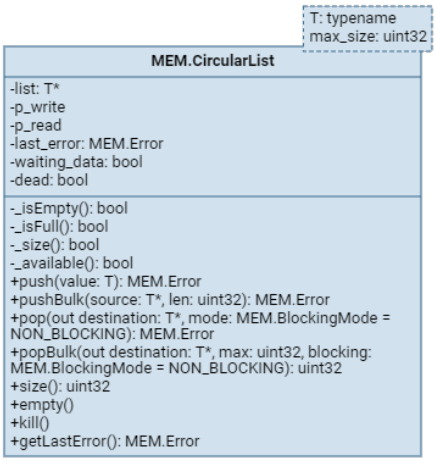

Figure 5.33.: MEM::CircularList interface and members

The **MEM::LinkedList** prioritizes quick insertion at the front or the back of any list and removal of specific objects from anywhere in the list with the same complexity. It it optimized exclusively for operations with single objects.

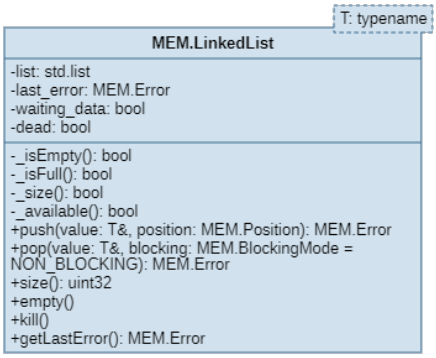

Figure 5.34.: MEM::LinkedList interface and members

The modules within this package are mainly used by the **IO** and **COM** class modules for buffering information and storing objects in lists where insertions and removals need to be fast but they can also be used by higher-level classes due to their **versatile interfaces** and **general-purpose** nature.

5.1.2.7. CLK: Timing Package

The **CLK** package is comprised by only the **Timer** module and convenient type definitions. It provides a means for setting up repeated timed delays and executing a specific routine automatically when each

delay ends. It also provides an interface for waiting for the end of the timed delay and/or the execution of the specified routine and autonomously notifies all waiting objects of these events. It is mainly used by the **IO::GPIO** class for timing conversions but it can as easily be used by higher-level classes due to its **versatile interface** and **general-purpose** nature.

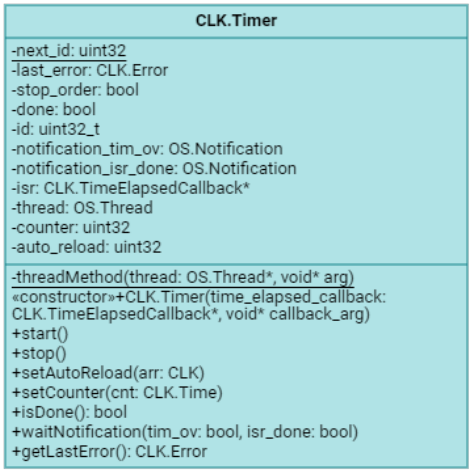

Figure 5.35.: CLK::Timer interface and members

5.1.2.8. APP: Main Application Package

The main application package is in practise comprised by the **main()** method which launches the Control Thread and any simulation thread that needs to be be running alongside it. This is explored further in section 6.1.2.

5.2. Physical Environment Virtual Subsystem

The Physical Environment Virtual Subsystem represents the interface with the physical environment, namely via sensors and actuators. It takes into consideration the model of the vehicle's dynamics and its effect on the environment, as well as the relevant events and inputs, thus leading to the loop illustrated in Fig. 5.36, consisting of controller and a simulator.

The user-side provides input values as the target velocity and steering angle for vehicle's navigation. Additionally, odometric sensors data must be collected to avoid obstacle collision, and, if necessary, override user commands to ensure its safety.

The controller is executed periodically ($t = T$), using the current speed of left and right wheels and the steering angle to generate new values for the command variables for the left and right wheels.

The simulator, responsible for modelling the vehicle's dynamics, is stimulated by the command variables and provides stimuli to the controller, determining the next values of the speed of the wheels and steering angle. Additionally, it assesses collisions, so the controller can take the appropriate measures to correct navigation. It is executed periodically, but at a higher frequency than the control, to add resolution, and out of phase the sampling period and out of phase with controller ($t = T/4 + \phi$), to avoid collisions between simulator and control processes.

The design of the physical environment subsystem paves the way for the assessment of the correctness of the control algorithms and its validation without requiring any hardware, which is a great advantage of the model-based design, diminishing development time and costs.

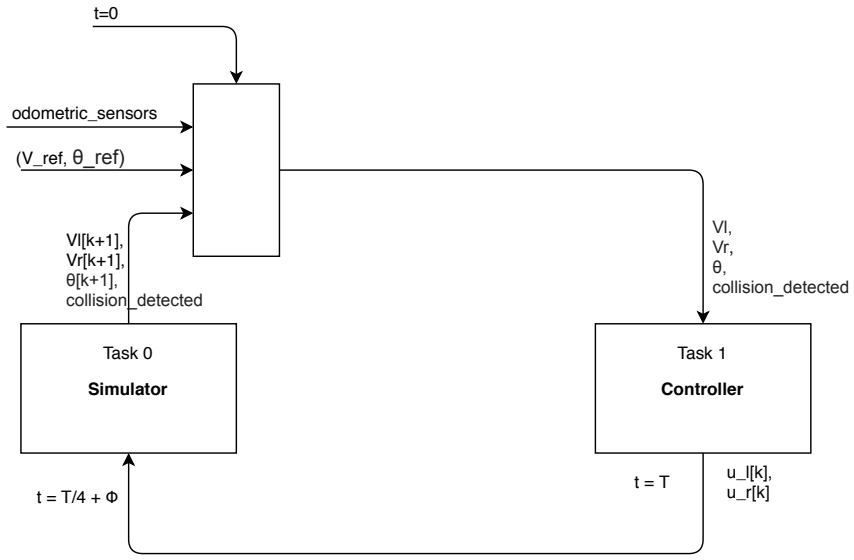

Figure 5.36.: Physical Environment Virtual Subsystem

5.3. Remote Vision Virtual Subsystem

The Remote Vision Virtual Subsystem (RVVS) is mainly responsible for providing visual feedback to the user of the vehicle's surroundings to assist its navigation. Additionally, it should offload the Navigation Virtual Subsystem (NVS) from other intensive, but no so critical, tasks — like telemetry data — as well as provide redundant paths for communications, especially in the most critical conditions, like off-grid navigation with the inclusion of the GPRS module. Thus, as aforementioned and illustrated in Fig. 4.2, it interfaces three

different subsystems: NVS via RS232 communication; smartphone via Wi-Fi or GPRS; and the web camera.

The RVVS design was divided into three phases, corresponding to the functional, object and dynamic models.

5.3.1. Functional model

The functional model describes the functionalities of the system from the actors' perspective, illustrated in Fig. 5.37 by an use case diagram. Two actors were identified: NVS and Smartphone. The Smartphone acts a proxy for user interaction, thus for clarity purposes, the actor was named User. Three main set of features were identified, namely:

1. **Communication**: deals with the communications interfaces between the various subsystems, further decomposed into the required functionalities for each interface (Comm Functions). The User may communicate with the Remote Vision Virtual Subsystem (RVVS) subsystem via GPRS or Wi-Fi, thus requiring the latter to provide network discovery capabilities, alongside with the conventional connect, send, receive and disconnect functionalities. Additionally, it may also required periodic KEEP ALIVE pings to check if the connection is still on. The NVS communicates via RS232, which does not require network discovery. Lastly, the user may require sensor information, which could potentially be connected to the RVVS subsystem (e.g., in a Inter-Integrated Circuit (I2C) network) for offloading the NVS subsystem.

2. **Image Acquisition**: responsible for providing visual feedback to the user. It can be configured, started, stopped and captured.

3. **Processing**: responsible for processing: user commands for image acquisition or forwarding them to the NVS subsystem as a redundant path; telemetry data, such as, overall distance traversed, maximum speed so far, and operation time.

5.3.2. Dynamic model

The dynamic model describes the internal behaviour of the system, represented in UML by sequence, state-machine, and activity diagrams.

The sequence diagram represents the sequence of events related to a particular use case. It aids to refine use cases and identify software objects (entity, boundary, control), providing a well-established path for the implementation of system's functionalities.

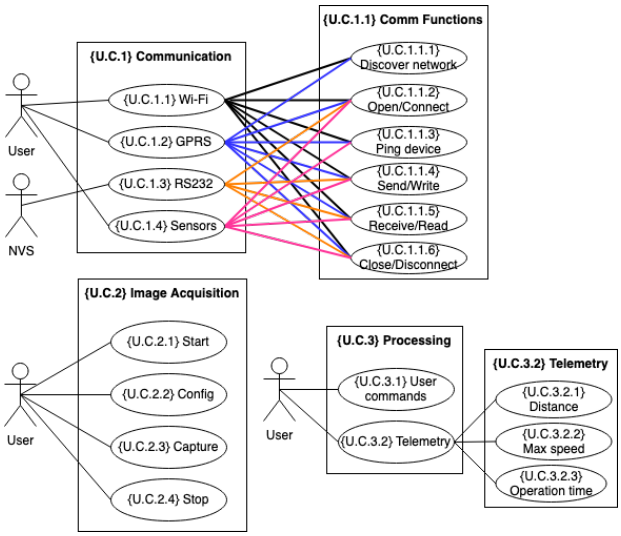

Figure 5.37.: Use case for RVVS subsystem

On the other hand, state-machine diagrams provide an overview of the system functionalities and how they interact, i.e., the system's states, transitions, and response to stimulus, internal or external. This makes it an excellent tool for designing overall system behaviour. Fig. 5.38 identifies the main elements used in the state-machine diagram.

The state-machine diagram for the RVVS subsystem is depicted in Fig. 5.39. On system startup is performed an initialization procedure, loading user-defined and machine settings and initializing the required hardware. After initialization is completed, four main processing units are executed in parallel until system is shutdown, namely:

- **Wi-Fi Manager**: manages Wi-Fi connection;
- **RS232 Manager**: manages RS232 connection;
- **Main**: idle processing unit; parses messages received through the available communications channels and, if a command is detected, pushes it to a command queue for later execution.
- **Scheduler**: periodically executed processing unit, responsible for executing periodical tasks and spawning tasks as a response to issued commands, with different priority levels.
  - **Task**: spawned as a response to a command, performs the required associated function and has a low expected lifetime. The commands can be classified as user, NVS, telemetry data, image acquisition, or communications.

Lastly, an example of a more refined state-machine diagram, in this case for communications, is illustrated in Fig. 5.40. In the initial state, a connection request is expected and if accepted, authentication

is requested. If successfully authenticated, the Communication Manager is ready to communicate, handling incoming/outgoing data by notifying the relevant entities and performing the associated operations. The RS232 does not require connection and authentication management, thus yielding only the states readyToCommunicate, MsgReceived, and MsgToSend.

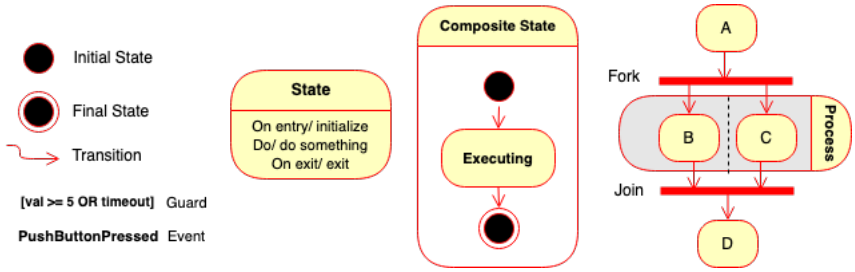

Figure 5.38.: State-machine diagram

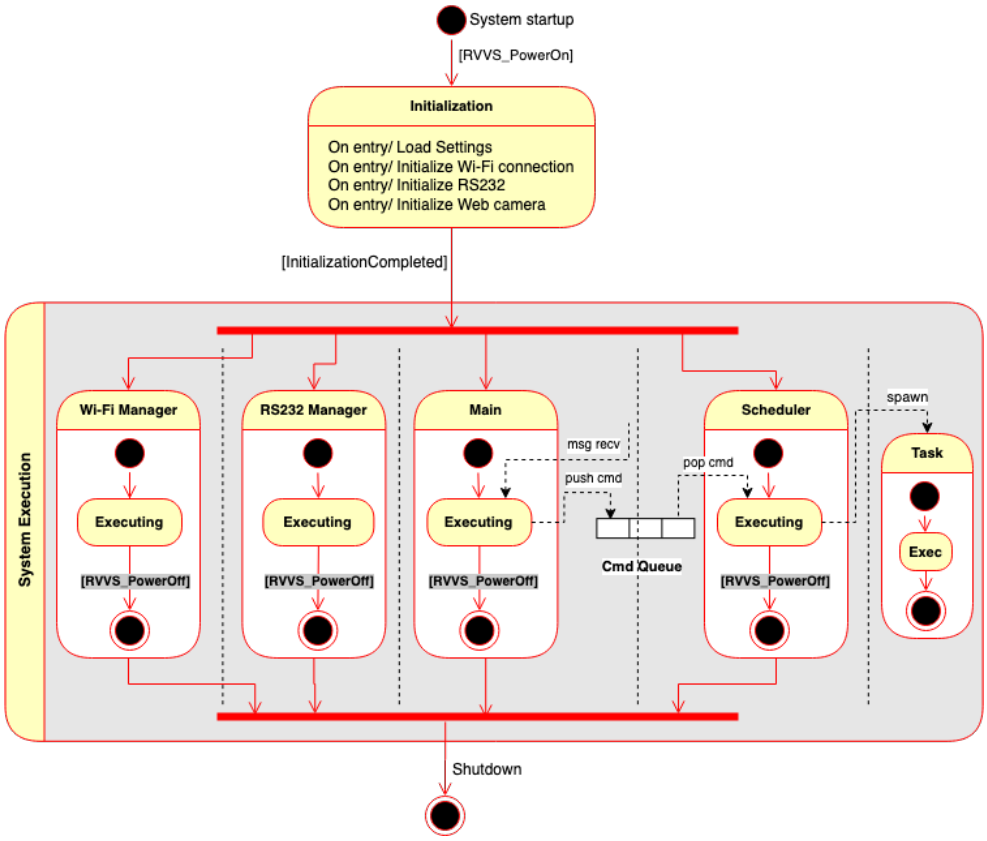

Figure 5.39.: RVVS state-machine diagram

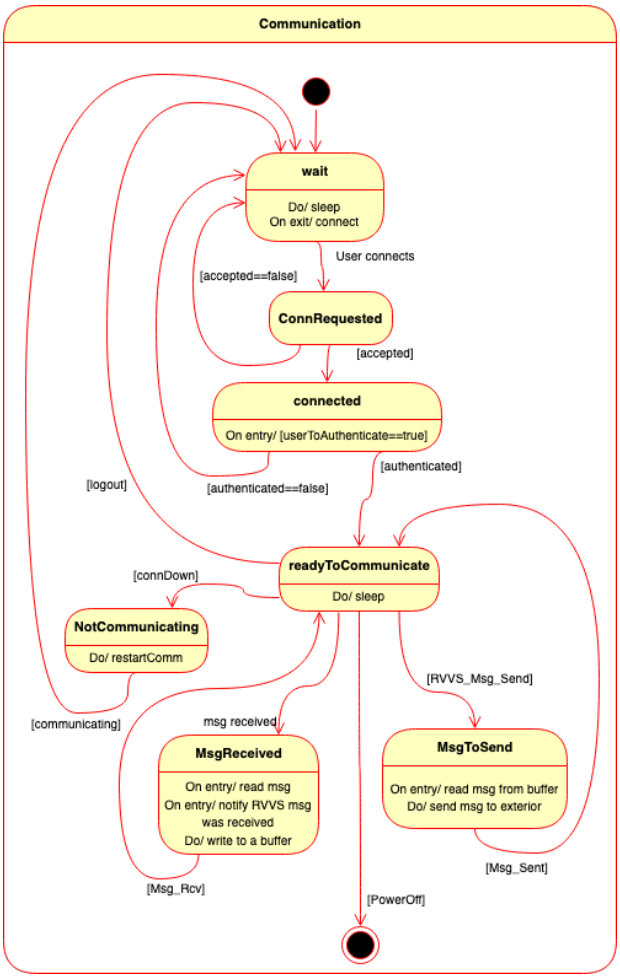

Figure 5.40.: Communication state-machine diagram

5.3.3. Subsystem decomposition

Following the same design rationale indicated in Section 5.1.2 for the NVS subsystem, the the RVVS system should be decomposed into smaller, more tractable, subsystems for easier development. Furthermore, modularity and reuse, whenever is possible, should be key design guidelines.

The subsystem decomposition for RVVS is illustrated in Fig. 5.40, taking into account these considerations. Thus, **OS**, **COM Transport**, and **COM Data** packages are reused from NVS for the following reasons: RVVS requires concurrent/parallel processing as provided by the **OS** package in a thread-safe way; it shares the communication interface with NVS (RS232), as provided by the **COM** packages. The same idea is extensible to the Wireless Communication (**WCOM** package), with the transport layer comprised of a **Manager**, a **RedundancyEngine** and a **Stream** for packet serialization/deserialization, and a low-level layer dependent of the technology used (Wi-Fi/GPRS). For **Image Acquisition**, a **Manager** handles image acquisition func-

tionalities, with the **Frame** acting as image data container; low-level layer **Img::LL** handles the hardware interface and low-level details. Completing the system stack is the **Telemetry** package, responsible for managing the telemetry data as required. This package could have a direct interface with the sensors, but, for the time being, it tracks information received by the NVS subsystem.

At the top of the software stack is the actual software executed on top of the system stack, containing the application logic and management. Thus, the **App::Manager** package is a event-driven processing unit, listening for relevant events signalled by the layers below and handling those events, as the one generated by successfully parsing of commands arriving from the available communications channels.

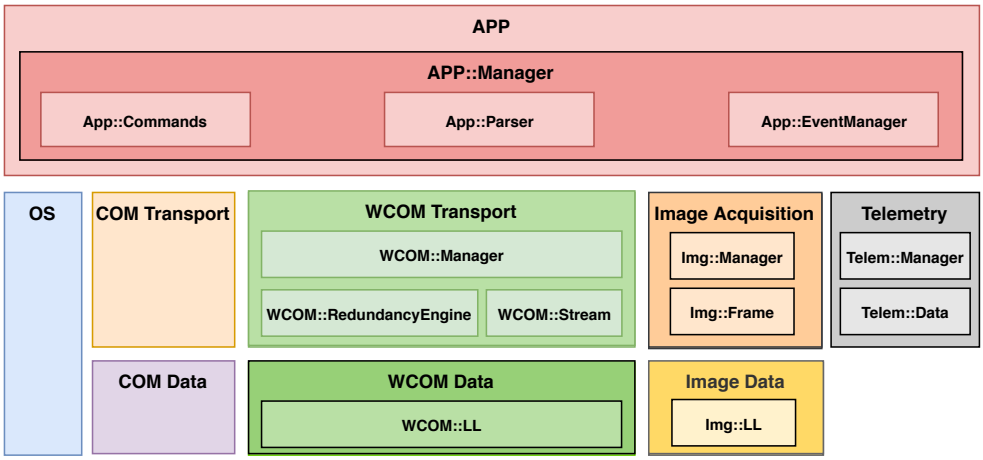

Figure 5.41.: RVVS full stack overview

5.3.4. Object model

The object model, represented in UML with class diagrams, describes the structure of the system in terms of objects, attributes, associations, and operations.

On Fig. 5.42 is displayed the class diagram of the remote vision subsystem. As aforementioned, as it shares common interfaces and design decisions with the NVS subsystem, only classes **WCOM**, **Webcam_V4L2** and **Telemetry** class will be discussed more in depth. The former is identical to **COM** class from the NVS, addressing the specifics of the **TCP** protocol and Internet programming sockets.

The class **Webcam_V4L2** has a constructor that receives the type of capture to perform and the type of buffer memory used. Additionally, it contains public functions to open/close the device, set the image parameters (dimensions and format), request memory buffers, and start a stream thtat outputs to a file.

In the private interface, helper functions **allocateBuffer**, **fileDescriptor** and **open** were defined. The first allocates the buffers as detailed by the setRequestedBuffer function. The second, maintains the integrity

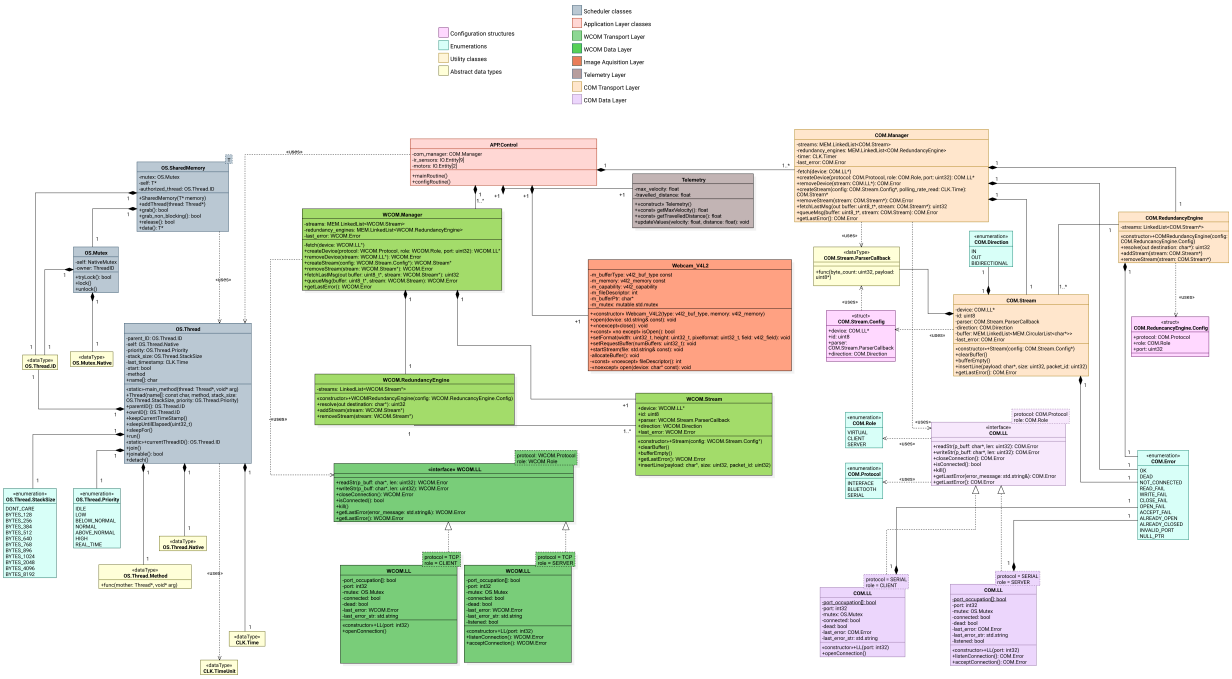

Figure 5.42.: Vision Class Diagram (augmented in Appendix B.1)

of the file descriptor. The last it is a low level call to open the device, called by the first open function that encapsulates it.

Lastly, the variable members of the class are from up to down, the buffer type, memory buffer, the buffer capabilities, file descriptor linked with the device, buffer pointer and a mutex variable to only allow access at the time, locking the desired resources to prevent unattended access. As illustrated in Fig. 5.43, the telemetry class has two variable members: the max velocity and distance travelled, accessed and updated with the succeeding member functions, in public.

In the first place, the default empty constructor, second by the functions that retrieve the member variable values, respectively the getMaxVelocity and getTravelledDistance. At last, an updateValues function is used to alter the aforementioned variables with new values.

5.4. Smartphone

The smartphone design was divided into three stages, the functional model, the object model and the dynamic model.

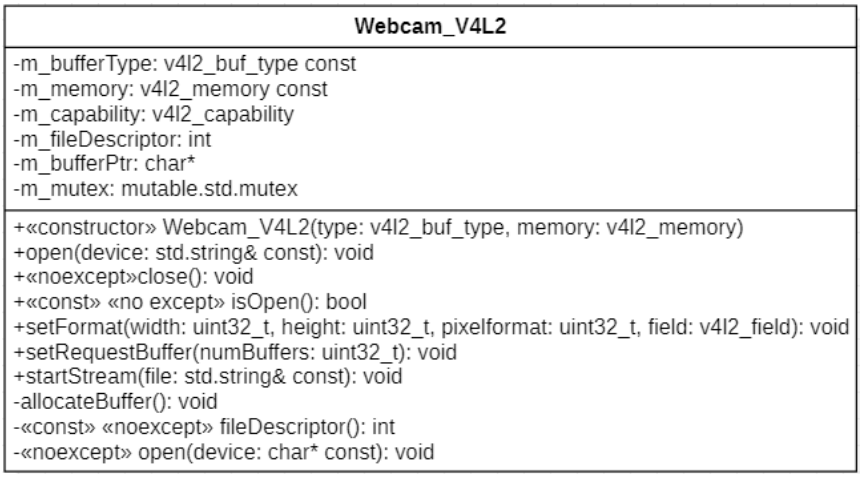

Figure 5.43.: Webcam class v4l2

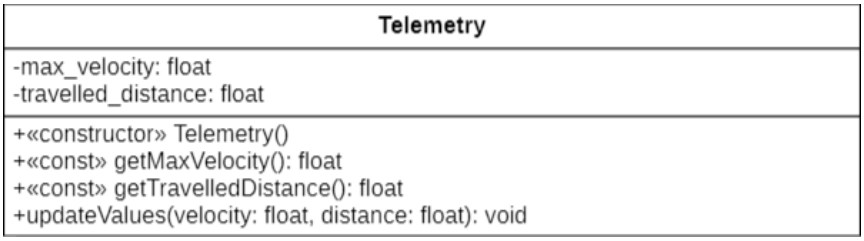

Figure 5.44.: Telemetry class

5.4.1. Functional Model

The functional model includes a description of the system in terms of accessible functionalities from the actors' point of view, represented in figure 5.45. In this stage, the actors identified were the user, the NVS and the RVVS. The user must have access to three key features:

- **Vehicle control**: the ability to control the car by tilting the smartphone and thereby changing the provided angle and speed reference. This feature implies a transmission of these values to the NVS and a UI value update.
- **Notification view**: pop-ups that allow a grasp of the current state of the system with informative or alert messages.
- **Video feed view**: Be able to visualize the RVVS' video transmission on screen.

On one hand, the NVS should be able to receive the accelerometer data provided while also sending notifications to the app, both via Bluetooth. On the other hand, the RVVS should be capable of transmitting the

camera´s video while also sending the same type of notifications to the user within the application. This communication is established via Wifi/GPRS.

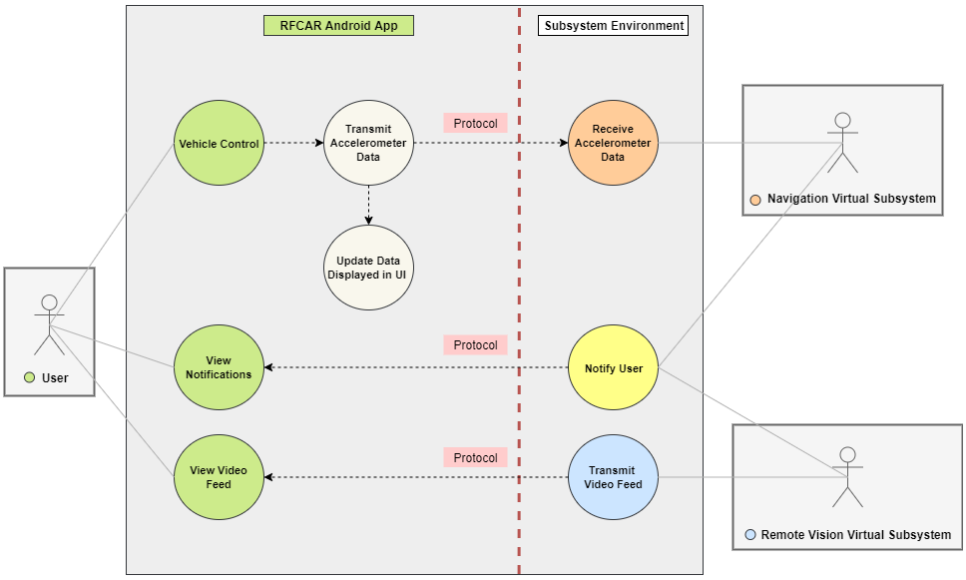

Figure 5.45.: Android app use case diagram.

5.4.2. Object/Static Model

In this step, the objective was to define the app system´s structure with UML class diagrams, concerning its objects, attributes, operations and associations established. The forementioned diagram is represented in figure 5.46.

5.4.3. Dynamic Model

For the dynamic model, were devised multiple state-machine diagrams to describe the internal behaviour of the application. Figure 5.47 illustrates the overall application behaviour. One can observe that all the intended features are meant to run in parallel, that will surely affect the system implementation in terms of concurrency (section 2.3) and thread management. Initially, the system loads the User Interface (UI) and waits for all connections to be established before moving to the next state. While the feature for vehicle control is running (figure 5.48) it retrieves the accelerometer data, sends the data to the NVS and displays it on the screen. Additionally, the feature that allows the user to see the video feed transmitted by the RVVS in figure 5.49 also displays it on the app screen. When this Wifi/GPRS connection is suspended

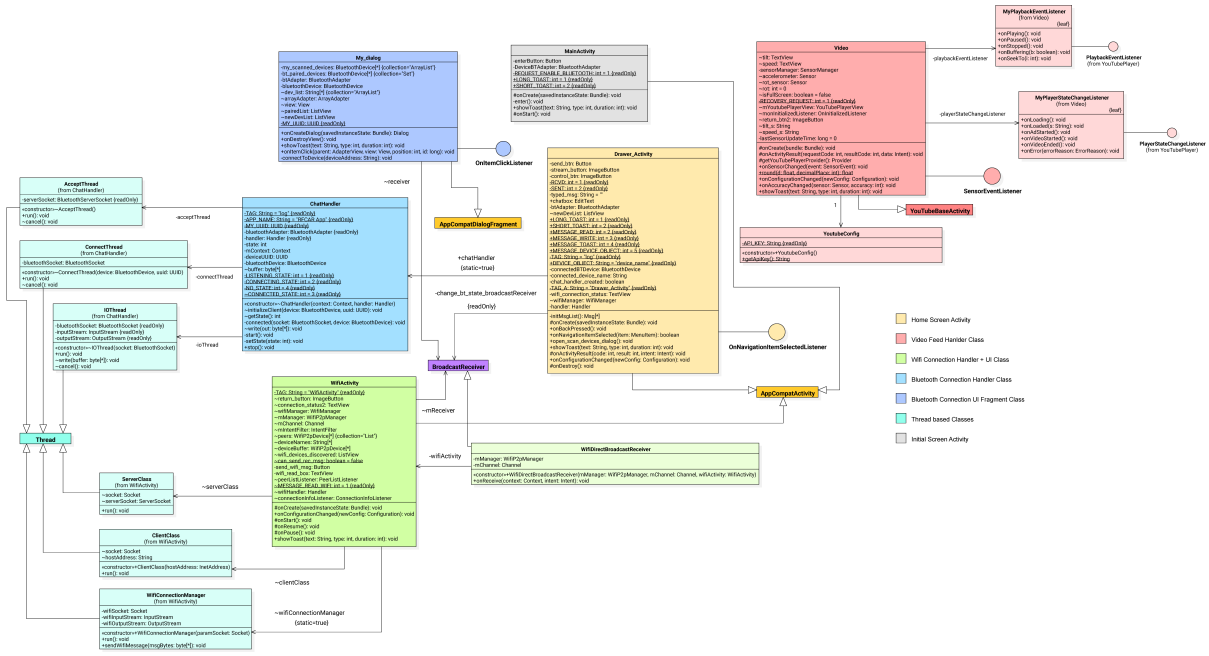

Figure 5.46.: Smartphone class diagram augmented in Appendix B.1

by any means, it should exist an immediate reconnection to the RVVS. The notifications presented to the user should indicate the current state of the system, which means displaying informative messages, like successful connections but also alert messages like the ones displayed in figure 5.50.

5.5. Hardware/Software mapping

In the hardware/software mapping activity, the functionality associated to a software component is mapped to the hardware responsible for executing it. It is represented by a UML deployment diagram used to depict the relationship between run-time components and nodes. Components are self-contained entities that provide services to other components or actors, e.g., the **RVVS_App** in Fig. 5.51. A node is a physical device or an execution environment in which components are executed, such as a desktop computer, or **myLinux** in Fig. 5.51, represented by boxes containing component icons. Furthermore, a node can contain another node, for example a device can contain an execution environment such as virtual machine.

The UML deployment diagram for the RFCAR system is depicted in Fig. 5.51, where the two nodes — Smartphone and LinuxVM — are represented in turquoise and orange, respectively. The ball-and-socket joint identifies the provided and required interfaces, respectively. Thus, the **RVVS_app** provides wireless communication interfaces that the **Android_app** can use, namely Wi-Fi and GPRS . Furthermore, it highlights

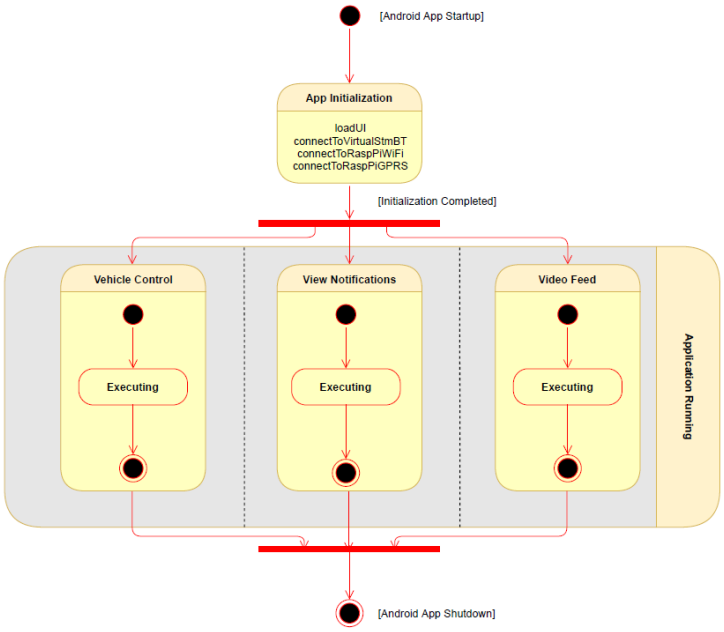

Figure 5.47.: Overall system behaviour diagram.

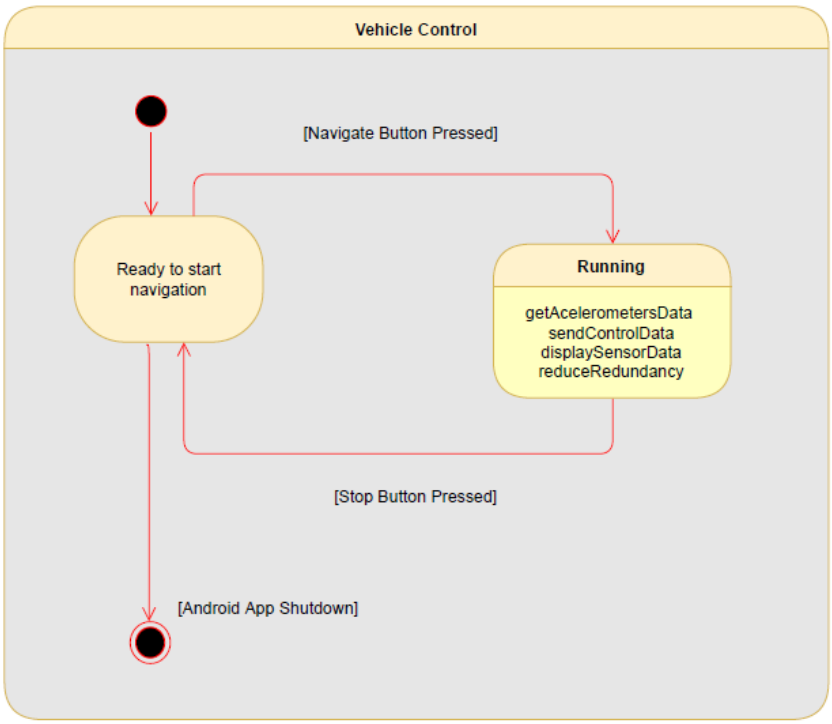

Figure 5.48.: Vehicle control feature diagram.

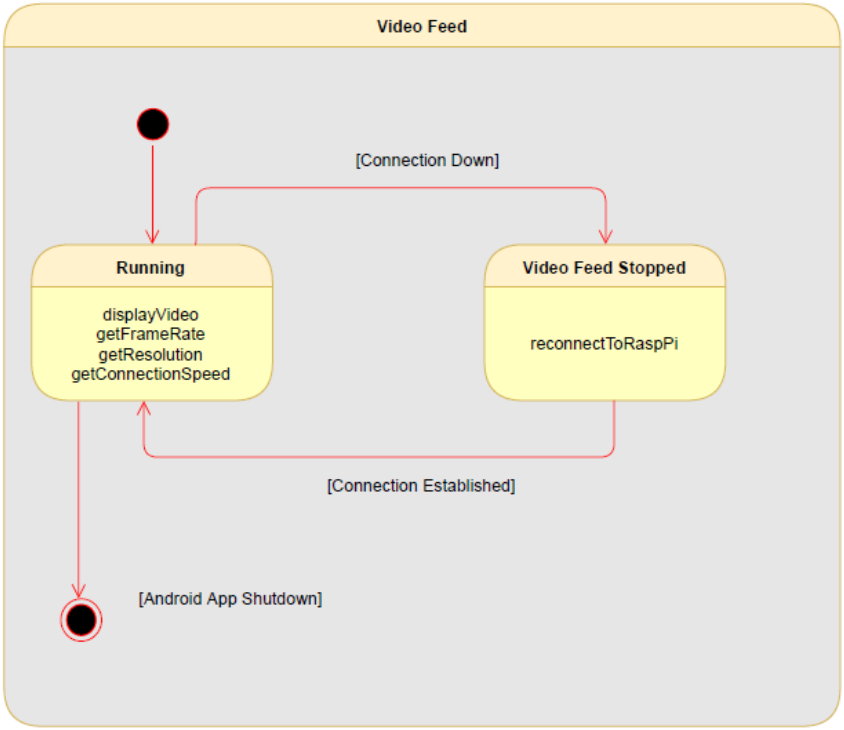

Figure 5.49.: Video feed feature diagram.

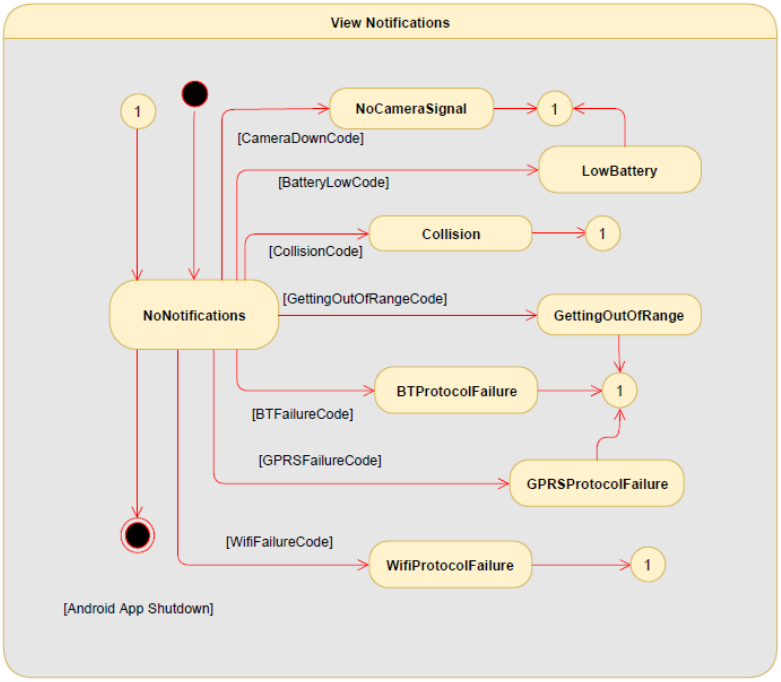

Figure 5.50.: Notification feature diagram.

the client-server software pattern, with the RVVS_app and NVS_app serving the requests (servers) that the Android_app requests (client). The Android_app communicates with the NVS_app via Bluetooth, or, if the communications fail, through any of the available communications channels for the RVVS_app that forwards that request for NVS_app.

Lastly, it should be noted that in a real-world scenario the NVS_app and RVVS_app are mapped to a different hardware, which is also distinct between them, e.g., the former in an STM32 and the latter in a Raspberry Pi. However, in the virtualized environment, and ideally, they represent two distinct processes that must communicate via an IPC mechanism, e.g., sockets. Nonetheless, to ease and speed up the development it is perfectly acceptable to implement both processes into the same application in the early development phase.

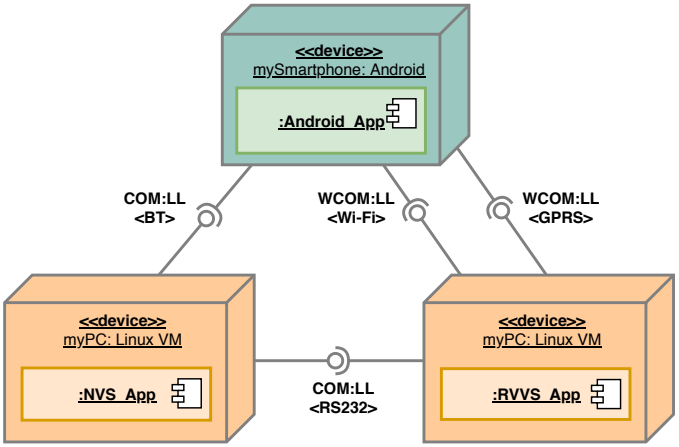

Figure 5.51.: RFCAR Deployment diagram

6. Implementation

In the implementation phase, the solution developed in the various domains is implemented into the target platforms, accordingly to the system design specification. In this chapter is presented the implementation for the various domains and subsystems identified.

6.1. Navigation Virtual Subsystem

6.1.1. Control

6.1.1.1. Implementation on STM32

In a first iteration of the implementation and subsequent testing, the STM32 was used, by implementing the control algorithm in equation 5.14 in a program designed to create command variables through simulated measurements of the desired control variables, the command variables would then act upon the simulated system model in equation 5.10 that would then return the measured variables for the controller to use; thus the real system's controller and behaviour can be simulated. The command variables and control variables are placed on a buffer for each iteration and at the end of the simulation the values are sent to an interface for analysis.

6.1.1.2. STM32 Program

The program utilizes two timer triggered ISRs with the same period, that of fifty miliseconds. However the system response ISR where the command variables shall act upon the system is offset by 500 ticks thus ensuring that the ISRs shall never be triggered at the same time whilst still having the same period.
Starting with the system ISR (List. 6.1), firstly obtaining the psi required for the model, which is done integrating psidot, which, in discrete time, is done through the summation of the product of the current psidot and the sampling time added to the summation of all previous psidot multiplied by the sampling time (this latter summation is called aux_psi). Then measuring the velocity and, with this information, calculate the

next x and y components of the linear velocity of the rover. After that integrate nVx and nVy the same way psi was, and obtaining the position of the car on the x and y axis; followed by obtaining Vr_m and Vl_m through the norm.

```
psi=psidot*0.05+aux_psi;
V_m= Uvr/2 + Uvl/2;
nVx = V_m* cos(psi) - L/2 * V_ref/L * teta * sin(psi);
nVy = V_m* sin(psi) - L/2 * V_ref/L * teta * cos(psi);
x=nVx*0.05 +aux_x;
y=nVy*0.05+aux_y;
aux_x=x;
aux_y=y;
aux_psi=psi;
norm=sqrt(pow(nVx,2)+pow(nVy,2));
psidot=V_m/L * teta;
Vr_m = norm + teta*V_m/2;
Vl_m = norm - teta*V_m/2;
x_buffer[uindex]=x;
y_buffer[uindex]=y;
```

Listing 6.1: System model

Next is the implementation of the controller. Firstly the obstacle avoidance (List. 6.2), it is done through a sensor_buffer, which, whenever one of the nine sensors (with a field angle of $\frac{2\pi}{9}$ rads) detects a nearby obstacle, a one will be placed on the index it is related to (the sensor that detects from 0 rad to $\frac{2\pi}{9}$ rad is related to index one then from $\frac{2\pi}{9}$ rad to $\frac{4\pi}{9}$ rad is index two and so on until 2π rad) from that buffer the controller interprets the angles which are not permitted, this is done thorough a "for" cycle that searches for ones on the buffer and through the corresponding index calculates the boundaries of the unpermitted angles; should the currently desired variation of steering angle (psidot) lead towards anywhere within the boundaries of those angles, the reference variables are set to zero and the controller will stop the car.

```
for(i=1;i<=9;i++)
    {
        if(sensor_buffer[i]==1 && ((psidot<=i*(3.14*2/9)) && (psidot>=(i-1)*(3.14*2/9)) &&
            vflag==0))
        {
            Vr_ref=0;
            V_ref=0;
            Vl_ref=0;
```

```
            vflag=1;
        }
    }
    if(!vflag)
    {
    V_ref=1;
    Vr_ref = V_ref + teta*V_ref/2;
    Vl_ref = V_ref - teta*V_ref/2;
    }
```

Listing 6.2: Obstacle Avoidance Algorithm

The controller (List. 6.3) follows the equation 5.14, as previously stated, therefore it was merely converted without a kd; for the optimal control obtained in previous chapters was with kd=0.

```
    Uvr = Uvr_minus1 + Kp*(Vr_minus1-Vr_m) + Ki*(Vr_ref-Vr_m);  //PID_r
    Uvl = Uvl_minus1 + Kp*(Vl_minus1-Vl_m) + Ki*(Vl_ref-Vl_m);  //PID_l

    Uvl_minus1=Uvl;
    Uvr_minus1=Uvr;

    Vr_minus1=Vr_m;
    Vl_minus1=Vl_m;

    ur_buffer[uindex]=Uvr;
    ul_buffer[uindex]=Uvl;
    uindex++;
```

Listing 6.3: Controller

6.1.2. Thread Mapping

With the control groundwork laid out in section 6.1.1.1, its thread mapping and that of the simulation can be devised following the flowchart in Fig. 6.2.

```
void controlThread(OS::Thread* thread, void* arg) {

//////////////////////////////// Motor implementation example
    ////////////////////////////////
```

```
  // Configure pwm to have an update rate of 2ms
  IO::Config config_pwm = {.update_period = 2, nullptr, nullptr};
  // Configure pulse counter to have a sample rate of 10ms
  IO ::Config config_pc = {.update_period = PC_UPDT_PERIOD, (IO::ConvCpltCallback*)&
      pcCallback, nullptr};

  IO::Entity<IO::MOTOR> motor_left(&config_pwm, &config_pc, IO::Entity<IO::MOTOR>::LEFT);
  IO::Entity<IO::MOTOR> motor_right(&config_pwm, &config_pc, IO::Entity<IO::MOTOR>::RIGHT);


//
    ////////////////////////////////////////////////////////////////////////////////////////////////////



//////////////////////////////// Sensor implementation ////////////////////////////////////

  // Configure sensor to have an update rate of 10ms
  IO::Config config_sensor = {.update_period = 10, (IO::ConvCpltCallback*)&irsensorCallback,
      nullptr};

  IO::Entity<IO::IR_SENSOR> sensor1(&config_sensor,FRONT_LEFT);
  IO::Entity<IO::IR_SENSOR> sensor2(&config_sensor,FRONT_CENTER);
  IO::Entity<IO::IR_SENSOR> sensor3(&config_sensor,FRONT_RIGHT);
  IO::Entity<IO::IR_SENSOR> sensor4(&config_sensor,RIGHT_FRONT);
  IO::Entity<IO::IR_SENSOR> sensor5(&config_sensor,RIGHT_BACK);
  IO::Entity<IO::IR_SENSOR> sensor6(&config_sensor,BACK_RIGHT);
  IO::Entity<IO::IR_SENSOR> sensor7(&config_sensor,BACK_LEFT);
  IO::Entity<IO::IR_SENSOR> sensor8(&config_sensor,LEFT_BACK);
  IO::Entity<IO::IR_SENSOR> sensor9(&config_sensor,LEFT_FRONT);

//
    ////////////////////////////////////////////////////////////////////////////////////////////////////



  ////////////////////////////////////////////// CONFIGURE STREAMS
      ////////////////////////////////////////////////////////////////
  //configure commands and request streams rapsberry and smartphone
  COM::Manager::CreateDevice Smartphone(COM::BLUETOOTH  ,COM::CLIENT,4);
  COM::Manager::CreateDevice Raspberry(COM::BLUETOOTH,COM::SERVER,1);

  COM::Stream::Config config_stream_sp={&Smartpone,STREAM_ID_SMARTPHONE_COMMAND,(COM::Stream
```

```
    ::ParserCallback*) &parseCallback(),COM::BIDIRECTIONAL};
  COM::Stream::Config config_stream_rasp={&Raspberry,STREAM_ID_RASPBERRY_COMMAND,(COM::Stream
    ::ParserCallback*) &parseCallback(),COM::BIDIRECTIONAL};

  COM::Manager::createStream stream_sp(&config_stream_sp,5);
  COM::Manager::createStream stream_rasp(&config_stream_rasp,5);
//
    ////////////////////////////////////////////////////////////////////////////////////////////


  char temp_buffer[50];
  float distances[9];
  float angles[2];

  float ref_theta;
  float ref_speed;

  float out_pwm[2];

//CONTROL VARS BEGIN
float nUvr=0,nUvl=0, nVx=0,nVy=0,Vr_m=0,Vl_m=0,Vr_minus1=0, Vl_minus1=0;
float Uvl=0,Uvr=0, Vr_ref=0, Vl_ref=0, Uvl_minus1=0, Uvr_minus1=0;
int vflag=0;

float L=WHEELBASE;
float Kp=0.05;
float Ki=0.1;
//CONTROL VARS END


  thread->sleepFor(5);
  thread->keepCurrentTimestamp();

  while(1) {

    vflag=0;

    //SEND CMD REQUEST TO SMARTPHONE
    COM::Manager::queueMSG(REQUEST_COMMAND,&stream_sp);

    //REQUESTS COMMAND TO RASPBERRY PI
    COM::Manager::queueMSG(REQUEST_COMMAND,&stream_rasp);
```

```
    //SLEEP FOR 38 MS
    thread->sleepUntilElapsed(38);

    //FETCHING IR SENSORS DISTANCES
83  distances[0] = sensor1.inputDistance();
    distances[1] = sensor2.inputDistance();
    distances[2] = sensor3.inputDistance();
    distances[3] = sensor4.inputDistance();
    distances[4] = sensor5.inputDistance();
88  distances[5] = sensor6.inputDistance();
    distances[6] = sensor7.inputDistance();
    distances[7] = sensor8.inputDistance();
    distances[8] = sensor9.inputDistance();

93  for(i=0;i<9;i++){
      if(distances[i]<40/*cm*/){
        distances[i]=1;
      }
    }
98 //  attribution
    motor_left.inputshaftAngle();
    motor_right.inputshaftAngle();

    //SLEEP FOR 2 MS
103 thread->sleepUntilElapsed(40);

    // CONTROL RULE
    psi_mutex.lock();
    for(i=0;i<9;i++)
108     {
          if(distances[i]==1 && ((psidot<=(i+1)*(3.14*2/9)) && (psidot>=i*(3.14*2/9))))
          {
          Vr_ref=0;
        V_ref=0;
113         Vl_ref=0;
        teta=0;
        vflag=1;
          }
     }
118 if(!vflag)
    {
    Vr_ref = V_ref + teta*V_ref/2;
      Vl_ref = V_ref - teta*V_ref/2;
```

```
    }
123 psi_mutex.unlock();


    Uvr = Uvr_minus1 + Kp*(Vr_minus1-Vr_m) + Ki*(Vr_ref-Vr_m);  //PID_r
      Uvl = Uvl_minus1 + Kp*(Vl_minus1-Vl_m) + Ki*(Vl_ref-Vl_m);  //PID_l

    Uvl_minus1=Uvl;
    Uvr_minus1=Uvr;

    Vr_minus1=Vr_m;
133 Vl_minus1=Vl_m;

    out_pwm[IO::LEFT]   = (Uvl/MOTOR_MAX_ABS_VOLT)*100;
    out_pwm[IO::RIGHT]  = (Uvr/MOTOR_MAX_ABS_VOLT)*100;

138 thread->sleepUntilElapsed(50);
    thread->keepCurrentTimestamp();

    motor_left.outputPulseWidth(out_pwm[IO::LEFT]);
    motor_right.outputPulseWidth(out_pwm[IO::RIGHT]);


  }
}
```

Listing 6.4: Control Thread

Firstly the motors were configured (lines 6 through 13) with a callback that will update the measured linear velocity of each wheel (List. 6.5). Afterwards, the nine infrared odometric sensors were configured (lines 19 through 30), each sensor is configured with a callback (List. 6.6) that will update the distance to the nearest obstacle every update period, devised as 10 miliseconds in this instance. This callback checks the value read by the sensors and through a linear interpolation of the voltage/distance curve of the sensor, Fig. 6.1, the accurate distance in cm between the rover and the nearest obstacle can be measured. Next the streams for the communication between the navigation subsystem and the smartphone, along with the latter and the raspberry pi (lines 33 through 43), the callbacks in these streams (List. 6.8) are always updating the velocity and angle references, that should be altered whenever a command is received. Afterwards the navigation subsystem requests commands, fills an array with the distances for the obstacle avoidance algorithm to use and then makes use of the control algorithm established in section 6.1.1.1.

```
void pcCallback(number value, void* p) {

  IO::Entity<IO::MOTOR>* p_motor = reinterpret_cast<IO::Entity<IO::MOTOR>*>(p);
  float last_angle = p_motor->inputShaftAngle();
  float angle;
  float omega;
  uint32_t pulses = value._uint32;

  angle=(last_angle + PULSE_ANGLE * pulses)%360;
  omega=(angle-last_angle)/PC_UPDT_PERIOD;
  V_m=omega*WHEEL_RADIUS;

  psi_mutex.lock();
  Vr_m= V_m + WHEELBASE/2 * omega;
  Vl_m= V_m - WHEELBASE/2 * omega;
  psi_mutex.unlock();
  p_motor->setShaftAngle(angle);
}
```

Listing 6.5: Pc Callback

```
void irsensorCallback(number value, void* p){

  IO::Entity<IO::IR_SENSOR>* p_sensor = reinterpret_cast<IO::Entity<IO::IR_SENSOR>*>(p);
  float distance_temp;
  value *= 3.3;

  //LINEAR INTERPOLATION OF THE SENSOR'S VOLT/DISTANCE CURVE
  if(value>=2.5 && value<=3.1)
  {
    distance_temp= 20/3 *(4-value);
    p_sensor->setDistance(distance_temp);
  }
  else if(value>=1.4 && value<2.5);
  {
    distance_temp=10/-1.1 * (value-3.6);
    p_sensor->setDistance(distance_temp);
  }
  else if(value>=0.9 && value<1.4)
  {
    distance_temp=20*(2.4-value);
    p_sensor->setDistance(distance_temp);
```

```
22    }
      else if(value >=0.75 && value <=0.9)
      {
        distance_temp=200/3 * (1.35-value);
        p_sensor->setDistance(distance_temp);
27    }
      else if(value >=0.6 && value <=0.75)
      {
        distance_temp=200/3 * (1.2-value);
        p_sensor->setDistance(distance_temp);
32    }
      else if(value >=0.5 && value <=0.6)
      {
        distance_temp=100 * (1.1-value);
        p_sensor->setDistance(distance_temp);
37    }
      else if(value >=0.45 && value <=0.5)
      {
        distance_temp=200 * (0.8-value);
        p_sensor->setDistance(distance_temp);
42    }
      else if(value <0.45)
      {
        distance_temp=0.8;
        p_sensor->setDistance(distance_temp);
47    }
  }
```

Listing 6.6: Ir Sensors Callback

```

2 void parseCallback(uint32_t buffer_size, uint8_t* buff_ptr) {

      psi_mutex.lock();

      teta = (float)*buff_ptr;
7     V_ref = (float)*(buff_ptr+sizeof(float));
      V_ref=(V_ref/100)*6;
      teta=(teta/100)*0.5;

      psi_mutex.unlock();
12 }
```

Listing 6.7: Parse Callback

Afterwards the simulation thread merely makes use of the system model already used in the aforementioned section 6.1.1.1.

```
void simulationThread(OS::Thread* thread, void* arg) {

float psi=0,x=0,y=0, aux_x=0, aux_y=0, aux_psi=0;
  while(1) {

  thread->keepCurrentTimeStamp();

  psi_mutex.lock();
  psi=psidot*0.05+aux_psi;
  aux_psi=psi;
  nVx = V_m* cos(psi) - L/2 * V_ref/L * teta * sin(psi);
    nVy = V_m* sin(psi) - L/2 * V_ref/L * teta * cos(psi);
  psi_mutex.unlock();
  x=nVx*0.05 +aux_x;
  y=nVy*0.05+aux_y;
  aux_x=x;
  aux_y=y;

  psi_mutex.lock();
    psidot=V_m/L * teta;
  psi_mutex.unlock();

  thread->sleepUntilElapsed(10);
  }
}
```

Listing 6.8: Simulation Thread

Whenever the program needs to use global variables it makes use of a mutual exclusion object (mutex) that ensures that those variables cannot be shared simultaneously, ensuring a smooth and error avoiding execution.

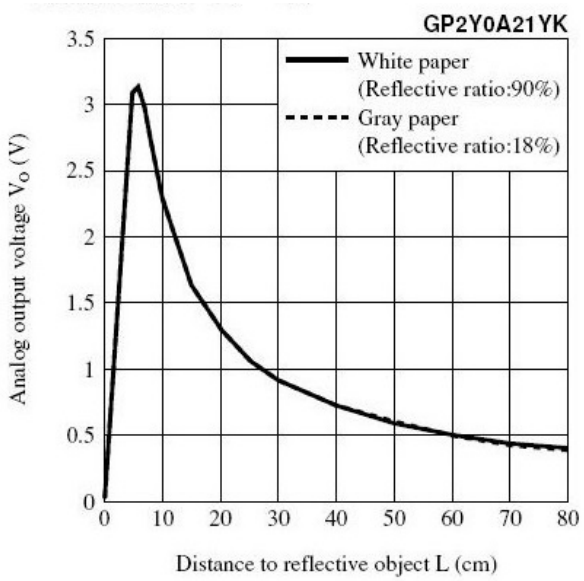

Figure 6.1.: Voltage/Distance Curve of Sensor

6.1.3. Stack

The chosen language of implementation was C++ due to its low-level nature and backwards compatibility with the C programming language, which is what most Board Support Packages (BSP's) are written in. The fact that it was chosen over C, though, is mostly due to its object-oriented nature, that translates very easily from the package and entity-oriented design, UML class diagrams and package diagrams and makes for cleaner, more easily maintainable code. To mitigate the effects of the C++ runtime on performance, inheritance was thoroughly avoided, class templates and template specializations being the alternative to this method, as seen in the IO::Entity and COM::LL subpackages. This allows us to trade storage footprint for better runtime performance. Furthermore, the use of C-style types was preferred whenever possible in the higher-level layers, practicality As forementioned, the initial approach consisted of the three-way interaction depicted in the RFCAR deployment diagram, in section 5.5 - figure 5.51. However, due to the extraordinary circumstances the need to virtualize the navigation and remote vision subsystems rose. Specifically, both were simulated on a virtual machine with an **18.04 Lubuntu image**, a Ubuntu-based lightweight Linux distribution. This implies that the High-level Hardware Abstraction layer's implementation should be done using Linux-specific APIs and libraries. As such, in an effort to meet the established deadlines the liberty of using C++'s Standard Template Language (STL) was taken. This stability and convenience was also a strong motivation for choosing C++ as the language of implementation.

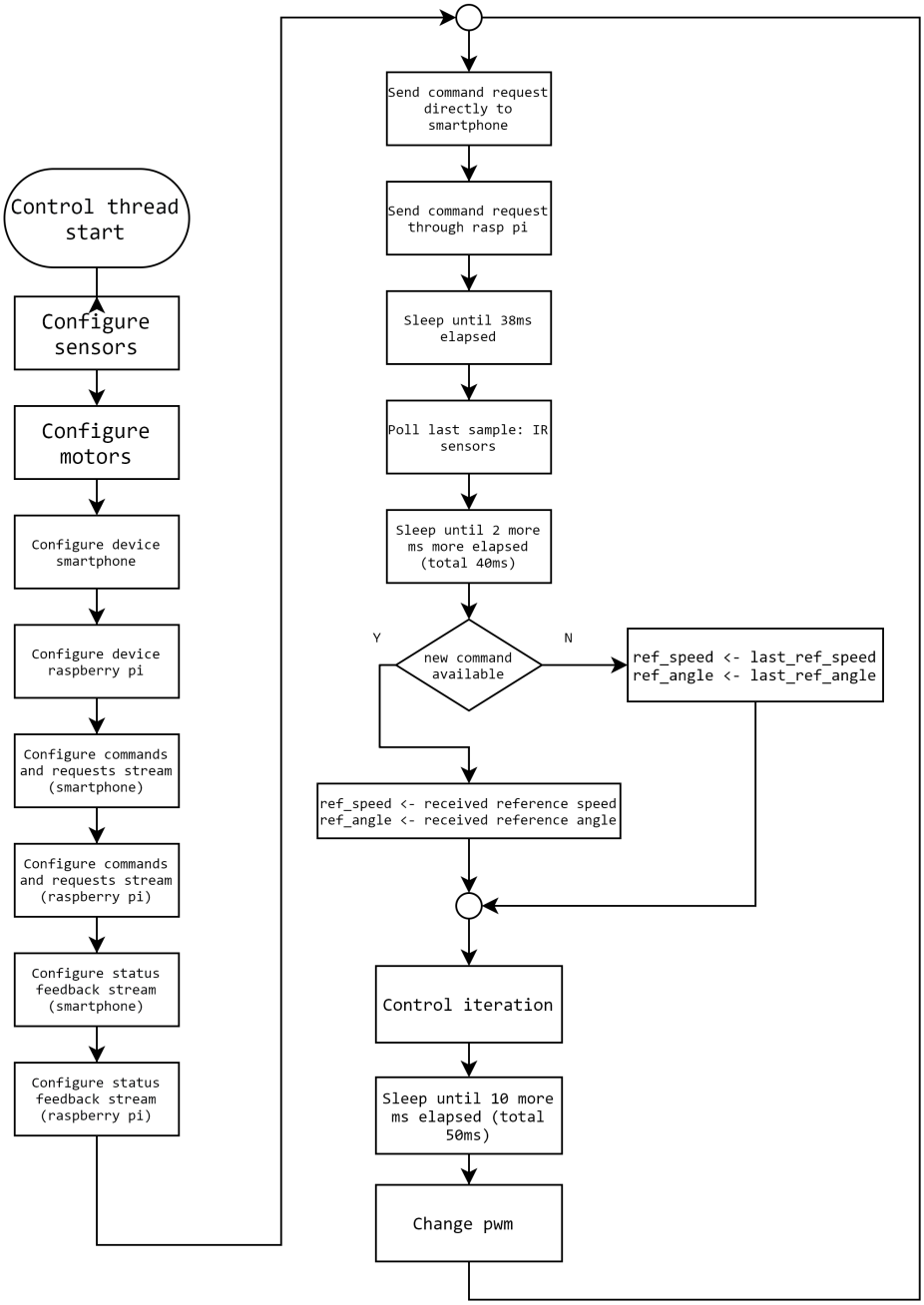

Figure 6.2.: ControlThreadFlowchart

6.1.3.1. IO: Input/Output Package

As foresaid in section 5.1.2.3, the IO package includes the IO Entity and IO Link subpackages. The IO_Entity header file represented in listing **??**, includes a declaration of a generic template (within IO namespace) that allows the specialization of GPIO objects (belonging to the IO Link package) in physical entities like a motor or an infrared sensor, in this case. Each specialization of an Entity object requires the definition of the targeted constructor. There, one can decide which type the entity is going to take and its position within the rover model whilst configuring the entity attending to its GPIO requirements. This is done through preemptively defined GPIO-targeted modes in **??**.

The second subpackage, IO Link, is listed in **??** and 6.9. This generic package interacts directly with the virtualization of the machine itself (sensors and actuators) through timed input and output in binary files while also defining targeted GPIO modes for certain entities. The IO interface in listing **??** allows the definition of transversal structures and typedefs for the IO-related packages.

```
#include "IO_GPIO.hpp"

#ifdef _LINUX_
namespace IO {

  GPIO::States GPIO::global_states = {0, 0, 0};

  void* GPIO::timeElapsedCallback(void* ptr) {

    IO::GPIO* gpio_ptr = (IO::GPIO*)ptr;
    number conversion;

    // If mode is input
    if (gpio_ptr->mode < Mode::OUTPUT_PWM) {

      uint32_t line_id;
      std::ifstream conversions_file(gpio_ptr->filename, std::ios::binary | std::ios::in);

      do {
        // Move back in the file the right amount of characters to read the conversion ID and
            the conversion value
        conversions_file.seekg(-(int32_t)(sizeof(line_id) + sizeof(conversion)), std::
            ios_base::end);

        conversions_file.read(reinterpret_cast<char*>(&line_id), sizeof(last_line_id));
        conversions_file.read(reinterpret_cast<char*>(&(conversion._uint8_arr)), sizeof(
            conversion));
```

```
25        } while (line_id == gpio_ptr->last_line_id);

          conversions_file.close();

          gpio_ptr->last_line_id = line_id;
30        gpio_ptr->insertNewConversion(conversion);
          // Else if mode is output
        } else {

          gpio_ptr->fetchLastConversions(conversion);

          std::ofstream conversions_file(gpio_ptr->filename, std::ios::binary | std::ios::out |
              std::ios::app);

          if (!conversions_file.is_open()) {
            gpio_ptr->last_error = IO::Error::FAIL_OUTPUT;
40          return nullptr;
          }

          // Praying to the Endian gods
          conversions_file.write(reinterpret_cast<char*>(&(gpio_ptr->last_line_id)), sizeof(
              gpio_ptr->last_line_id));
45        conversions_file.write(reinterpret_cast<char*>(&(conversion)), sizeof(conversion));

          gpio_ptr->last_line_id += 1;
          conversions_file.close();
        }

        // Call conversion complete callback
        if (gpio_ptr->conv_cplt_callback != NULL)
          (*(gpio_ptr->conv_cplt_callback))(conversion, gpio_ptr->callback_arg);

55      gpio_ptr->last_error = IO::OK;

        return nullptr;
      }

60    GPIO::GPIO() : timer(&timeElapsedCallback, this) {

        this->id = GPIO::grabAvailableObject();

        if (id == IO_GPIO_MAX_OBJECT_COUNT) {
65        this->last_error = OBJECT_UNAVAILABLE;
```

```
    } else {
      this->filename = std::string(IO_FOLDER) + std::string("gpio") + std::to_string(id) + ".
          bin";
      std::ofstream file(this->filename, std::ios::trunc);
      file.close();
      this->last_error = OK;
    }

}


Error GPIO::configure(Config* config, Mode mode) {

    this->mode = mode;

    if (config == nullptr) {
      last_error = NULL_CONFIG;

    } else {

      update_period_ms = config->update_period;
      conv_cplt_callback = config->conv_cplt_callback;

      markConfigured(this);
      last_error = OK;
    }

    return last_error;
}


void GPIO::run() {

    markRunning(this);
    timer.setCounter(update_period_ms);
    timer.setAutoReload(update_period_ms);
    timer.start();

    last_error = OK;
}
```

```
Error GPIO::insertNewConversion(number value) {

  if (mode < Mode::OUTPUT_PWM)
    this->last_error = INVALID_OPERATION;

  else if (this->conversion_buffer.push(value) != MEM::OK)
    last_error = BUFFER_FULL;

  else
    last_error = OK;

  return this->last_error;
}


Error GPIO::fetchLastConversions(number& value) {

  // Fetch only last conversion to guarantee that synchronism is maintained
  while (conversion_buffer.pop(&value) != MEM::EMPTY);
  last_error = OK;
  return this->last_error;
}

}

#endif
```

Listing 6.9: IO_GPIO Source

6.1.3.2. COM: Communications Package

6.1.3.2.1. COM::LL

The COM::LL package possesses 3 templates, depending on which entities are communicating (client or server) and which protocol is being used to communicate (serial or Bluetooth). The main difference between the three lies in how they communicate, the serial protocol uses local sockets (AF_LOCAL) to be able to communicate with the Remote Vision Subsystem in the same machine whilst Bluetooth uses a connection to a serial port that is associated with the Bluetooth adapter of the host device and any communication via object of this class is done through this port. The package provides means of communication via reading and writing on buffers and is implemented with several error verifications and feedback messages indicating

either the reason for errors or that the operation was a success. The error verifying during the creation of the of the objects makes sure of the existence of the requested port, followed by ensuring that no two objects access the same port. Each time one of the available means is used for communicating, it is always verified if the client-server connection has already been established and if not it doesn't permit sending or reading messages.

```
#include "COM_LL.hpp"

#ifdef _LINUX_

#include <iostream>
#include <sstream>
#include <atomic>
#include <algorithm>
#include <thread>
#include <cerrno>
#include <unistd.h>

#include <termios.h> // Contains POSIX terminal control definitions

// AF_UNIX is used for communication between processes in the same machine efficiently
#define SERIAL_SOCKET_FAMILY  AF_UNIX
// SOCK_STREAM Provides sequenced, reliable, two-way, connection-based byte streams
#define SERIAL_SOCKET_TYPE    SOCK_STREAM
// Prefix for the serial communication servers
#define SERIAL_SERVERS_PREFIX "serial"
// Default port to be assigned to a serial object when none is provided
#define SERIAL_DEFAULT_PORT   0

// Default port to be assigned to a bluetooth object when none is provided
#define BLUETOOTH_DEFAULT_PORT    0

// Folder where the socket files are hosted
#define SERVERS_FOLDER    "/tmp/servers/"



namespace COM {


  template<Protocol protocol, Role role>
  LL<protocol, role>::LL() {
```

```
  }

  template<Protocol protocol, Role role>
  LL<protocol, role>::~LL() {

  }
}



//////////////////////////////// SERIAL, CLIENT
    ////////////////////////////////////////////////////////////////////////////////

namespace COM {

52  bool LL<SERIAL, CLIENT>::port_occupation[SERIAL_AVAILABLE_PORTS + 1] = {0};

  LL<SERIAL, CLIENT>::LL(int32_t port) {

    this->port = port;

    if (port > 0 && port <= SERIAL_AVAILABLE_PORTS) {

      if (port_occupation[port] == true) {
        this->dead = true;
62      last_error = INVALID_PORT;
        last_error_str = "ERROR[INVALID_CONFIG]: The object was given an valid port number
            that was already taken\n";
      } else {
        this->dead = false;
        last_error = OK;
67      last_error_str = "OK: The object was correctly configured\n";
        port_occupation[port] = true;
      }
    } else {
      this->dead = true;
72    last_error = INVALID_PORT;
      last_error_str = "ERROR[INVALID_CONFIG]: The object was given an invalid port number\n"
          ;
    }

    this->connected = false;
77  this->dead = false;
```

```
    this->connect_socket_fd = -1;
  }

82 LL<SERIAL, CLIENT>::LL() : LL(SERIAL_DEFAULT_PORT) {

  }

  Error LL<SERIAL, CLIENT>::readStr(char * p_buff, uint32_t len) {

    std::unique_lock<std::mutex>lock(this->mutex.native());

    if (!this->connected) {
      last_error = NOT_CONNECTED;
92    last_error_str = "ERROR[NOT_CONNECTED]: Not connected\n";

    } else if (this->dead) {
      last_error = DEAD;
      last_error_str = "ERROR[DEAD]: Object is dead\n";

    } else if (read(this->connect_socket_fd, p_buff, len) < 0) {
      last_error = READ_FAIL;
      last_error_str = "ERROR[READ_FAIL]: Failed reading from socket\n";

102   } else {
      last_error = OK;
      last_error_str = "OK: Read string\n";
    }

107   return last_error;
  }

  Error LL<SERIAL, CLIENT>::writeStr(const char * p_buff, uint32_t len) {

112   std::unique_lock<std::mutex>lock(this->mutex.native());

    if (!this->connected) {
      last_error = NOT_CONNECTED;
      last_error_str = "ERROR[NOT_CONNECTED]: Not connected\n";

    } else if (this->dead) {
      last_error = DEAD;
      last_error_str = "ERROR[DEAD]: Object is dead\n";
```

```
122     } else if (write(this->connect_socket_fd, p_buff, len) < 0) {
          last_error = WRITE_FAIL;
          last_error_str = "ERROR[WRITE_FAIL]: Failed writing to socket\n";

        } else {
127       last_error = OK;
          last_error_str = "OK: Write string\n";
        }

        return last_error;
132   }

      bool LL<SERIAL, CLIENT>::openConnection() {

        std::string server_path = SERVERS_FOLDER SERIAL_SERVERS_PREFIX + std::to_string(this->
            port);
137     std::unique_lock<std::mutex>lock(this->mutex.native());

        // Check if the socket is already open
        if (this->connect_socket_fd > 0) {
          last_error = ALREADY_OPEN;
142       last_error_str = "ERROR[ALREADY_OPEN]: A connection with this server or another one is
              already open\n";
          return this->connected;
        }

        // Attempt to open socket
147     this->connect_socket_fd = socket(SERIAL_SOCKET_FAMILY, SERIAL_SOCKET_TYPE, 0);

        // If this->connect_socket_fd == 1, it means that the socket could not be open and thus
            the file descriptor is invalid
        if (this->connect_socket_fd < 0) {
          last_error = OPEN_FAIL;
152       last_error_str = "ERROR[OPEN_FAIL]: Failed to open socket\n";
          return this->connected;
        }

        std::fill_n((char*)&this->server_address, sizeof(this->server_address), 0);

        // Set server socket family and socket file path
        this->server_address.sun_family = SERIAL_SOCKET_FAMILY;
        strcpy(this->server_address.sun_path, server_path.c_str());
```

```
162     // Attempt to establish a connection
        if (connect(this->connect_socket_fd, (struct sockaddr*) &this->server_address, sizeof(
            this->server_address)) < 0) {
          last_error = OPEN_FAIL;
          last_error_str = "ERROR[OPEN_FAIL]: Failed connecting\n";
          return this->connected;
167     }

        this->connected = true;

        last_error = OK;
172     last_error_str = "OK: Open connection\n";

        return this->connected;
      }
    }


    ////////////////////////////// SERIAL, SERVER
        ////////////////////////////////////////////////////////////////////////////////

    namespace COM {

      bool LL<SERIAL, SERVER>::port_occupation[SERIAL_AVAILABLE_PORTS + 1] = {0};

      LL<SERIAL, SERVER>::LL(int32_t port) {

187     this->port = port;

        if (port > 0 && port <= SERIAL_AVAILABLE_PORTS) {

          if (port_occupation[port] == true) {
192         this->dead = true;
            last_error = INVALID_PORT;
            last_error_str = "ERROR[INVALID_CONFIG]: The object was given an valid port number
                that was already taken\n";
          } else {
            this->dead = false;
197         last_error = OK;
            last_error_str = "OK: The object was correctly configured\n";
            port_occupation[port] = true;
          }
```

```
    } else {

      this->dead = true;
      last_error = INVALID_PORT;
      last_error_str = "ERROR[INVALID_CONFIG]: The object was given an invalid port number\n"
          ;
    }

    this->connected = false;
    this->listened = false;

  }

  LL<SERIAL, SERVER>::LL() : LL(SERIAL_DEFAULT_PORT) {

  }

217 Error LL<SERIAL, SERVER>::closeConnection(void) {

    std::unique_lock<std::mutex>lock(this->mutex.native());

    if (this->connected) {

      if (close(this->listen_socket_fd) < 0) {
        last_error = CLOSE_FAIL;
        last_error_str = "ERROR[CLOSE_FAIL]: Failed closing connection\n";
        return CLOSE_FAIL;
227   }
    }

    this->connected = false;
    last_error = OK;
232 last_error_str = "OK: Close connection\n";

    return last_error;
  }


  Error LL<SERIAL, SERVER>::listenConnection(void) {

    std::string server_path = SERVERS_FOLDER SERIAL_SERVERS_PREFIX + std::to_string(this->
        port);
    std::string command = "mkdir -p " SERVERS_FOLDER;
```

```
242     std::unique_lock<std::mutex>lock(this->mutex.native());

        // There is already a socket file descriptor atributed
        if (this->listen_socket_fd > 0) {
          last_error = ALREADY_OPEN;
247       last_error_str = "ERROR[ALREADY_OPEN]: There is already a socket file descriptor
              atributed\n";
          return last_error;
        }

        // Request an endpoint for communication
252     this->listen_socket_fd = socket(SERIAL_SOCKET_FAMILY, SERIAL_SOCKET_TYPE, 0);

        // If the socket fails to open
        if (this->listen_socket_fd < 0) {
          last_error = OPEN_FAIL;
257       last_error_str = "ERROR[OPEN_FAIL]: Failed to open socket\n";
          return last_error;
        }

        std::fill_n((char*)&this->listen_serv_addr, sizeof(this->listen_serv_addr), 0);

        system(command.c_str());

        this->listen_serv_addr.sun_family = SERIAL_SOCKET_FAMILY;
        strcpy(this->listen_serv_addr.sun_path, server_path.c_str());

        // Path should be unlinked before a bind() call
        // https://stackoverflow.com/questions/17451971/getting-address-already-in-use-error-
            using-unix-socket
        unlink(server_path.c_str());
        int opt = 1;
272     if (setsockopt(this->listen_socket_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt,
            sizeof(opt))) {
          last_error = OPEN_FAIL;
          last_error_str = "ERROR[OPEN_FAIL]: Failed to set socket options\n";
          return last_error;
        }

        // Assign the address specified by addr to the socket referred to by the file descriptor
            sockfd
        if (bind(this->listen_socket_fd, (struct sockaddr*) &this->listen_serv_addr, SUN_LEN(&
            this->listen_serv_addr)) < 0) {
```

```
      // Error feedback
282   last_error = OPEN_FAIL;
      last_error_str = "ERROR[OPEN_FAIL]: Failed to bind\n";
      return last_error;
    }

287 if (listen(this->listen_socket_fd, 5) < 0) {

      // Undo binding. Not absolutely necessary as it is done before the 'bind' instruction
          already.
      unlink(server_path.c_str());

292   // Error feedback
      last_error = OPEN_FAIL;
      last_error_str = "ERROR[OPEN_FAIL]: Failed to listen\n";
      return last_error;
    }

    listened = true;
    last_error = OK;
    last_error_str = "OK: Listening for connection\n";

302 return last_error;
  }

  Error LL<SERIAL, SERVER>::acceptConnection(void) {

307 socklen_t clilen;

    // Make sure the listen() operation has been executed
    if (this->listened == false) {
      last_error = ACCEPT_FAIL;
312   last_error_str = "ERROR[ACCEPT_FAIL]: Should listen before accepting\n";
      return last_error;
    }

    clilen = sizeof(this->listen_serv_addr);

    // Accept connection. This blocks the thread's execution until it returns.
    this->connect_socket_fd = accept(this->listen_socket_fd, (struct sockaddr*) &this->
        connect_serv_addr, &clilen);
```

```
    // Check if accept() returned a valid file descriptor
322 if (this->connect_socket_fd <= 0) {
      last_error = ACCEPT_FAIL;
      last_error_str = "ERROR[ACCEPT_FAIL]: Failed acceptance (invalid file descriptor)\n";
      return last_error;
    }

    // Signal connection established
    this->connected = true;
    last_error = OK;
    last_error_str = "OK: Acepted connection\n";

    return last_error;
  }


337 }


////////////////////////////// BLUETOOTH, SERVER
    ////////////////////////////////////////////////////////////////////////////////

342 namespace COM {

  bool LL<BLUETOOTH, SERVER>::port_occupation[BLUETOOTH_AVAILABLE_PORTS + 1] = {0};

  LL<BLUETOOTH, SERVER>::LL(int32_t port) {

    this->port = port;

    if (port > 0 && port <= BLUETOOTH_AVAILABLE_PORTS) {

352   if (port_occupation[port] == true) {
        this->dead = true;
        last_error = INVALID_PORT;
        last_error_str = "ERROR[INVALID_CONFIG]: The object was given an valid port number
            that was already taken\n";
      } else {
357     this->dead = false;
        last_error = OK;
        last_error_str = "OK: The object was correctly configured\n";
        port_occupation[port] = true;
      }
```

```
362     } else {

        this->dead = true;
        last_error = INVALID_PORT;
        last_error_str = "ERROR[INVALID_CONFIG]: The object was given an invalid port number\n"
            ;
367     }

      this->connected = false;
      this->listened = false;

372   }


    Error LL<BLUETOOTH, SERVER>::listenConnection(void) {

377     std::unique_lock<std::mutex>lock(this->mutex.native());
      std::string serial_port = "/dev/ttyS" + std::to_string(port - 1);

      this->serial_fd = open(serial_port.c_str(), O_RDWR);

382     // Check for errors
      if (serial_fd < 0) {
        last_error = OPEN_FAIL;
        last_error_str = "ERROR[OPEN_FAIL]: Failed to open " + serial_port + "\n";
        return last_error;
387     }

      // Create new termios struct, we call it 'tty' for convention
      struct termios tty;
      memset(&tty, 0, sizeof tty);

      // Read in existing settings, and handle any error
      if(tcgetattr(serial_fd, &tty) != 0) {
        last_error = OPEN_FAIL;
        last_error_str = "ERROR[OPEN_FAIL]: Failed to get port attributes\n";
397       return last_error;
      }

      tty.c_cflag &= ~PARENB; // Disable parity
      tty.c_cflag &= ~CSTOPB; // One stop bit
402     tty.c_cflag |= CS8;   // 8 bits per bytetty.c_cflag |= CS8; // 8 bits per byte
      tty.c_cflag &= ~CRTSCTS; // Disable RTS/CTS hardware flow control
```

```
    tty.c_cflag |= CREAD | CLOCAL; // Turn on READ & ignore ctrl lines

    tty.c_lflag &= ~ICANON; // DIsable canonical mode (not wait for \n)
    tty.c_lflag &= ~ECHO; // Disable echo
    tty.c_lflag &= ~ECHOE; // Disable erasure
    tty.c_lflag &= ~ECHONL; // Disable new-line echo
    tty.c_lflag &= ~ISIG; // Disable interpretation of INTR, QUIT and SUSP

    tty.c_iflag &= ~(IXON | IXOFF | IXANY); // Turn off s/w flow ctrl
    tty.c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP|INLCR|IGNCR|ICRNL); // Disable any special
        handling of received bytes

    tty.c_oflag &= ~OPOST; // Prevent special interpretation of output bytes (e.g. newline
        chars)
    tty.c_oflag &= ~ONLCR; // Prevent conversion of newline to carriage return/line feed

    tty.c_cc[VTIME] = 0;
    tty.c_cc[VMIN] = 0;

    // Set in/out baud rate to be 115200
    cfsetispeed(&tty, B115200);
    cfsetospeed(&tty, B115200);

    // Save tty settings, also checking for error
    if (tcsetattr(serial_fd, TCSANOW, &tty) != 0) {
      last_error = OPEN_FAIL;
      last_error_str = "ERROR[OPEN_FAIL]: Failed to Set desired port attributes\n";
      return last_error;
    }

    listened = true;
    last_error = OK;
    last_error_str = "OK: Listening for connection\n";

    return last_error;

  }

  Error LL<BLUETOOTH, SERVER>::acceptConnection(void) {

    std::unique_lock<std::mutex>lock(this->mutex.native());

    // Make sure the listen() operation has been executed
```

```
    if (this->listened == false) {
      last_error = ACCEPT_FAIL;
      last_error_str = "ERROR[ACCEPT_FAIL]: Should listen before accepting\n";
      return last_error;
    }

    // Signal connection established
    this->connected = true;
    last_error = OK;
    last_error_str = "OK: Acepted connection\n";

    return last_error;
  }
}


#endif    // _LINUX_
```

Listing 6.10: COM_LL

6.1.3.3. OS: Scheduler Package

6.1.3.3.1. OS::Mutex

Has the purpose of managing concurrent access in a way that avoids errors, as previously mentioned in section 5.1.2.5.

```
#include "OS_Mutex.hpp"

#ifdef _LINUX_

#include <cstring>

namespace OS {

  Mutex::Mutex() : self(), owner(empty_thread_id) {}

  Mutex::Mutex(const Mutex& other_mutex) {

    memcpy(&(this->self), &(other_mutex.self), sizeof(this->self));
    this->owner = other_mutex.owner;
  }
```

```
  Mutex::~Mutex() {
    self.unlock();
  }

  bool Mutex::tryLock() {
    if (self.try_lock()) {
      owner = Thread::currentThreadID();
      return true;
    }

    return false;
  }

  void Mutex::lock() {
    self.lock();
    owner = Thread::currentThreadID();
  }

  void Mutex::unlock() {
    if (owner == Thread::currentThreadID()) {
      owner = empty_thread_id;
      self.unlock();
    }
  }

  Mutex::Native& Mutex::native() {
    return self;
  }
}

#endif  // _LINUX_
```

Listing 6.11: OS_Mutex

6.1.3.3.2. OS::SharedMemory

Makes use of the std::list container for storing the thread whitelist, for it requires efficient methods for insertion and removal of elements.

```
#ifndef SHARED_MEMORY_H
#define SHARED_MEMORY_H
```

```
#include "main.hpp"
#include "OS_Mutex.hpp"
#include "OS_Thread.hpp"

#ifdef _LINUX_

#include <list>

namespace OS {

  template<class T>
  class SharedMemory {


  private:  // Private members

    // Linked list of associated threads. Implemented as an STL structure in Windows/Linux
        for simplicity of operation
    std::list<Thread*> threads;

    // Mutex that will help manage access to the memory
    Mutex mutex;

    // Memory region to be managed
    T* self;

    // Thread that has last been authorized to access to the memory
    Thread::ID authorized_thread;

  public:   // Public methods

    // Constructor/default constructor
    SharedMemory(T* memory = nullptr) {
      authorized_thread = Thread::ID();
      self = memory;
    }

    // Destructor
    ~SharedMemory() {}

    // Add thread to list of threads with access to the memory
    inline void addThread(Thread* thread) {
```

```
    threads.push_back(thread);
    threads.unique();
  }

  // Request access to memory. This has to be done by a thread in order to access the
      memory and when it is done, only release()
  // may free it for another one
  inline bool grab() {
    mutex.lock();
    if (authorized_thread == empty_thread_id) {
      for (auto thread : threads) {
        if (thread->ownID() == Thread::currentThreadID()) {
          // mutex.lock();
          authorized_thread = Thread::currentThreadID();
          return true;
        }
      }
    }
    mutex.unlock();

    return false;
  }

  // grab() but in a non-blocking fashion
  inline bool grab_non_blocking() {
    for (auto thread : threads) {
      if (thread->ownID() == authorized_thread) {
        if (mutex.tryLock()) {
          authorized_thread = Thread::currentThreadID();
          return true;
        }
        break;
      }
    }
    // return false if the thread is not authorized, the mutex is locked or the function is
        not implemented for the specific platform
    return false;
  }

  // Releases the memory to be used by another thread. Only the authorized thread can do
      this.
  inline bool release() {
    if (Thread::currentThreadID() == authorized_thread) {
```

```
        authorized_thread = empty_thread_id;
        mutex.unlock();
        return true;
      }
      return false;
    }

    // Controlled access to the memory
    inline T* data() {

      // Return immediately if no thread has requested access
      if (authorized_thread == empty_thread_id)
        return nullptr;

      // If memory access has been requested and granted to the current thread, return the
          pointer to the protected memory
      if (authorized_thread == Thread::currentThreadID())
        return self;

      // Returns nullptr if the current thread has no privilege over the protected memory
      return nullptr;
    }


  };

}

#endif   // _LINUX_

#endif // !SHARED_MEMORY_H
```

Listing 6.12: OS_SharedMemory

6.1.3.3.3. OS::Thread

The execution of the method is controlled by a call to the **run()** method which gives the start signal to the main method guaranteeing that whichever thread called the **run()** method is the parent thread. It permits a precise control of the total time of the thread whilst making sure that the only thread that can control it is itself.

```
#include "OS_Thread.hpp"
#include "CLK.hpp"

#ifdef _LINUX_

#include <iostream>
#include <string>
#include <cstring>


namespace OS {

  const Thread::ID empty_thread_id = Thread::ID();

  Thread::Thread(const char name[20], Method method, void* args, StackSize stack_size,
      Priority priority) :

    self(&(Thread::main_method), this, args), parent_id(std::this_thread::get_id()) {

    start = false;
    this->method = method;
    std::memcpy(this->name, name, 20);

    // In Linux we ignore priority and stack size altogether
    this->priority = priority;
    this->stack_size = stack_size;

    last_timestamp = std::chrono::system_clock::now();
  }

  Thread::~Thread() {
    self.detach();

  }

  Thread::ID Thread::ownID() {
    return self.get_id();

  }

  void Thread::keepCurrentTimestamp() {
    // Check if it is the thread itself calling
    if (std::this_thread::get_id() == self.get_id())
```

```
      last_timestamp = std::chrono::system_clock::now();
  }

46  void Thread::sleepUntilElapsed(uint32_t time_ms) {
    // Check if it is the thread itself calling
    if (std::this_thread::get_id() == self.get_id())
      std::this_thread::sleep_until(this->last_timestamp + CLK::TimeUnit(time_ms));
  }

  void Thread::sleepFor(uint32_t time_ms) {
    // Check if it is the thread itself calling
    if (std::this_thread::get_id() == self.get_id())
      std::this_thread::sleep_for(static_cast<CLK::TimeUnit>(time_ms));
56  }

  void Thread::run() {
    // Check if it is the parent thread calling
    if (Thread::currentThreadID() == parent_id)
61    start = true;
  }

  Thread::ID Thread::currentThreadID() {
    return std::this_thread::get_id();
66  }

  void Thread::join() {
    if (self.joinable())
      self.join();
71  }

  bool Thread::joinable() {
    return self.joinable();
  }

  void Thread::detach() {
    self.detach();
    parent_id = empty_thread_id;
  }

}

#endif   // _LINUX_
```

Listing 6.13: OS_Thread

6.1.3.3.4. OS::Notification

```
#include "OS_Notification.hpp"

#ifdef _LINUX_

namespace OS {

  Notification::Notification() : awaiting_notification(false) {}

  Notification::~Notification() {}


  void Notification::notifyOne() {
    condition.notify_one();
  }

  void Notification::notifyAll() {
    condition.notify_all();
  }

  void Notification::wait() {
    std::mutex mut;
    std::unique_lock<std::mutex> lock(mut);
    condition.wait(lock);
  }
}

#endif   // _LINUX_
```

Listing 6.14: OS_Notification

6.1.3.4. MEM: Memory Structures Package

Both lists provide efficient methods for insertion and removal of elements and completely clearing the list, all of these operations with O(1) complexity.

6.1.3.4.1. MEM::CircularList

It is implemented with recourse to a C-style array to optimize for use as a single-byte data container (e.g. strings of characters in message buffers).

```
#ifndef CIRCULAR_LIST_H
#define CIRCULAR_LIST_H

#include "main.hpp"
#include "MEM.hpp"

#ifdef _LINUX_
#include <list>
#include <mutex>
#include <condition_variable>
#include <iostream>


#define MAX_ALLOWED_SIZE  8192
#define MIN_ALLOWED_SIZE  2

#endif

namespace MEM {

  template <typename T, uint32_t max_size = 0>
  class CircularList {

  private:

#ifdef _LINUX_
    std::mutex list_mutex;          /*! Mutex for use with std::unique_lock in managing
        accesses and operations */
    std::condition_variable list_condition; /*! Condition variable for use in waits (see
        waiting_data: bool) */
#endif

    T list[max_size + 1]; /*! Array for element storage */
    int32_t p_write;    /*! Writing iterator */
```

```
    int32_t p_read;     /*! Reading iterator */

    Error last_error; /*! Result from the execution of the last function */
    bool waiting_data;  /*! True when pop() or popBulk() are waiting for an element in
        BlockingMode::BLOCKING */
    bool dead;      /*! Once dead, the list may no longer be used. Accessing or operating
        upon it will yield Error::DEAD */


    /*! @return Whether or not the list is empty
    */
    inline bool _isEmpty() {
      return (p_write == p_read);
    }


    /*! @return Whether or not the list is full
    */
    inline bool _isFull() {
      return (((p_read - p_write) % (max_size + 1)) == 1);
    }


    /*! @return Number of elements present in the list
    */
    inline uint32_t _size() {
      return ((p_write - p_read) % (max_size + 1));
    }


    /*! @return Number of spaces left in the list
    */
    inline uint32_t _available() {
      return ((p_read - p_write - 1) % (max_size + 1));
    }

  public:

    CircularList() {

      this->waiting_data = false;

      // Reset list size
```

```
      p_write = 0;
      p_read = 0;

77    // If max_size is not within allowed bounds, the class should not be allowed to
          function at all, as it might behave
      // unpredictably otherwise
      if (max_size <= MAX_ALLOWED_SIZE || max_size < MIN_ALLOWED_SIZE) {
        this->dead = false;
        last_error = OK;

      } else {
        this->dead = true;
        last_error = INVALID_SIZE;
      }
87  }


    ~CircularList() {
      this->empty();
92  }


    /*! @brief Push one element into the list
    *
97  * @param value Value to be stored  in the list
    *
    * @return Result of the operation
    */
    Error push(T value) {
102 #ifdef DEBUG
      auto _full_ = _isFull();
      auto _empty_ = _isEmpty();
      auto _size_ = _size();
      auto _available_ = _available();
107 #endif

      std::unique_lock<std::mutex> lock(this->list_mutex);

      if (this->dead) {
112     last_error = DEAD;

      } else if (_isFull()) {
        last_error = FULL;
```

```
117       } else {

            // Insert the element into the list and increment iterator
            list[(p_write++) % (max_size + 1)] = value;

122         // Notify any thread that might be waiting for an object to be inserted into the
                empty list
            if (this->waiting_data)
              this->list_condition.notify_one();

            last_error = OK;
127       }

          return last_error;
        }


        /*! @brief Push multiple elements into the list
        *
        * @param source Array with the elements to store in the list
        * @param len Length of elements to be pushed into the list, should there be available
            space
137     *
        * @return Result of the operation
        */
        Error pushBulk(T* source, uint32_t len) {
#ifdef DEBUG
142       auto _full_ = _isFull();
          auto _empty_ = _isEmpty();
          auto _size_ = _size();
          auto _available_ = _available();
#endif

          std::unique_lock<std::mutex> lock(this->list_mutex);

          if (this->dead) {
            last_error = DEAD;

          } else if (_isFull()) {
            last_error = FULL;

          } else if (_available() < len) {
```

```
157           last_error = INVALID_SIZE;

        } else {

          uint32_t it = 0;
          while (it < len) {

            // Insert the element into the list and increment iterator
            list[(p_write++) % (max_size + 1)] = source[it++];
          }

          last_error = OK;

          // Notify any thread that might be waiting for an object to be inserted into the
              empty list
          if (this->waiting_data)
172         this->list_condition.notify_one();
        }

        return last_error;
      }


      /*! @brief Pop one element from the list
      *
      * @param destination Pointer to the position in memory whre to store the value
182   * @param blocking Whether the function operating mode is BLOCKING or NON_BLOCKING
      *          - In BLOCKING mode, whenever the list is empty, it waits for there to be
          something in the list and
      *          pops the first element.
      *          - In NON_BLOCKING mode, whenever the list is empty, it returns halfway through
          the operation,
      *          returning Error::EMPTY
187   *
      * @return Result of the operation
      */
      Error pop(T* destination, BlockingMode blocking = BlockingMode::NON_BLOCKING) {
#ifdef DEBUG
192       auto _full_ = _isFull();
          auto _empty_ = _isEmpty();
          auto _size_ = _size();
          auto _available_ = _available();
#endif
```

```
    std::unique_lock<std::mutex> lock(this->list_mutex);

    if (this->dead) {
      last_error = DEAD;
      return DEAD;

    } else if (_isEmpty()){

      // If in BlockingMode::NON_BLOCKING, give up immediately
      if (!blocking) {
        last_error = EMPTY;
        return EMPTY;
      }

      // If in BlockingMode::BLOCKING, wait until something appears in the list
      while (_isEmpty()) {
        this->waiting_data = true;
        this->list_condition.wait(lock, [this]() {return this->dead || !_isEmpty(); });
        this->waiting_data = false;
      }

    }

    // Take element from the list and increment iterator
    *destination = this->list[(p_read++) % (max_size + 1)];

    last_error = OK;

    return last_error;
  }


  /*! @brief Pop multiple elements from the list
  *
  * @param destination Array for storing the elements into
  * @param max Maximum number of elements to be popped from the list, should they be
      present
  * @param blocking Whether the function operating mode is BLOCKING or NON_BLOCKING
  *          - In BLOCKING mode, whenever the list is emptied but the operation is still
      running, it waits for
  *          there to be something in the list and continues popping elements.
  *          - In NON_BLOCKING mode, whenever the list is emptied but the operation is still
```

```
         running, it returns
     *          halfway through the operation
     *
     * @return The number of elements actually popped from the list
     */
242  uint32_t popBulk(T* destination, uint32_t max, BlockingMode blocking = BlockingMode::
         NON_BLOCKING) {
#ifdef DEBUG
       auto _full_ = _isFull();
       auto _empty_ = _isEmpty();
       auto _size_ = _size();
247    auto _available_ = _available();
#endif

       std::unique_lock<std::mutex> lock(this->list_mutex);
       uint32_t popped = 0;  /*! Counter of popped items*/

       if (this->dead) {
         last_error = DEAD;

       } else if (_isEmpty()) {
257      last_error = EMPTY;

       } else {

         while (popped < max) {

           // Take element from the list and increment iterator and counter of popped elements
           destination[popped++] = list[(p_read++) % (max_size + 1)];


267        // If in BlockingMode::BLOCKING, block until something appears in the list
           if (_isEmpty() && popped == max) {

             // If in BlockingMode::NON_BLOCKING, give up immediately
             if (!blocking)
272            break;

             // If in BlockingMode::BLOCKING, wait until something appears in the list
             while (_isEmpty()) {
               this->waiting_data = true;
277            this->list_condition.wait(lock, [this]() {return this->dead || !_isEmpty(); });
               this->waiting_data = false;
```

```
            }

          }
282       }

        last_error = OK;
      }

287   return popped;
    }


    /*! @brief Get the number of elements in the list
292 *
    * @return The number of elements in the list
    */
    uint32_t size() {

297   std::unique_lock<std::mutex> lock(this->list_mutex);

      if (this->dead)
        last_error = DEAD;
      else
302     last_error = OK;

      return _size();
    }


    /*! @brief Break the functioning of the list
    *
    * Once dead, the list may no longer be used. Accessing or operating upon it will yield
        Error::DEAD
    */
312 void kill() {

      std::unique_lock<std::mutex> lock(this->list_mutex);

      this->dead = true;
317   if (this->waiting_data)
        this->list_condition.notify_all();

      last_error = OK;
```

```
    }


    /*! @brief Empty the list
    */
    inline void empty() {
327   std::unique_lock<std::mutex> lock(this->list_mutex);
      this->p_read = this->p_write;
      last_error = OK;
    }


    /*! @brief Get the result from the last function execution
    *
    * @return General result code from the last function execution
    */
337 inline Error getLastError() {
      std::unique_lock<std::mutex> lock(this->list_mutex);
      return last_error;
    }
  };

}

#endif
```

Listing 6.15: MEM_CircularList

6.1.3.4.2. MEM::LinkedList

Implemented with a std::list from the STL of C++ due to the well devised methods made available by this type of container, ensuring a smooth and practical implementation. Also std::list is in and of itself the STL implementation of a linked list, guaranteeing the expected algorithm complexity of a true linked list.

```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H

#include "MEM.hpp"

#ifdef _LINUX_
#include <list>
```

```
#include <mutex>
#include <condition_variable>

#endif

namespace MEM {

  template <typename T>
  class LinkedList {

  private:

#ifdef _LINUX_
    std::mutex list_mutex;          /*! Mutex for use with std::unique_lock in managing
        accesses and operations */
    std::condition_variable list_condition; /*! Condition variable for use in waits (see
        waiting_data: bool) */
#endif

    std::list<T> list;    /* Native list */

    Error last_error; /*! Result from the execution of the last function */
    bool waiting_data;  /*! True when pop() or popBulk() are waiting for an element in
        BlockingMode::BLOCKING */
    bool dead;      /*! Once dead, the list may no longer be used. Accessing or operating
        upon it will yield Error::DEAD */


    /*! @return Whether or not the list is empty
    */
    inline bool _isEmpty() {
      return list.size() == 0;
    }


    /*! @return Whether or not the list is full
    */
    inline bool _isFull() {
      return list.size() == UINT32_MAX;
    }


    /*! @return Number of elements present in the list
```

```
    */
    inline uint32_t _size() {
      return list.size();
    }


    /*! @return Number of spaces left in the list
    */
    inline uint32_t _available() {
      return UINT32_MAX - list.size();
    }

  public:

    LinkedList() {
      this->waiting_data = false;
    }


    ~LinkedList() {
      this->empty();
    }


    /*! @brief Push one element into the list
    *
    * @param value Value to be stored  in the list
    * @param position Where to insert the value
    *          - When position = BACK it will be inserted in the first position of the list
    *          - When position = FRONT it will be inserted after the last position of the list
    *
    * @return Result of the operation
    */
    Error push(T& value, Position position = BACK) {
#ifdef DEBUG
      auto _full_ = _isFull();
      auto _empty_ = _isEmpty();
      auto _size_ = _size();
      auto _available_ = _available();
#endif

      std::unique_lock<std::mutex> lock(this->list_mutex);
```

```
90        if (this->dead) {
            last_error = DEAD;

          } else if (_isFull()) {
            last_error = FULL;

          } else {

            // Insert the element into the list
            auto it = position == BACK ? list.end() : list.begin();
100         list.insert(it, value);

            // Notify any thread that might be waiting for an object to be inserted into the
                empty list
            if (this->waiting_data)
              this->list_condition.notify_one();

            last_error = OK;
          }

          return last_error;
110     }



        /*! @brief Pop one element from the list
115     *
        * @param value Value of the element to be removed from the list
        * @param blocking Whether the function operating mode is BLOCKING or NON_BLOCKING
        *          - In BLOCKING mode, whenever the list is empty, it waits for there to be
            something in the list and
        *          pops the first element.
120     *          - In NON_BLOCKING mode, whenever the list is empty, it returns halfway through
            the operation,
        *          returning Error::EMPTY
        *
        * @return Result of the operation
        */
125     Error pop(T& value, BlockingMode blocking = BlockingMode::NON_BLOCKING) {
    #ifdef DEBUG
          auto _full_ = _isFull();
          auto _empty_ = _isEmpty();
          auto _size_ = _size();
```

```
130       auto _available_ = _available();
#endif

      std::unique_lock<std::mutex> lock(this->list_mutex);

135       if (this->dead) {
        last_error = DEAD;
        return DEAD;

      } else if (_isEmpty()) {

        // If in BlockingMode::NON_BLOCKING, give up immediately
        if (!blocking) {
          last_error = EMPTY;
          return EMPTY;
145       }

        // If in BlockingMode::BLOCKING, wait until something appears in the list
        while (_isEmpty()) {
          this->waiting_data = true;
          this->list_condition.wait(lock, [this]() {return this->dead || !_isEmpty(); });
          this->waiting_data = false;
        }

      }

      // Take element from the list
      list.remove(value);

      last_error = OK;

      return last_error;
    }


165   /*! @brief Get the number of elements in the list
    *
    * @return The number of elements in the list
    */
    uint32_t size() {

      std::unique_lock<std::mutex> lock(this->list_mutex);
```

```
    if (this->dead)
      last_error = DEAD;
    else
      last_error = OK;

    return _size();
  }


  /*! @brief Break the functioning of the list
  *
  * Once dead, the list may no longer be used. Accessing or operating upon it will yield
      Error::DEAD
  */
  void kill() {

    std::unique_lock<std::mutex> lock(this->list_mutex);

    this->dead = true;
    if (this->waiting_data)
      this->list_condition.notify_all();

    last_error = OK;
  }


  /*! @brief Empty the list
  */
  inline void empty() {
    std::unique_lock<std::mutex> lock(this->list_mutex);
    list.clear();
    last_error = OK;
  }


  /*! @brief Get the result from the last function execution
  *
  * @return General result code from the last function execution
  */
  inline Error getLastError() {
    std::unique_lock<std::mutex> lock(this->list_mutex);
    return last_error;
  }
```

```
215   };

}

#endif
```

Listing 6.16: MEM_LinkedList

6.1.3.5. CLK: Timing Package

The **CLK** package is comprised by only the **Timer** module and convenient type definitions. It is very useful for dealing with the periodicity of each thread and provides a means for setting up repeated timed delays and executing a specific routine automatically when each delay ends. It provides an interface for waiting for the end of the timed delay and/or the execution of the specified routine and autonomously notifies all waiting objects of these events. It is mainly used by the **IO::GPIO** class for timing conversions but it can as easily be used by higher-level classes due to its **versatile interface** and **general-purpose** nature.

```
1 #include "CLK_Timer.hpp"

#include <string>
#include <chrono>
#include <iostream>

namespace CLK {

  Timer::Timer() : Timer((CLK::TimeElapsedCallback*)nullptr, nullptr) {

11  }

  Timer::Timer(CLK::TimeElapsedCallback* time_elapsed_callback, void* callback_arg) : id(
      next_id), thread(std::string("Timer " +
    std::to_string(next_id++)).c_str(), this->threadMethod, this, OS::Thread::StackSize::
        DONT_CARE, OS::Thread::Priority::REAL_TIME) {

16    this->isr = time_elapsed_callback;
    this->stop_order = false;
    this->done = true;
    this->isr_arg = callback_arg;
  }
```

```
void Timer::threadMethod(OS::Thread* thread, void* arg) {

  Timer* tmr_ptr = reinterpret_cast<Timer*>(arg);

  std::unique_lock<std::mutex> lock(tmr_ptr->mutex);

  auto start = std::chrono::steady_clock::now();

  while (1) {

    tmr_ptr->condition.wait_until(lock, start + std::chrono::milliseconds(tmr_ptr->counter)
        , [tmr_ptr]() {return tmr_ptr->stop_order == true; });

    // Keep timestamp
    start = std::chrono::steady_clock::now();

    // Notify of the end of the timer countdown
    tmr_ptr->notification_tim_ov.notifyAll();

    // If the wait ended because of a stop order, the program must exit the loop
    if (tmr_ptr->stop_order) {
      tmr_ptr->stop_order = false;
      break;
    }

    // Run interrupt service routine
    (*tmr_ptr->isr)(tmr_ptr->isr_arg);

    // Notify of the end of the ISR
    tmr_ptr->notification_isr_done.notifyAll();

    // Reload counter
    tmr_ptr->counter = tmr_ptr->auto_reload;
  }

  tmr_ptr->done = true;
};

void Timer::start() {

  std::unique_lock<std::mutex> lock(this->mutex);
  last_error = OK;
  done = false;
```

```
    thread.run();
  }

  void Timer::stop() {

    std::unique_lock<std::mutex> lock(this->mutex);
    stop_order = true;
    last_error = OK;
  }

  void Timer::waitNotification(bool isr_done, bool tim_ov) {

    if (tim_ov)
      notification_tim_ov.wait();

    if (isr_done)
      notification_isr_done.wait();

    last_error = OK;
  }

}
```

Listing 6.17: CLK::Timer

6.1.3.6. APP: Main Application Package

The main application uses the threads implemented in the section 6.1.2. These threads are configured and set to run in the **main** method.

```
int main(int argc, char* argv[]) {

  using namespace DEBUG;

  OS::Thread controlThread("Control Thread",
    controlThread, nullptr, OS::Thread::DONT_CARE, OS::Thread::HIGH);
  OS::Thread simulationThread("Simulation Thread",
    simulationThread, nullptr, OS::Thread::DONT_CARE, OS::Thread::HIGH);

  simulationThread.run();
```

```
    controlThread.run();

    simulationThread.join();
    controlThread.join();

    std::cout << "\nDone. Press ENTER to exit.\n";
    std::cin.get();
    return 0;
}
```

Listing 6.18: Main Application

6.2. Remote Vision Virtual Subsystem

The design specification devised in Section 6.2 can now be implemented to the target platform to fulfill the remote vision and telemetry functionalities required. In this section the relevant implementation details are addressed on the hardware and software domains.

6.2.1. Hardware

In the design stage, and in the first iterations of the implementation, it is perfectly reasonable to adopt general domain environments, such as virtual machines. Nonetheless, one must keep in mind that, ultimately, the devised software will be running on hardware nodes. Thus, an important implementation step is the selection of the target platform, taking into considerion several design criteria, such as, throughput, memory footprint, storage, response time, connectivity, etc.

An example of a viable platform is the Raspberry Pi Zero W (Fig. 6.3), which is a low cost board design (it retails for about 5 EUR) around the Broadcom system on-chip (SoC) BCM2835, which includes a 1 GHz single-core ARMv6 CPU, with a 64-bit architecture, allowing it to run the full range of GNU/Linux distribution. Among other resources, it includes 512 MB RAM, VideoCore IV GPU, 40 GPIO pins, mini HDMI and micro-USB ports, Camera Serial Interface (CSI) connector, Micro SD Card Slot and on-board Bluetooth Low Energy (BLE) 4.1 and Wi-Fi based.

The Raspberry Pi Zero W provides the required communications (Bluetooth and Wi-Fi) and camera interfaces, on a fully-fledged environment capable of running 64-bit OS and at low cost, thus, making it suitable for the implementation. Additionally, it packs in a small form-factor, even with the inclusion of the camera,

as illustrated in Fig. 6.4. The camera module has several models ranging from 5 to 12 megapixel resolution with high quality video capture (up to 1080p30) in the 20–35 EUR pricepoint.

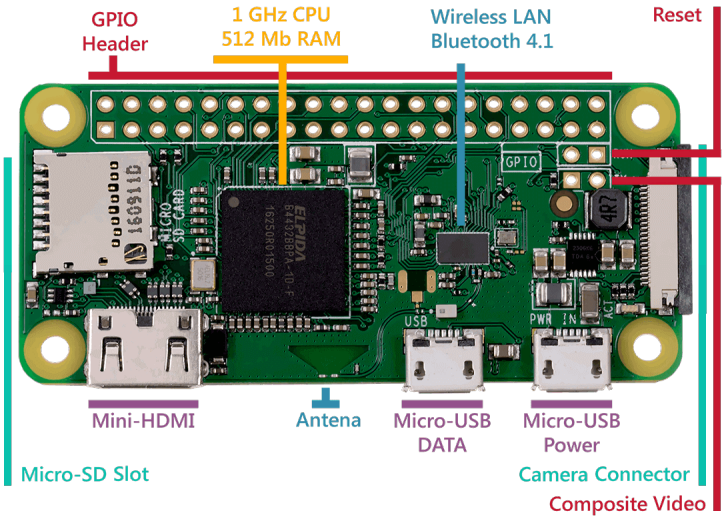

Figure 6.3.: Raspberry Pi Zero Wireless overview

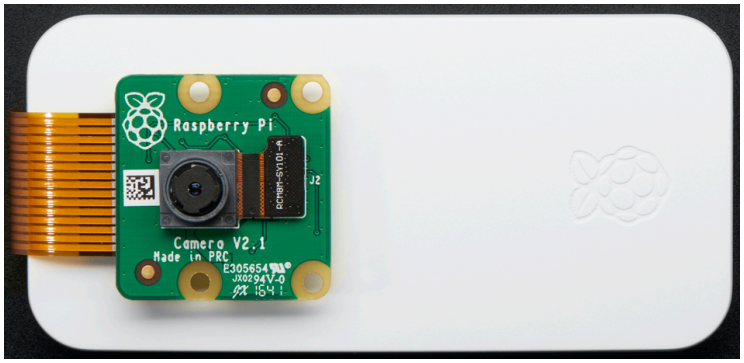

Figure 6.4.: Raspberry Pi Zero Wireless overview

6.2.2. Software

The software part is mainly comprised of two main components: the software environment support and the actual software running on top of that. The software environment provides the low-level layer(s) that interfaces the hardware, typically under some form of an OS and the associated development toolchain required to target the platform.

In the present case, the selected platform — Raspberry Pi — supports 64-bit GNU/Linux based operating systems, which can be used to bootstrap the implementation. Nonetheless, it is desirable to maintain the

code footprint and dependencies as low as possible, as required in the majority of the embedded systems, thus, a custom tailored Linux OS can be used. Contemplating the build tools required for the deployment of a tailored Linux OS, there are several solutions, such as, Buildroot, Yocto and OpenWRT. The selected build tool was Yocto, an open-source project project that provides templates, tools, and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture [12]. Additionally, it is required the cross-compilation toolchain that enables deployment to target from host. Yet, in this initial implementation phase, the RVVS susbsystem is virtualized in a Linux Virtual Machine (VM), thus, sparing this step for now. However, after implementing and testing the application in the Linux VM, this approach should be relatively straightforward, enabling fast deployment to the real hardware.

Hence, in the following sections, it is discussed the implementation of the RVVS stack designed in Section 5.3.3.

6.2.2.1. Image Acquisition

The usage of the Linux VM has another inherent advantage, associated to the UNIX philosophy in which it is based, that everything is a file (except for the network part). As a matter of fact, every device attached to the Linux OS can be easily and transparently accessed, e.g., via the /**dev** directory. Additionally, as every device is a file, the file API can be used to manage the device, using the typical system calls **open**/**close** and **read**/**write**.

The typical call stack for image acquisition and streaming is comprised of: low-level drivers that interface the hardware via the kernel (dependent on the type of interface), dealing with frame capture (e.g., Video4Linux (V4L)); and the commonly named frame grabber, which takes the raw frames and encodes it into streams, for posterior multiplexing into their final container (e.g., **ffmpeg**).

The Video4Linux2 (V4L2) is the second version of the API and framework, which is an integral part of the Linux kernel code, as opposed to many driver implementations [13]. The V4L2 API is mostly implemented as set of **ioctl** (control device) system calls [8] that enable easy manipulation of video devices. The workflow for a typical V4L2 application is as follows:

1. Open a descriptor to the device;
2. Retrieve and analyse the device's capabilities. V4L2 allows you to query a device for its capabilities, that is, the set of operations (roughly, IOCTL calls) it supports;
3. Set the capture format: frame size, format (MJPEG, RGB, YUV, ...), etc.; check the device handles the format.

4. Prepare the device for buffer handling. When capturing a frame, it has to submitted a buffer to the device (**queue**), and retrieved it once it's been illed with data (**dequeue**). However, before this, the device must be informed about the buffers (buffer request).

5. For each buffer, certain aspects must be negotiated with the device (buffer size, frame start offset in memory), and then created a new memory mapping for it.

6. Put the device into streaming mode.

7. Once the buffers are ready, it is only required the queueing/dequeuing of the buffers repeatedly, and every call will grab a new frame. The delay defined between each frames by putting the program to sleep is what determines the framerate.

8. Turn off streaming mode.

9. Close the descriptor to the device.

The V4L2 API is implemented in the **C** programming language. Thus, a basic wrapper class was implemented in **C++** to abstract from the low-level details and increase modularity, following the aforementioned workflow.

Listing 6.19 provides the webcam interface (public and private) using the V4L2 API. The construtor takes the type of capture to perform, e.g., video capture, and the associated memory type. The open/close member functions enable the opening and closing of the file descriptior associated with the device. **setFormat** sets the image format and dimensions, **startStream** starts the stream and writes the output to a file, and **setRequestBuffer** defines the number of buffers to use. In the private interface lies the **allocateBuffer** and **open** member functions which allocates a buffer for the device on behalf of the **setRequestBuffer**. As private variables there are the file descriptor associated with the device **m_FileDescriptor**, the pointer to a buffer frame **m_bufferPtr**, and the V4L2 variables representing the buffer's type, memory type and capabilities, respectively. Additionally, a mutually exclusive object variable **m_mutex** is used to manage concurrent access to the device via file descriptor.

```
class Webcam_V4L2
{
public:
/// @brief Constructor
/// @param type: type of capture to perform
/// @param memory: type of buffer memory used
  Webcam_V4L2(v4l2_buf_type type, v4l2_memory memory);
```

```
/// @brief Opens the web cam using a named device node
/// @param device: e.g, /dev/video0
  void open(const std::string& device);

/// @brief Close the webcam
  void close() noexcept;

/// @brief Queries if the webcam has been opened
/// @return True if opened, false otherwise
  bool isOpen() const noexcept;

/// @brief Sets the image format and dimensions
/// @param width        The width of the image in pixels
/// @param height       The height of the image in pixels
/// @param pixelformat  The encoding of the image (e.g., JPEG)
/// @param field        Unknown
  void setFormat(uint32_t width, uint32_t height, uint32_t pixelformat,
                 v4l2_field field = V4L2_FIELD_NONE);

/// @brief Sets the nr of buffers to use
/// @param numBuffers: nr of buffers to use
  void setRequestBuffer(uint32_t numBuffers);

/// @brief Starts the stream and writes the output to a file
/// @param file The filename to write to
  void startStream(const std::string& file);

private:

/// @brief allocates buffer on @setRequestBuffer behalf
  void allocateBuffer();

/// @brief Maintains integrity of the file descriptor (FID) and returns it
/// @return FID if valid, -1 otherwise
  int fileDescriptor() const noexcept;

/// @brief Low level call to open device
/// @param device: device to be opened
  void open(const char* device) noexcept;

    /* V4L2 variables */
  const v4l2_buf_type m_bufferType; /**< Buffer type */
  const v4l2_memory m_memory; /**< Buffer memory */
```

```
  v4l2_capability m_capability; /**< Buffer capabilites */

  int m_fileDescriptor; /**< file descriptior associated with the device */
  char* m_bufferPtr; /**< Pointer to a buffer */

  mutable std::mutex m_mutex; /**< mutex variable to prevent unattended access*/
};

#endif /* WEBCAM_V4L2_H */
```

Listing 6.19: Webcam wrapper interface using the V4L2 API

Next, a small driver program (Listing 6.20) was devised to test the webcam interface. The device location is set to /dev/video0 where the attached camera is placed under Linux filesystem. A webcam object is created for video capture and then is tried out the aforementioned workflow by opening the device, setting the dimensions to 320x240 pixels and the format to Motion JPEG (MJPEG), requesting one buffer, starting the stream to an output file and finally closing the device.

```
static const string DEVICE = "/dev/video0"; /**< device location */

int main(int argc, char* argv[])
{ /* Create webcam for video capture */
  Webcam_V4L2 wbc(V4L2_BUF_TYPE_VIDEO_CAPTURE, V4L2_MEMORY_MMAP);

  try{
/* Open the device */
    wbc.open(DEVICE);
/* Set format to 320x240 and Motion JPEG */
    wbc.setFormat(320, 240, V4L2_PIX_FMT_MJPEG);
/* Request 1 buffer */
    wbc.setRequestBuffer(1U);
/* Start stream and write to file */
    wbc.startStream("output.jpg");
/* Close device */
    wbc.close();
  }catch(const Webcam_V4L2Exception& e){
    std::cerr << e.what() << std::endl;
    return EXIT_FAILURE;
  }
  return EXIT_SUCCESS;
```

```
}
```

Listing 6.20: Webcam driver program

Later on, the client code that uses the Webcam interface was added to respective thread, responsible for starting a webcam stream on user-demand.

6.2.2.2. Wi-Fi

For the implementation of the Wi-Fi communication, TCP/IP sockets were used in conjunction with the client/server architecture. Although only the server is running on the RVVS subsystem, the instantiation of both client and server enables the testing of the complete communications workflow, without the need to add the uncertainty associated with the communication medium.

As aforementioned the RVVS stack shares common elements with the NVS due to the shared interfaces and common philosophy, enabling code reuse. In that sense, the **OS**, **COM Transport**, and **COM Data** were fully reused.

Additionally, the **WCOM** packages, corresponding to the Wi-Fi communication, were adapted from the **COM** packages, as both rely on sockets and on the client/server architecture with concurrent execution. The difference relies on the type of protocol used (TCP instead of RS232) and some idiosyncrasies of the Internet programming, namely:

- Socket Family The **TCP_SOCKET_FAMILY** is **AF_INET**, with the **SOCKET_TYPE** remaining the same (**SOCK_STREAM**).
- Port range:The port range available is from 1024–65535, as the ports between 0–1023 are well-known/reserved ports (system ports).
- Byte ordering(endianess): the network byte ordering is always big endian. To avaid translation mistakes, before sending data packets to the network, they must always be converted to the network byte ordering (big endian). For this purpose are available some utility functions for the conversion:
  - Host Byte Order to Network Byte Order: **htons**, **htonl**
  - Network Byte Order to Host Byte Order: **htons**, **htonl**
- **Padding**: the TCP header padding is used to ensure that header and data are aligned with multiples of 32-bits (4 bytes). For this purposes, the **bzero** function can be used, or, for above **C++11**, the library function **std::fill_n**.

- **IP Address format**: IP addresses for devices are encoded in binary, thus, to obtain a human readable address, the ASCII dotted to Binary (**inet_aton**) and Binary to ASCII dotted (**inet_ntoa**) functions can be used.

- **hostname**: in same cases, like connecting to a server, is required to obtain additional information from its hostname. For this purpose the function **gethostbyname** function can be used.

The remaining part of the implementation is identical to the **COM** sockets, thus, exempting further explanations.

6.2.2.3. Telemetry

The telemetry class provides basic functionalities for obtaining simple, yet relevant statistics about vehicle's operation. The implementation is trivial, following the devised in the class diagram, thus, exempting also further explanations.

6.3. Smartphone

The mobile OS chosen for this project was **Android**. Usually, android apps can be implemented using Kotlin, Java and C++. For the RFCAR project, the language chosen was **Java** due to the knowledge gained in prior course years where the need to implement android-targeted applications rose. Additionally, the code was conceived using the **Android Studio** Integrated Development Environment (IDE). One must notice that despite the user-friendliness of the development environment through context-sensitive guidelines and code suggestions, this language and IDE are not the best in terms of full system control due to the multiple abstraction layers preemptively defined. Therefore, one might wonder why the C++ route with a cross-platform framework like QT wasn't selected. Multiple options were taken into account at this stage and its a fact that the one that ended up being selected might not deliver as much implementation freedom as the second option forementioned. However, it allowed deadline fulfilment and code reuse notwithstanding the additional effort put into delving deeper within some contexts.

6.3.1. Sensor Interaction

The smartphone has a built-in MEMS accelerometer, this means that at micro-level it can measure acceleration values through capacitance changes in multiple capacitors as a result of its internal assembly

(calibration mass and spring contacts) displacement. With at least a MEMS system in each plane (x,y,z), one can measure the acceleration per axis.

6.3.1.1. Sensor Data Retrieval

The code in listing 6.21 (based on [14]) represents the retrieval of the linear acceleration for each plane. In the first place, the definition of the sensor type is crucial to access its values (line 9). **SensorManager** grants access to the sensors of the android device (line 7). Following, one must create a listener that checks the sensors values at a determined sampling frequency using the SensorManager's method **registerListener** (line 10). Upon doing so, one overwrites the **onSensorChanged** method (line 15) so the pretended variables that hold the values of the sensor can only be updated on smartphone movement. An acceleration sensor measures the acceleration applied to the device but regarding the force of gravity. For this reason, the values retrieved do not represent the linear accelerations for each plane. To resolve this problem, a low pass filter can be used to isolate the force of gravity in each axis (line 28) and then remove its contibution from the acceleration values (line 33).

```
TextView x_val, y_val, z_val;
private float sensorX, sensorY, sensorZ;

...

Log.d(TAG, "onCreate: Initializing Sensor Services");
sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
sensorManager.registerListener(MainActivity.this, accelerometer, SensorManager.
    SENSOR_DELAY_NORMAL);
Log.d(TAG, "onCreate: Registered accelerometer listener");

...

@Override
    public void onSensorChanged(SensorEvent event) {
        Sensor mysensor = event.sensor;
        if(mysensor.getType() == Sensor.TYPE_ACCELEROMETER) {

            final float alpha = (float) 0.8;
            float gravityX = 0;
            float gravityY = 0;
```

```
            float gravityZ = 0;
            float linear_accelerationX = 0;
            float linear_accelerationY = 0;
            float linear_accelerationZ = 0;

            // Isolate the force of gravity with the low-pass filter.
            gravityX = alpha * gravityX + (1 - alpha) * event.values[0];
            gravityY = alpha * gravityY + (1 - alpha) * event.values[1];
            gravityZ = alpha * gravityZ + (1 - alpha) * event.values[2];

            // Remove the gravity contribution with the high-pass filter.
            linear_accelerationX = event.values[0] - gravityX;
            linear_accelerationY = event.values[1] - gravityY;
            linear_accelerationZ = event.values[2] - gravityZ;

            Log.d(TAG, "onSensorChanged: X: " + linear_accelerationX + " Y: " +
                linear_accelerationY + " Z: " + linear_accelerationZ);

            x_val.setText(String.format("X: %s", linear_accelerationX));
            y_val.setText(String.format("Y: %s", linear_accelerationY));
            z_val.setText(String.format("Z: %s", linear_accelerationZ));

            long currTime = System.currentTimeMillis();

            if((currTime - lastSensorUpdateTime) > 50){
                lastSensorUpdateTime = currTime;

                sensorX = linear_accelerationX;
                sensorY = linear_accelerationY;
                sensorZ = linear_accelerationZ;
            }
        }
    }
```

Listing 6.21: Accelerometer data retrieval code

As the control module uses both wheels tilt angle and tension applied to the motor as input, the interface with the smartphone's accelerometer wasn't enough. To generate the angles necessary and vary the tension accordingly one needs to access the phone's rotation sensors. This code is implemented in listing 6.22. As the interface with the accelerometer, this latter interface needs a **SensorManager** and the creation of a listener with the **registerListener** method. Some calibrations were crucial to ensure the calculated values of the angles matched the initial smartphone position chosen. Note that percentages were used as a way

to implement a control independent of the motor and wheels used. This means that using a percentage of the voltage applied and wheel tilt, one could simply use a 12V or 6V motor, for example. Therefore, the control values would be immutable. This will be tested in section 7.1.3.1.

```
public class MainActivity extends AppCompatActivity implements SensorEventListener {

    private static final String TAG = "MainActivity";

    private SensorManager sensorManager;
    Sensor rot_sensor;

    TextView x_val, y_val, z_val, inclination, rotation, pitch, roll, azimuth, voltage,
        wheel_t;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        pitch = (TextView) findViewById(R.id.pitch);
        roll = (TextView) findViewById(R.id.roll);
        azimuth = (TextView) findViewById(R.id.azimuth);
        voltage = (TextView) findViewById(R.id.voltage_layout);
    wheel_t = (TextView) findViewById(R.id.wheel_tilt_layout);

        Log.d(TAG, "onCreate: Initializing Sensor Services");
        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
        rot_sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ROTATION_VECTOR);
        sensorManager.registerListener(MainActivity.this, rot_sensor, 10000);
    }

        @Override
        public void onAccuracyChanged (Sensor sensor,int i){}

        @Override
        public void onSensorChanged (SensorEvent event){
            Sensor mysensor = event.sensor;

            if (mysensor.getType() == Sensor.TYPE_ROTATION_VECTOR) {
                float[] g = event.values.clone();

                //Normalise
```

```
                double norm = Math.sqrt(g[0] * g[0] + g[1] * g[1] + g[2] * g[2] + g[3] * g
                    [3]);
                g[0] /= norm;
                g[1] /= norm;
                g[2] /= norm;
                g[3] /= norm;

                //Set values to commonly known quaternion letter representatives
                double x = g[0];
                double y = g[1];
                double z = g[2];
                double w = g[3];

                //Calculate Pitch in degrees
                double sinP = 2.0 * (w * x + y * z);
                double cosP = 1.0 - 2.0 * (x * x + y * y);
                int pitch_ = (int) Math.round(Math.atan2(sinP, cosP) * (180 / Math.PI)) + 30;

                int roll_ = 0;
                //Calculate Tilt in degrees
                double sinT = 2.0 * (w * y - z * x);
                if (Math.abs(sinT) >= 1)
                    roll_ = (int) (Math.copySign(Math.PI / 2, sinT) * (180 / Math.PI)) + 90;
                else
                    roll_ = (int) (Math.asin(sinT) * (180 / Math.PI)) + 90;

                //Calculate Azimuth in degrees (0 to 360; 0 = North, 90 = East, 180 = South,
                    270 = West)
                double sinA = 2.0 * (w * z + x * y);
                double cosA = 1.0 - 2.0 * (y * y + z * z);
                int azimuth_ = (int) (Math.atan2(sinA, cosA) * (180 / Math.PI));

                int rollThreshold = 60;
                int pitchThreshold = 40;
                int rollInitialVal = 5;
                int pitchInitialVal = 0;
                int max_tension = 6;
                float voltage_ = 0;
        float wheel_tilt = 0;
                int max_wheel_tilt = 45;
                int right_w = 64;
                int left_w = 56;
                int init_tilt = 94;
```

```
                int temp_diff = Math.abs(rot - init_tilt);

81              if(roll_ > 0)
                    voltage_ = (float) (Math.abs(roll_) * max_tension) / rollThreshold;
                if(voltage_ > max_tension)
                    voltage_ = max_tension;
                else if(Math.abs(pitch_) > 70)
86                  voltage_ = 0;

                if(rot > init_tilt) {
                    wheel_tilt = (float) (temp_diff * 100) / left_w;
                if(wheel_tilt > 100)
91                  wheel_tilt = 100;
                }
                else if(rot == init_tilt)
                    wheel_tilt = 0;
                else {
96                  wheel_tilt = (float) (temp_diff * 100) / right_w;
                    if(wheel_tilt > 100)
                        wheel_tilt = -100;
                    else
                        wheel_tilt = -wheel_tilt;

                }

                Log.d(TAG, "onSensorChanged: Voltage_: " + String.format("%s", voltage_));

106                 pitch.setText(String.format("Pitch: %s", pitch_));
                    roll.setText(String.format("Roll: %s", roll_));
                    azimuth.setText(String.format("Azimuth: %s", azimuth_));
                    voltage.setText(String.format("Speed(%%Max): %s", voltage_));
                    wheel_t.setText(String.format("Wheel tilt(%%Max): %s", wheel_tilt));
111             }
        }
```

Listing 6.22: Rotation sensor data retrieval code

6.3.1.2. Applying Sensor Data

The project requires that the acceleration values obtained from the sensors are applied to make the vehicle move in the intended direction with the expected speed. To implement the code referent to this

sub-subsection, one must pay close attention to the axis orientation in a common device, represented in figure 6.5. In order to test this concept, the code presented in 6.3.1.1 that refers to the accelerometer was

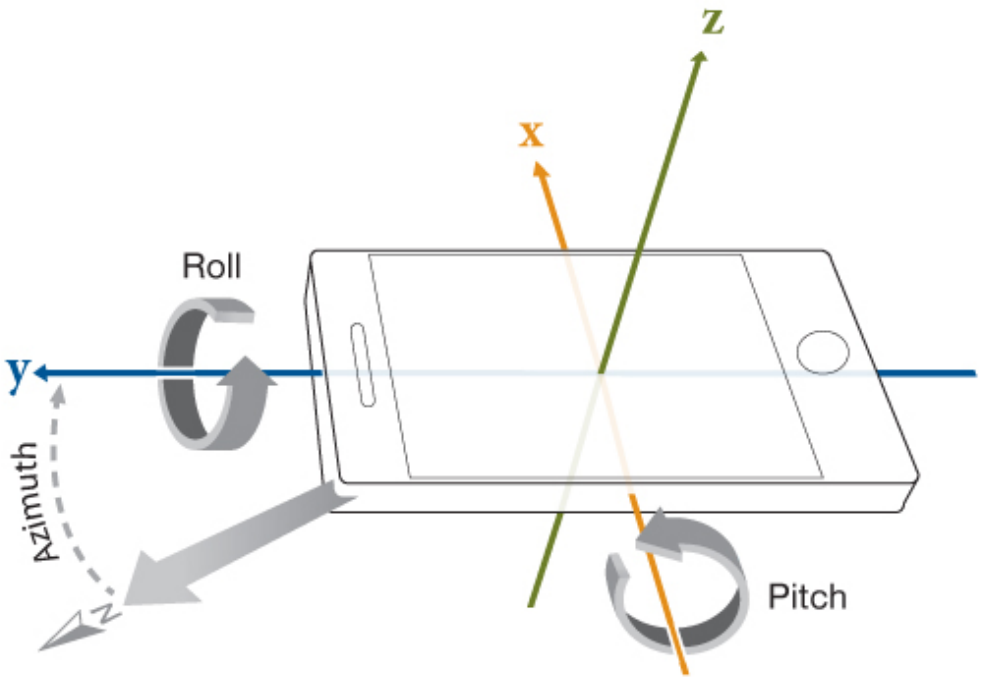

Figure 6.5.: Axis orientation in a smartphone

added to another project that allowed ball movement based on the accelerometer values. It must be noticed that the z acceleration value it's not relevant to the ball movement (from line 59 to 67) since only **roll** and **pitch** affect a 2D (bidimensional) object and the smartphone height relative to the ground doesn't, as one should expect by analysing figure 6.5. This code in listing 6.23 representing subsection 6.3.1 will be tested in 7.1.3.1.

```
public class MainActivity extends AppCompatActivity implements SensorEventListener {

    private static final String TAG = "MainActivity";

    private SensorManager sensorManager;
    Sensor accelerometer;

    TextView x_val, y_val, z_val;
    private float sensorX, sensorY, sensorZ;

    private CanvasView canvas;
```

```
    private int circleRadius = 30;
    private float circleX, circleY;

    private Timer timer;
    private Handler handler;
    private long lastSensorUpdateTime = 0;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        x_val = (TextView) findViewById(R.id.xValue);
        y_val = (TextView) findViewById(R.id.yValue);
        z_val = (TextView) findViewById(R.id.zValue);

        Log.d(TAG, "onCreate: Initializing Sensor Services");
        sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

        accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        sensorManager.registerListener(MainActivity.this, accelerometer, SensorManager.
            SENSOR_DELAY_NORMAL);
        Log.d(TAG, "onCreate: Registered accelerometer listener");

        Display display = getWindowManager().getDefaultDisplay();
        Point size = new Point();
        display.getSize(size);

        int screenWidth = size.x;
        int screenHeight = size.y;

        circleX = (screenWidth >> 1) - circleRadius;
        circleY = (screenHeight >> 1) - circleRadius;

        canvas = new CanvasView(MainActivity.this);
        setContentView(canvas);

        handler = new Handler(){
            public void handleMessage(Message message){
                canvas.invalidate();
            }
        };
```

```
        timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {

                if(sensorX > 0)
                    circleX -= 10;
                else
                    circleX += 10;

                if(sensorY > 0)
                    circleY += 10;
                else
                    circleY -= 10;

                handler.sendEmptyMessage(0);
            }
        }, 0, 50);
    }

    @Override
    public void onAccuracyChanged(Sensor sensor, int i){}

    @Override
    public void onSensorChanged(SensorEvent event) {
        Sensor mysensor = event.sensor;
        if(mysensor.getType() == Sensor.TYPE_ACCELEROMETER) {

            final float alpha = (float) 0.8;
            float gravityX = 0;
            float gravityY = 0;
            float gravityZ = 0;
            float linear_accelerationX = 0;
            float linear_accelerationY = 0;
            float linear_accelerationZ = 0;

            // Isolate the force of gravity with the low-pass filter.
            gravityX = alpha * gravityX + (1 - alpha) * event.values[0];
            gravityY = alpha * gravityY + (1 - alpha) * event.values[1];
            gravityZ = alpha * gravityZ + (1 - alpha) * event.values[2];

            // Remove the gravity contribution with the high-pass filter.
            linear_accelerationX = event.values[0] - gravityX;
```

```
            linear_accelerationY = event.values[1] - gravityY;
            linear_accelerationZ = event.values[2] - gravityZ;

            Log.d(TAG, "onSensorChanged: X: " + linear_accelerationX + " Y: " +
                linear_accelerationY + " Z: " + linear_accelerationZ);

            x_val.setText(String.format("X: %s", linear_accelerationX));
            y_val.setText(String.format("Y: %s", linear_accelerationY));
            z_val.setText(String.format("Z: %s", linear_accelerationZ));

            long currTime = System.currentTimeMillis();

            if((currTime - lastSensorUpdateTime) > 50){
                lastSensorUpdateTime = currTime;

                sensorX = linear_accelerationX;
                sensorY = linear_accelerationY;
                sensorZ = linear_accelerationZ;
            }
        }
    }

    //Canvas
    private class CanvasView extends View {
        private Paint pen;
        public CanvasView(Context context){
            super(context);
            setFocusable(true);

            pen = new Paint();
        }

        public void onDraw(Canvas screen){
            pen.setStyle(Paint.Style.FILL_AND_STROKE);
            pen.setAntiAlias(true);
            pen.setTextSize(30f);

            int color = ContextCompat.getColor(MainActivity.this, R.color.Orange);
            pen.setColor(color);

            screen.drawCircle(circleX, circleY, circleRadius, pen);
        }
```

```
    }
}
```

Listing 6.23: Code for ball movement based on accelerometer data

6.3.2. Bluetooth

Bluetooth is one of the communication technologies specified in the project analysis (section 4), hence it is important to assure the data flow between the smartphone and the NVS on another redundancy communication failure like Wi-Fi.

6.3.2.1. Bluetooth Connection Setup

Starting from the design and reusing previous Java code made on other course units (based on [15]), the Bluetooth setup was implemented on Android Studio and then deployed in a smartphone, the code is depicted in listing 6.24. Note that the forementioned code has **threads** to simulate parallel computing since the app should be able to send data to the paired device and receive data from it. In lines 10, 11 and 12, are represented the three thread classes used for the Bluetooth connection setup. These classes inherit from the **Thread class** (extends Thread - line 114), considering it must exist **concurrency** and **resource sharing** for the application to send and receive data simultaneously. The **AcceptThread** (line 10) it's related to the discovery of new connections since it plays an important role in the creation of a **BluetoothServerSocket** that allows two intended devices to find and accept each other as a part of the initial device inquiry prior to the connection stage. Line 11 represents the **ConnectThread** responsible for managing the connection between two devices. This thread starts as soon as the two devices accept each other and then proceeds to create a **RfcommSocket** and connect the devices (pair) when the user clicks on an unpaired device from the list presented. Finally, the **IOThread** assures the message exchange between devices by accessing the input and output streams of the **Bluetooth Socket**. As stated earlier, it was necessary to make some includes in the Bluetooth app as well as virtualize some ports in order to interface the Personal Computer (PC) virtual machine with the smartphone, this will be more thoroughly explained in the testing section (**??**). Finally, from the smartphone point of view, the protocol purpose was to transmit commands to control the vehicle by sending the phone accelerometers' values and receive its message warnings on screen.

```
public class ChatHandler {
    private static final String TAG ="log";
```

```
    private static final String APP_NAME = "RFCAR App";

    // Unique UUID for this application
    private static final UUID MY_UUID = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB
        ");

    private final BluetoothAdapter bluetoothAdapter;
    private final Handler handler;
    private AcceptThread  acceptThread;
    private ConnectThread  connectThread;
    private IOThread ioThread;
    private int state;
    private Context mContext;
    private UUID deviceUUID;
    private BluetoothDevice bluetoothDevice;
    byte[] buffer;

    private static final int LISTENING_STATE = 1;
    private static final int CONNECTING_STATE = 2;
    private static final int NO_STATE = 4;
    static final int CONNECTED_STATE = 3;

    //Constructor
    ChatHandler(Context context, Handler handler) {
        mContext = context;
        bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
        state = NO_STATE;
        this.handler = handler;
        start();
    }

    // initiate connection to remote device
    synchronized void initializeClient(BluetoothDevice device, UUID uuid) {
        // Cancel any thread
        if (state == CONNECTING_STATE) {
            if (connectThread != null) {
                connectThread.cancel();
                connectThread = null;
            }
        }

        // Cancel running thread
        if (ioThread != null) {
```

```
            ioThread.cancel();
            ioThread = null;
        }

        // Start the thread to connect with the given device
        connectThread = new ConnectThread(device, uuid);
        connectThread.start();
        setState(CONNECTING_STATE);
    }

    synchronized int getState() {
        return state;
    }

    private synchronized void connected(BluetoothSocket socket, BluetoothDevice device) {

        Log.d(TAG, "connected: Starting connected thread");

        //Cancelling all kinds of threads that may be running to start a new one

        // Stopping the thread if it is in connecting state
        if (connectThread != null) {
            connectThread.cancel();
            Log.d(TAG, "connected: Cancelled connectThread");
            connectThread = null;
        }

        // Cancel running thread that refers to the connected state
        if (ioThread != null) {
            ioThread.cancel();
            Log.d(TAG, "connected: Cancelled ioThread");
            ioThread = null;
        }

        //Stopping the scanning thread
        if (acceptThread != null) {
            acceptThread.cancel();
            Log.d(TAG, "connected: Cancelled acceptThread");
            acceptThread = null;
        }

        // Start the thread to manage the connection and perform transmissions
        ioThread = new IOThread(socket);
```

```
        ioThread.start();

        Log.d(TAG,"IOthread: ioThread.start() called");

        Message msg = handler.obtainMessage(Drawer_Activity.MESSAGE_DEVICE_OBJECT);
        Bundle bundle = new Bundle();
        bundle.putParcelable(Drawer_Activity.DEVICE_OBJECT, device);
        msg.setData(bundle);
        handler.sendMessage(msg);

        setState(CONNECTED_STATE);
    }

    void write(byte[] out) {
        IOThread r;
        synchronized (this) {
            if (state != CONNECTED_STATE) {
                Log.d(TAG, "IOthread: Write: state != CONNECTED_STATE");
                return;
            }
            r = ioThread;
        }
        r.write(out);
    }

    // runs while scanning for connections
    private class AcceptThread  extends Thread {
        private final BluetoothServerSocket serverSocket;

        //Constructor
        AcceptThread() {
            BluetoothServerSocket bluetoothServerSocket = null;
            try {
                bluetoothServerSocket = bluetoothAdapter.
                    listenUsingInsecureRfcommWithServiceRecord(APP_NAME, MY_UUID);
                Log.d(TAG, "AcceptThread: Setting up the Server using: " + MY_UUID);
            } catch (IOException ex) {
                ex.printStackTrace();
                Log.e(TAG, "AcceptThread: IOException " + ex.getMessage());
            }
            serverSocket = bluetoothServerSocket;
        }
```

```
130         public void run() {


            Log.d(TAG, "run: AcceptThread Running.");
            BluetoothSocket bluetoothSocket = null;
135         while (state != CONNECTED_STATE) {
                try {
                    Log.d(TAG, "Running: RFCOM server socket started...");
                    bluetoothSocket = serverSocket.accept();
                    int d = Log.d(TAG, "AcceptThread: accepted the request");
140             } catch (IOException e) {
                    Log.e(TAG, "AcceptThread: IOException" + e.getMessage() );
                    break;
                }

145             // If a connection was accepted
                if (bluetoothSocket != null) {
                    synchronized (ChatHandler.this) {
                        switch (state) {
                            case LISTENING_STATE:
150                         case CONNECTING_STATE:
                                // start the connected thread.
                                Log.i(TAG, "Call to connected method");
                                connected(bluetoothSocket, bluetoothSocket.getRemoteDevice())
                                    ;
                                Log.i(TAG, "END initializeClientAcceptThread ");
155                             break;
                            case NO_STATE:
                            case CONNECTED_STATE:
                                // Either not ready or already connected. Terminate
                                // new socket.
160                             try {
                                    bluetoothSocket.close();
                                } catch (IOException e) {}
                                break;
                        }
165                 }
                }
            }
        }

170     void cancel() {
            try {
```

```
                serverSocket.close();
                Log.i(TAG, "END mAcceptThread ");
            } catch (IOException e) {
                Log.e(TAG, "cancel: Close of AcceptThread ServerSocket failed. " + e.
                    getMessage() );
            }
        }
    }

    // runs while attempting to pair with a device
    private class ConnectThread  extends Thread {
        private BluetoothSocket bluetoothSocket;

        //Constructor
        ConnectThread(BluetoothDevice device, UUID uuid) {
            Log.d(TAG, "ConnectThread: started.");
            bluetoothDevice = device;
            deviceUUID = uuid;
        }

        public void run() {
            setName("PairingThread");
            Log.i(TAG, "Running ConnectThread ");

            BluetoothSocket temp = null;
            try {
                Log.d(TAG, "ConnectThread: Trying to create InsecureRfcommSocket using UUID:
                    "+ MY_UUID );
                temp = bluetoothDevice.createInsecureRfcommSocketToServiceRecord(MY_UUID);

            } catch (IOException e) {
                e.printStackTrace();
                Log.e(TAG, "ConnectThread: Could not create InsecureRfcommSocket " + e.
                    getMessage());
            }

            bluetoothSocket = temp;

            //Cancel discovery
            bluetoothAdapter.cancelDiscovery();

            // Make a connection to the BluetoothSocket
            try {
```

```
                bluetoothSocket.connect();
            } catch (IOException e) {
                try {
                    bluetoothSocket.close();
                    Log.d(TAG, "run: Closed Socket");
                } catch (IOException exception) {
                    Log.e(TAG, "ConnectThread: run: Unable to close connection in socket " +
                        exception.getMessage());
                }

                //connectionFailed
                Log.d(TAG, "run: ConnectThread: Could not connect to UUID: " + MY_UUID );
                Message msg = handler.obtainMessage(Drawer_Activity.MESSAGE_TOAST);
                Bundle bundle = new Bundle();
                bundle.putString("Toast", "Unable to connect to the device");
                msg.setData(bundle);
                handler.sendMessage(msg);
            }

            // Reset the ConnectThread
            synchronized (ChatHandler.this) {
                connectThread = null;
            }

            // Start the connected thread
            connected(bluetoothSocket, bluetoothDevice);
        }

        void cancel() {
            try {
                Log.d(TAG, "cancel: Closing Client Socket");
                bluetoothSocket.close();
            } catch (IOException e) {
                Log.e(TAG, "cancel: close() of mmSocket in Connectthread failed. " + e.
                    getMessage());
            }
        }
    }

    // runs during a connection with a remote device
    private class IOThread extends Thread {
        private final BluetoothSocket bluetoothSocket;
        private final InputStream inputStream;
```

```
    private final OutputStream outputStream;

255 IOThread(BluetoothSocket socket) {
        Log.d(TAG, "IOThread: Starting.");

        bluetoothSocket = socket;
        InputStream auxIn = null;
260     OutputStream auxOut = null;

        try {
            auxIn = socket.getInputStream();
            auxOut = socket.getOutputStream();
265     } catch (IOException ex) {
            ex.printStackTrace();
        }
        inputStream = auxIn;
        outputStream = auxOut;
270 }

    public void run() {
        buffer = new byte[1024]; // buffer for the stream
        int bytes;

        //Listening to the InputStream
        while (true) {
            try {
                // Reading from the InputStream

                bytes = inputStream.read(buffer);

                String incomingMessage = new String(buffer, 0, bytes);
                Log.d(TAG, "InputStream: " + incomingMessage);
285             // Send the obtained bytes to the UI Activity
                handler.obtainMessage(Drawer_Activity.MESSAGE_READ, bytes, -1,
                        buffer).sendToTarget();
                Log.d(TAG, "InputStream: " + "Msg Received :" + incomingMessage);


            } catch (IOException e) {
                Log.e(TAG, "write: Error reading Input Stream. " + e.getMessage() );
                Message msg = handler.obtainMessage(Drawer_Activity.MESSAGE_TOAST);
                Bundle bundle = new Bundle();
295             bundle.putString("Toast", "Device connection was lost");
```

```
                msg.setData(bundle);
                handler.sendMessage(msg);

                // Start the service over to restart listening mode
                ChatHandler.this.start();
            }
        }
    }

    // writing to OutputStream
    void write(byte[] buffer) {
        try {
            Log.d(TAG, "write: Writing to outputstream: " + Arrays.toString(buffer));
            outputStream.write(buffer);
            Log.d(TAG, "InputStream: " + "Msg Sent :" + Arrays.toString(buffer));
            handler.obtainMessage(Drawer_Activity.MESSAGE_WRITE, -1, -1, buffer).
                sendToTarget();
        } catch (IOException e) { Log.e(TAG, "write: Error writing to output stream. " +
            e.getMessage() );}
    }

    void cancel() {
        try {
            bluetoothSocket.close();
        } catch (IOException e) {
            e.printStackTrace();
            Log.e(TAG, "cancel: Error closing the socket " + e.getMessage() );
        }
    }
}

private synchronized void start() {
    // Cancel any thread
    if (connectThread != null) {
        connectThread.cancel();
        connectThread = null;
    }

    // Cancel any running thread
    if (ioThread != null) {
        ioThread.cancel();
        ioThread = null;
    }
```

```
        setState(LISTENING_STATE);
        if (acceptThread == null) {
            acceptThread = new AcceptThread();
            acceptThread.start();
        }
    }

    private synchronized void setState(int state) {
        this.state = state;
    }

    public synchronized void stop() {
        if (connectThread != null) {
            connectThread.cancel();
            connectThread = null;
        }

        if (ioThread != null) {
            ioThread.cancel();
            ioThread = null;
        }

        if (acceptThread != null) {
            acceptThread.cancel();
            acceptThread = null;
        }
        setState(NO_STATE);
    }
}
```

Listing 6.24: Code for bluetooth connection setup

6.3.3. Wi-Fi

Wi-Fi is the second communication technology adopted to guarantee effective control of the rover with the commands from the smartphone.

6.3.3.1. Wi-Fi Connection Setup

The Wi-Fi connection setup code implementation is listed in 6.25. As one would expect it follows a **client-server approach** represented in the mentioned listing by the classes **ServerClass** (line 218) and **ClientClass** (line 238). Note that both classes inherit from the **Thread** class since, once again, **concurrency** is a relevant factor. On one hand, the **ServerClass** it's responsible for creating a **ServerSocket** and specify the port that will be used in the connection. This type of protocol usually requires the determination of the IP address and port number that will be utilized. On the other hand, the **ClientClass** also has to specify the port used but also the forementioned IP. Due to the programming language option chosen in section 6.3, one is now limited to use the predetermined IP protocol version preemptively defined (in this case **IPv6**) what might not be optimal. Both classes use another class that inherits from **Thread**, the **WifiConnectionManager** class, to handle the **Wi-Fi input and output streams** of the **server and client sockets**, if the device is host or client, respectively. With this **WifiConnectionManager** class, one can manage the message exchange between the devices and assure its simultaneity.

```
public class WifiActivity extends AppCompatActivity{

    private static final String TAG = "WifiActivity";
    ImageButton return_button;
    TextView connection_status2;

    WifiManager wifiManager;
    WifiP2pManager mManager;
    WifiP2pManager.Channel mChannel;

    BroadcastReceiver mReceiver;
    IntentFilter mIntentFilter;

    List<WifiP2pDevice> peers = new ArrayList<WifiP2pDevice>();
    String[] deviceNames;
    WifiP2pDevice[] deviceBuffer;
    ListView wifi_devices_discovered;

    ServerClass serverClass;
    ClientClass clientClass;
    static WifiConnectionManager wifiConnectionManager;
    static boolean can_send_rec_msg = false;

    private Button send_wifi_msg;
    private TextView wifi_read_box;
```

```
    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.wifi_devices_discovered);

        wifi_read_box = (TextView) findViewById(R.id.wifi_msg_read);

        send_wifi_msg = (Button) findViewById(R.id.wifi_msg_btn);
        send_wifi_msg.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                if(can_send_rec_msg) {
                    String msg2send = "Hello World! - Android";
                    wifiConnectionManager.sendWifiMessage(msg2send.getBytes());
                }
                else {
                    showToast("Error: No connection established",4,SHORT_TOAST);
                }
            }
        });

        return_button = (ImageButton) findViewById(R.id.return_btn);
        return_button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(getApplicationContext(), Drawer_Activity.class);
                startActivity(intent);
            }
        });

        wifi_devices_discovered = (ListView) findViewById(R.id.wifi_list_view);
        wifi_devices_discovered.setOnItemClickListener(new AdapterView.OnItemClickListener()
            {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int position, long id)
                {
                final WifiP2pDevice device = deviceBuffer[position];
                WifiP2pConfig config = new WifiP2pConfig();
                config.deviceAddress = device.deviceAddress;

                mManager.connect(mChannel, config, new WifiP2pManager.ActionListener() {
                    @Override
```

```
                public void onSuccess() {
                    showToast("Acknowledge: Connected to " + device.deviceName,3,
                        LONG_TOAST);
                    connection_status2.setText(MessageFormat.format("Connected to {0}",
                        device.deviceName));
                }

                @Override
                public void onFailure(int reason) {
                    showToast("Error: Unable to connect to " + device.deviceName,4,
                        LONG_TOAST);
                    connection_status2.setText("Connection Fail");
                }
            });
        }
    });


    connection_status2 = (TextView) findViewById(R.id.connection_status2);

    wifiManager = (WifiManager) getApplicationContext().getSystemService(Context.
        WIFI_SERVICE);
    mManager = (WifiP2pManager) getSystemService(Context.WIFI_P2P_SERVICE);
    mChannel = mManager.initialize(this,getMainLooper(),null);

    mReceiver = new WifiDirectBroadcastReceiver(mManager,mChannel,this);
    mIntentFilter = new IntentFilter();
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION);
    mIntentFilter.addAction(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION);

    mManager.discoverPeers(mChannel, new WifiP2pManager.ActionListener() {
        @Override
        public void onSuccess() {
            showToast("Acknowledge: Wifi discovery started",3,LONG_TOAST);
            connection_status2.setText(R.string.disc_start);
        }

        @Override
        public void onFailure(int reason) {
            showToast("Error: Wifi discovery failed, please enable android location",4,
                LONG_TOAST);
```

```
                connection_status2.setText(R.string.disc_fail);
            }
        });

    }

    @Override
    public void onConfigurationChanged(Configuration newConfig) {
        // ignore orientation/keyboard change
        super.onConfigurationChanged(newConfig);
    }

    WifiP2pManager.PeerListListener peerListListener = new WifiP2pManager.PeerListListener()
        {
        @Override
        public void onPeersAvailable(WifiP2pDeviceList peerList) {
            if(!peerList.getDeviceList().equals(peers)){
                peers.clear();
                peers.addAll(peerList.getDeviceList());

                deviceNames = new String[peerList.getDeviceList().size()];
                deviceBuffer = new WifiP2pDevice[peerList.getDeviceList().size()];

                int i = 0;
                for(WifiP2pDevice device : peerList.getDeviceList()){
                    deviceNames[i] = device.deviceName;
                    deviceBuffer[i] = device;
                    i++;
                }

                ArrayAdapter<String> adapter = new ArrayAdapter<String>(getApplicationContext
                    (), android.R.layout.simple_list_item_1,deviceNames);
                wifi_devices_discovered.setAdapter(adapter);
            }

            if(peers.size() == 0){
                showToast("Warning: No devices found", 5, SHORT_TOAST);
            }
        }
    };

    /****************
     * WIFI HANDLER *
```

```
     ***************/

    static final int MESSAGE_READ_WIFI = 1;

    Handler wifiHandler = new Handler(new Handler.Callback() {
        @Override
        public boolean handleMessage(Message msg) {
            switch (msg.what){
154                 case MESSAGE_READ_WIFI:
                        byte[] readBuffer = (byte[]) msg.obj;
                        String Msg = new String(readBuffer,0,msg.arg1);
                        wifi_read_box.setText(Msg);
                        break;

            }
            return true;
        }
    });

    /**************************
     * WIFI CONNECTION MANAGER *
     **************************/

169 class WifiConnectionManager extends Thread{
        private Socket wifiSocket;
        private InputStream wifiInputStream;
        private OutputStream wifiOutputStream;

174         //Constructor
        public WifiConnectionManager(Socket paramSocket){
            wifiSocket = paramSocket;

            try {
179                 wifiInputStream = wifiSocket.getInputStream();
                    wifiOutputStream = wifiSocket.getOutputStream();
            } catch (IOException e) {
                e.printStackTrace();
                Log.e(TAG, "WifiConnectionManager: Couldn't get Input/Output Stream");
184             }
        }

        @Override
        public void run() {
```

```
189             super.run();
                byte[] wifiBuff = new byte[1024];
                int num_bytes_read;

                while(wifiSocket != null){
194                 try {
                        num_bytes_read = wifiInputStream.read(wifiBuff); //-1 returned if there's
                             no more data

                        if(num_bytes_read>0){ //message valid
                            wifiHandler.obtainMessage(MESSAGE_READ_WIFI,num_bytes_read,-1,
                                wifiBuff).sendToTarget();
199                     }

                    } catch (IOException e) {
                        e.printStackTrace();
                        Log.e(TAG, "run: Failed Reading InputStream into wifi buffer");
204                 }
                }
            }

            public void sendWifiMessage(byte[] msgBytes){
209             try {
                    wifiOutputStream.write(msgBytes);
                } catch (IOException e) {
                    e.printStackTrace();
                    Log.e(TAG, "SendWifiMessage: Couldn't send message over Wifi");
214             }
            }
        }

        public class ServerClass extends Thread{
219         Socket socket;
            ServerSocket serverSocket;

            @Override
            public void run() {
224             try {
                    serverSocket = new ServerSocket(8888);
                    socket = serverSocket.accept();
                    //Communication
                    wifiConnectionManager = new WifiConnectionManager(socket);
229                 wifiConnectionManager.start();
```

```
            } catch (IOException e) {
                e.printStackTrace();
                Log.e(TAG, "run: Server Socket creation failed");
            }

        }
    }

    public class ClientClass extends Thread{
        Socket socket;
        String hostAddress;

        public ClientClass(InetAddress hostAddress){
            this.hostAddress = hostAddress.getHostAddress();
            socket = new Socket();

        }

        @Override
        public void run() {
            try {
                socket.connect(new InetSocketAddress(this.hostAddress,8888),500);
                //Communication
                wifiConnectionManager = new WifiConnectionManager(socket);
                wifiConnectionManager.start();
            } catch (IOException e) {
                e.printStackTrace();
                Log.e(TAG, "run: Socket Connection Failed - Client");
            }
        }
    }

    WifiP2pManager.ConnectionInfoListener connectionInfoListener = new WifiP2pManager.
        ConnectionInfoListener() {
        @Override
        public void onConnectionInfoAvailable(WifiP2pInfo info) {
            final InetAddress groupOwnerAddress = info.groupOwnerAddress;

            if(info.groupFormed && info.isGroupOwner){
                connection_status2.setText(R.string.host);
                serverClass = new ServerClass();
                serverClass.start();
                can_send_rec_msg = true;
```

```
            }
            else if(info.groupFormed){
                connection_status2.setText(R.string.client);
                clientClass = new ClientClass(groupOwnerAddress);
                clientClass.start();
                can_send_rec_msg = true;
            }
        }
    };

    @Override
    protected void onStart() {
        super.onStart();

    }

    @Override
    protected void onResume() {
        super.onResume();
        registerReceiver(mReceiver, mIntentFilter);
    }

    @Override
    protected void onPause() {
        super.onPause();
        unregisterReceiver(mReceiver);
    }
}
```

Listing 6.25: Code for Wi-fi connection setup

6.3.3.2. Wi-Fi Video Feed

One important user feature defined in the design of the smartphone application in section 5.4 was the capability of watching the RVVS' live camera feed within the app. The applied solution is based on the **Youtube Android Player API** [16] as it enables a more **reliable video stream** whilst ensuring **project off-load**. Resorting to the latter, one can also tackle the fullscreen video display which was a big concern in terms of user-friendliness. Nevertheless, one can still claim that a framewise transmission approach via Wi-Fi is even more fail-safe and secure since one doesn't need to rely on an external entity for video communication. Despite this, the framewise approach was discarded to ensure deadline meeting, hence

its implementation required some prior subject experience or further research. The implementation of the abovementioned feature is depicted in listing 6.26.

```
public class Video extends YouTubeBaseActivity{

    private static final int RECOVERY_REQUEST = 1;
    YouTubePlayerView mYoutubePlayerView;
    YouTubePlayer.OnInitializedListener monInitializedListener;
    ImageButton return_btn2;
    private MyPlayerStateChangeListener playerStateChangeListener;
    private MyPlaybackEventListener playbackEventListener;

    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView(R.layout.youtube_video);
        return_btn2 = (ImageButton) findViewById(R.id.return_btn2);
        return_btn2.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(getApplicationContext(), Drawer_Activity.class);
                startActivity(intent);
            }
        });
        mYoutubePlayerView = (YouTubePlayerView) findViewById(R.id.Youtube_display);
        playerStateChangeListener = new MyPlayerStateChangeListener();
        playbackEventListener = new MyPlaybackEventListener();
        monInitializedListener = new YouTubePlayer.OnInitializedListener() {
            @Override
            public void onInitializationSuccess(YouTubePlayer.Provider provider,
                YouTubePlayer youTubePlayer, boolean b) {
                Log.d("Video", "onNavigationItemSelected: Done initializing");
                youTubePlayer.setPlayerStateChangeListener(playerStateChangeListener);
                youTubePlayer.setPlaybackEventListener(playbackEventListener);
                String video_url = "y7e-GC6oGhg";
                youTubePlayer.loadVideo(video_url);
            }

            @Override
            public void onInitializationFailure(YouTubePlayer.Provider provider,
                YouTubeInitializationResult youTubeInitializationResult) {
                Log.d("Video", "onNavigationItemSelected: Fail initializing");
```

```
                showToast("Video initialization failed",4,SHORT_TOAST);
            }
        };

        mYoutubePlayerView.initialize(YoutubeConfig.getApiKey(), monInitializedListener);
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == RECOVERY_REQUEST) {
            // Retry initialization if user performed a recovery action
            getYouTubePlayerProvider().initialize(YoutubeConfig.getApiKey(),
                monInitializedListener);
        }
    }

    protected YouTubePlayer.Provider getYouTubePlayerProvider() {
        return mYoutubePlayerView;
    }

    private final class MyPlaybackEventListener implements YouTubePlayer.
        PlaybackEventListener {

        @Override
        public void onPlaying() {
            // Called when playback starts, either due to user action or call to play().
            showToast("Streaming RFCAR camera",6,SHORT_TOAST);
        }

        @Override
        public void onPaused() {
            // Called when playback is paused, either due to user action or call to pause().
            showToast("Stream Paused",6,SHORT_TOAST);
        }

        @Override
        public void onStopped() {
            // Called when playback stops for a reason other than being paused.
        }

        @Override
        public void onBuffering(boolean b) {
            // Called when buffering starts or ends.
```

```
        }

81      @Override
        public void onSeekTo(int i) {
            // Called when a jump in playback position occurs, either
            // due to user scrubbing or call to seekRelativeMillis() or seekToMillis()
        }
86  }

    private final class MyPlayerStateChangeListener implements YouTubePlayer.
        PlayerStateChangeListener {

        @Override
91      public void onLoading() {
            // Called when the player is loading a video
            // At this point, it's not ready to accept commands affecting playback such as
                play() or pause()
        }

96      @Override
        public void onLoaded(String s) {
            // Called when a video is done loading.
            // Playback methods such as play(), pause() or seekToMillis(int) may be called
                after this callback.
        }

        @Override
        public void onAdStarted() {
            // Called when playback of an advertisement starts.
        }

        @Override
        public void onVideoStarted() {
            // Called when playback of the video starts.
        }

        @Override
        public void onVideoEnded() {
            // Called when the video reaches its end.
        }

        @Override
        public void onError(YouTubePlayer.ErrorReason errorReason) {
```

```
            // Called when an error occurs.
        }
    }
```

Listing 6.26: Code for video feed view within the application

6.3.4. User Interface

The UI implementation consists of various **XML files complemented with java code**. Consequently, one can consider the UI features and present images related to those rather than simply overload the document with XML code. Firstly, as one enters the app, an initial screen is presented (figure 6.6 - **1**), in this screen the app checks for if the phone supports the Bluetooth services necessary to run the application and requests user permission to enable Bluetooth (figure 6.6 - **2**). The second and main screen of the app is composed of a navigation drawer (figure 6.6 - **4**) and a screen with two buttons for initiating the video stream (figure 6.6 - **3**). The navigation drawer is the place where the user can perform all the communication-related actions like enabling Wifi or setting up a Wi-fi or Bluetooth connection. The aforesaid main screen also as has a status indicator. As one enters the rover control screen (figure 6.6 - **11**) the orientation is automatically changed to the one (preemptively defined) that allows the correct initial position for the car control. In this screen, one can see the RVVS' live transmission and the speed and wheel tilt percentages sent to the same module. The video allows the fullscreen display (figure 6.6 - **9,10,12**), still showing the latter percentages as a periodic pop-up (figure 6.6 - **9**). Note that one tried to achieve a balance between app functionality while also creating a user-friendly environment.

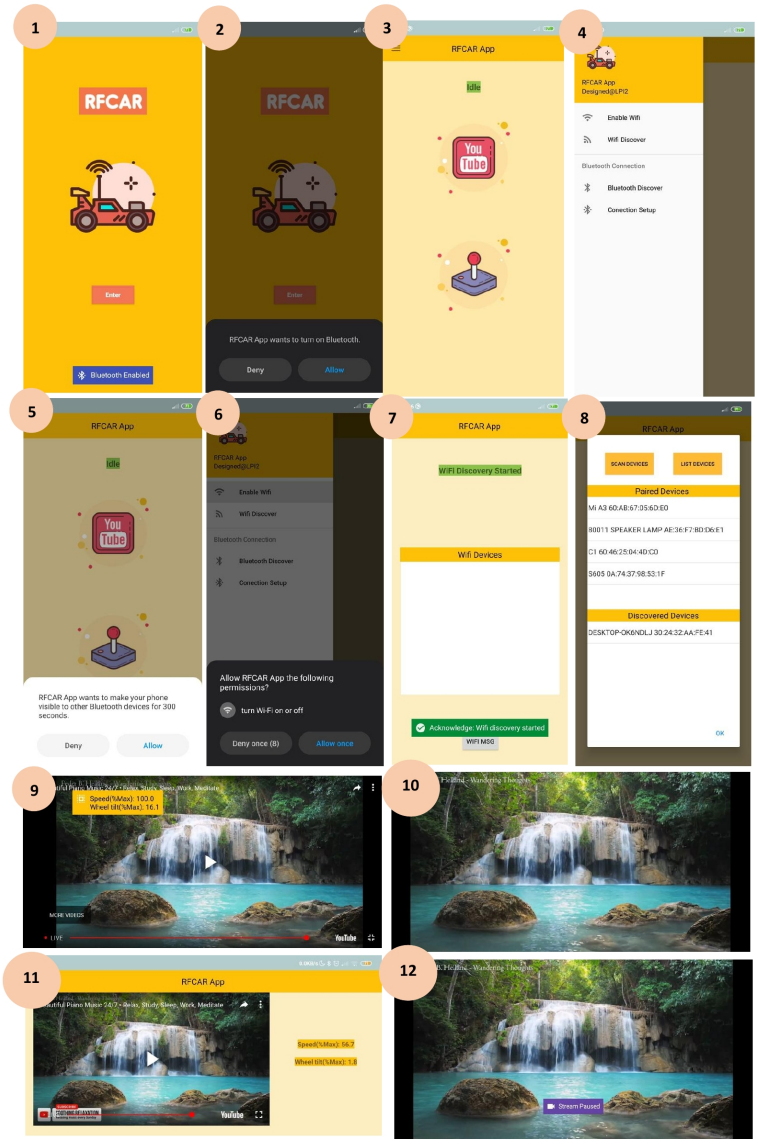

Figure 6.6.: General App Overview

7. Testing

After implementing the solution developed in the various domains into the target platforms, the system's behaviour is tested in several levels of granularity: at the subsystem level — unit testing, and system level — integrated testing. The idea is progressively test the behaviour of each subsystem and its integration into the system. In this chapter are presented the unit testing and integrated testing for the RFCAR.

7.1. Unit testing

The unit tests are briefly described in this section.

7.1.1. Navigation Virtual Subsystem

The NVS susbsystem tests are presented next. The stack implementation tests were mainly focused on the interactions established between the packages at play and the surrounding systems both lower-tiered and high-tiered.

7.1.1.1. IO: Input/Output Package

The IO package testing (figure 7.1) consisted in the creation of a GPIO object with which one could simulate motor PWM outputs that would generate a file, through which one would simulate the IR sensor readings or motor pulse readings. The first GPIO test relied on a configuration of an object in OUTPUT_PWM mode. With the latter test, it was possible to write three 32 bit integers to a binary file and observe the results with notepad++. The second test hinged on reading from the forementioned binary file, expecting to retrieve the corresponding values. Should the latter be as expected the test could be considered a success.

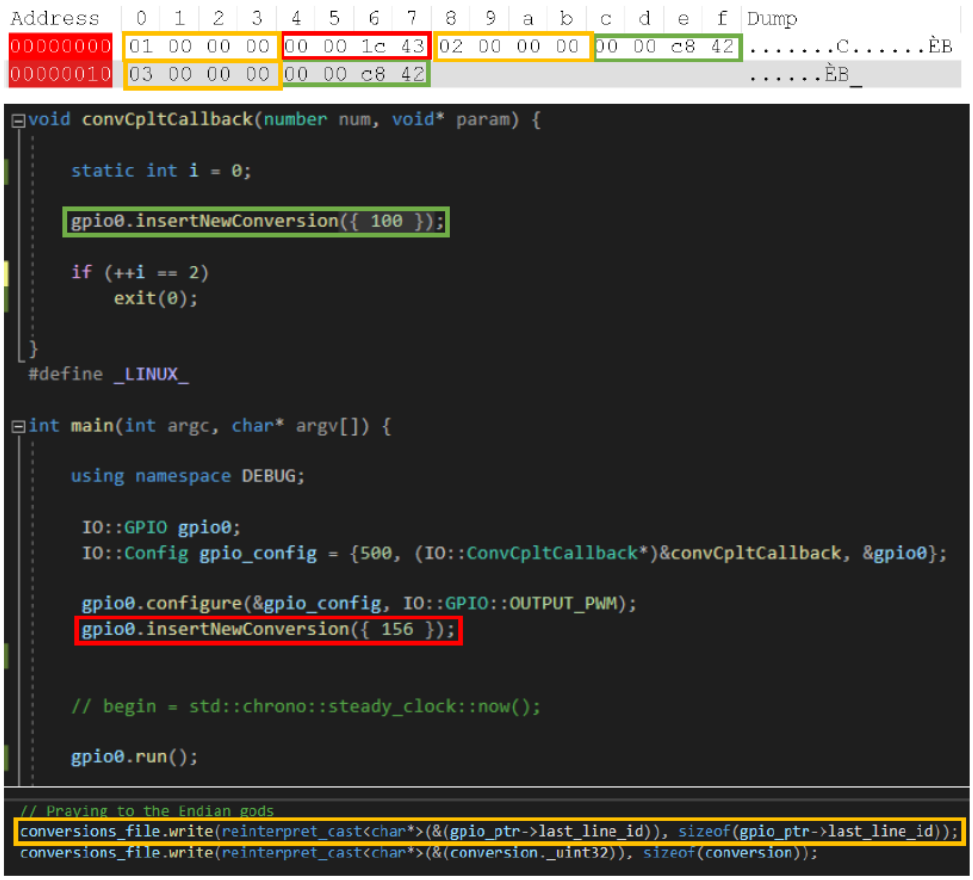

Figure 7.1.: IO package OUTPUT_PMW mode tests

7.1.1.2. COM: Communications Package

The COM package testing involved effectuating some experiments between the NVS and the smartphone (using Bluetooth) and the former and the RVVS (using RS232). These experiments will be performed in section 7.1.1.5.

7.1.1.3. OS: Scheduler Package

The OS package testing was based on the creation of two types of Threads:

- **Producer Thread (P)**
- **Consumer Thread (C)**

The OS package testing was based on the creation of two types of Threads. These tests were made to simulate the way how the operating system handles multi-thread contexts. The Producer Thread type

publishes values in a list while the Consumer Thread type removes values from the latter. The introduction of an initial delay in the Producer Thread function allows the OS to assign the CPU more evenly. The test is represented in figure 7.2.

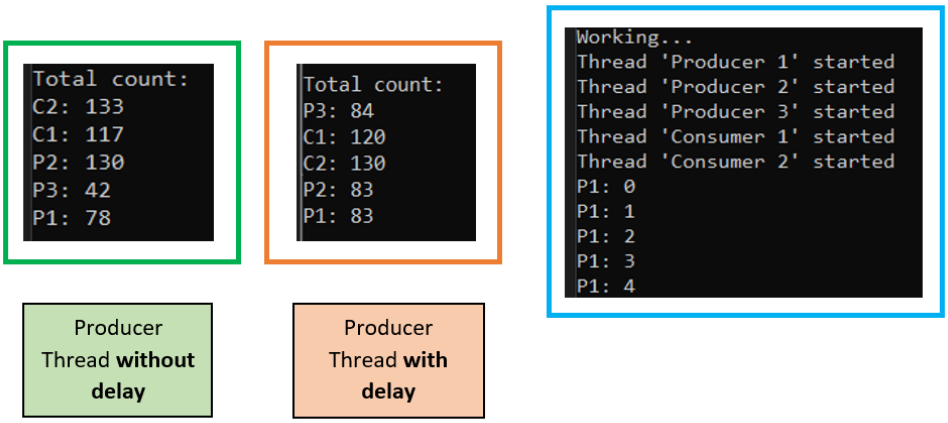

Figure 7.2.: OS package tests

7.1.1.4. MEM: Memory Structures Package

The MEM Package testing consisted of creating two projects, one that used linked lists and another that used a circular buffer. In the former, one simply pushed numerical characters and verified that the appropriate memory spaces were filled. Afterwards, those characters were popped and the result was as expected, the memory spaces were removed from the list. This test is depicted in figure **??**.

```
char_list.push(buff2[0], Position::BACK);
char_list.push(buff2[1], Position::BACK);
char_list.push(buff2[2], Position::FRONT);
char_list.push(buff2[3], Position::FRONT);

char_list.pop(buff2[2]);
char_list.pop(buff2[3]);
char_list.pop(buff2[0]);
```

Figure 7.3.: MEM package linked list tests excerpt

7.1.1.5. CLK: Timing Package

The CLK package testing (figure 7.4) included the creation of a Timer object, associating it to a callback and proceed to verify if the callback was called at the specified time preemptively defined in the Config object configuration.

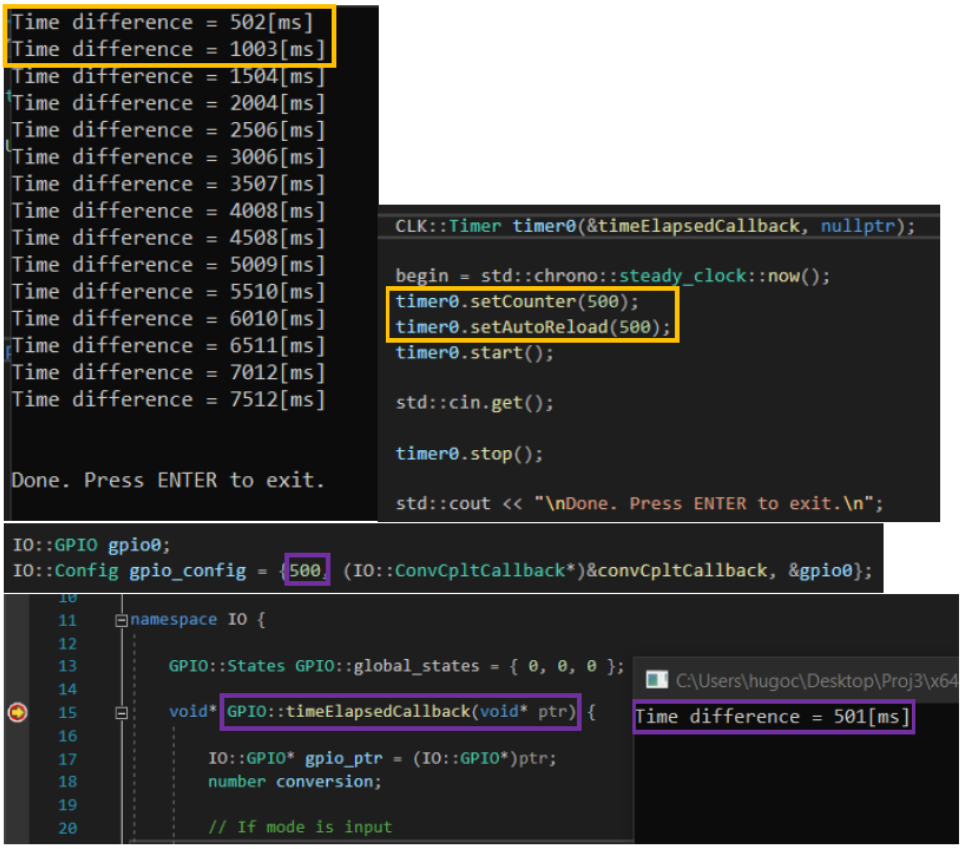

Figure 7.4.: CLK package tests

7.1.1.6. System response

The implementation of the control module was made using the modified speed algorithm mentioned and explained in 5.1.1.3.

The tests for this module consists in the comparison between the results obtained and simulated in 5.1.1.6. Starting with speed reference = 1 m/s and $\theta = 0$ rad:

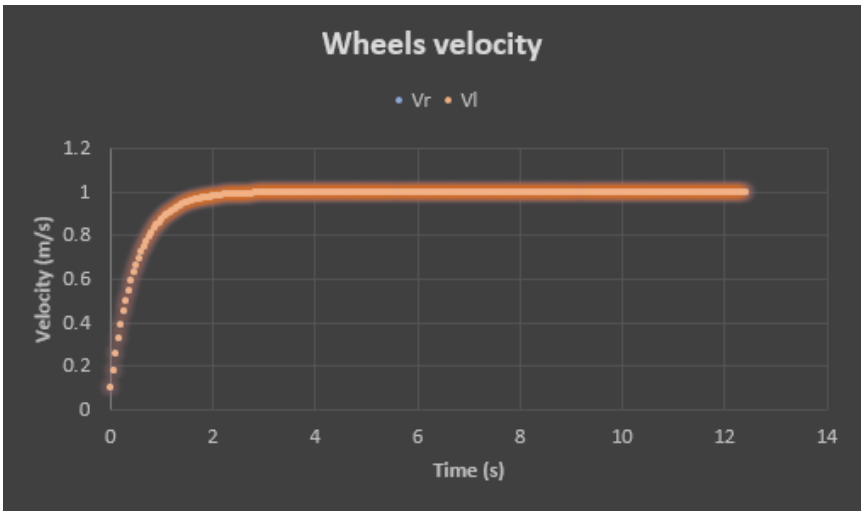

Figure 7.5.: Wheels velocity v=1m/s,$\theta = 0$ rad

In the figure 7.5 shows that the velocity of both wheels reaches the reference value in nearly 1.5 seconds, while in the simulation 5.8 it takes about 3 seconds. This discrepancy is due to the controller in use. The simulations were made using the PID block provided by Simulink, while in the implementation, the controller algorithm used was as mentioned before, the modified speed algorithm. With this algorithm the behavior of the car is the same as with the traditional PID, but faster.

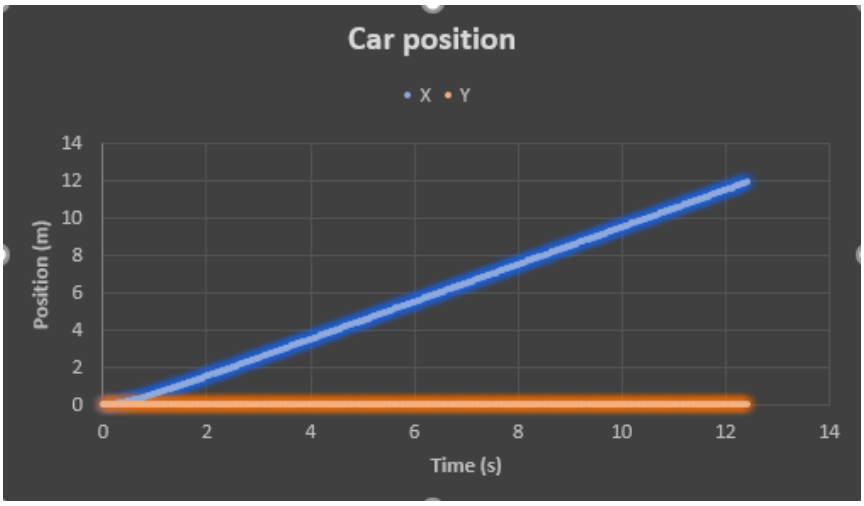

Figure 7.6.: Position v=1m/s,$\theta = 0$ rad

In the figure 7.6 it can be observed that the position of the car is the same as in the simulation 5.9. With speed reference = 1m/s and $\theta = 0.1$ rad:

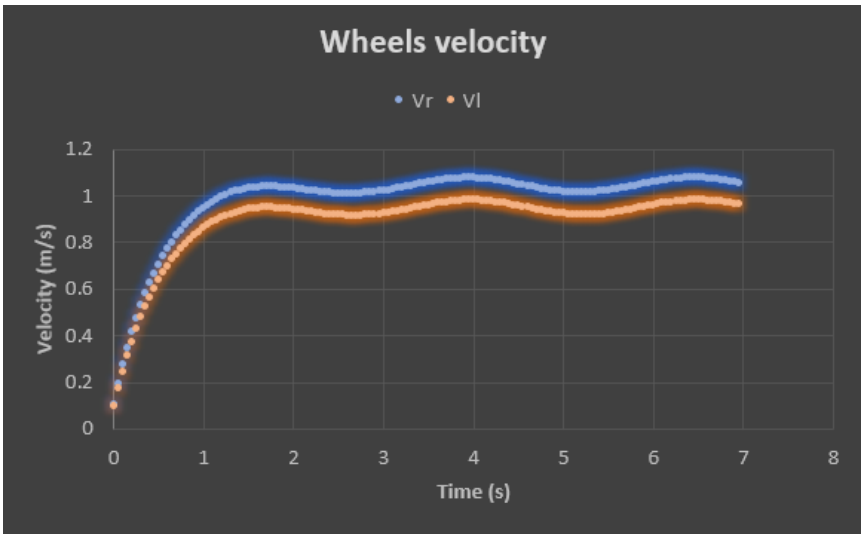

Figure 7.7.: Wheels velocity v=1m/s,$\theta = 0.1$ rad

In the figure 7.7 comparing to 5.10 it's possible to see that once again the time it takes to reach steady state is smaller for the reason mention before. In the implemented algorithm it's also possible to see that the steady value of the velocity of both wheels has a ripple around the reference value.

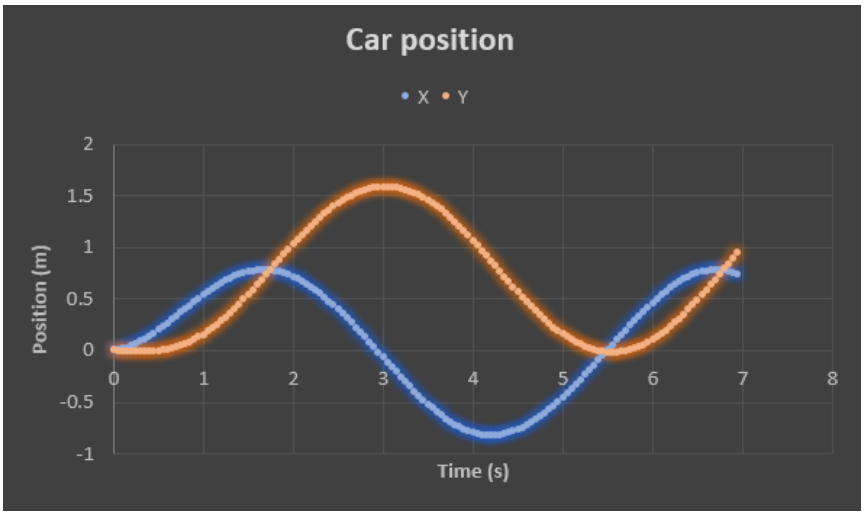

Figure 7.8.: Position v=1m/s,$\theta = 0.1$ rad

In the figure 7.8 it is present the position of the car. In comparison to the simulated 5.11 it is possible to see that the results are the same, thus validating the implementation.

7.1.1.7. Obstacle Avoidance Through Odometric Sensors

The car composed by 9 odometric sensors radially distributed, 40 degrees apart.
In order to test if the car can avoid obstacles, the values for the sensors were generated, and the behavior of the car was as follows:

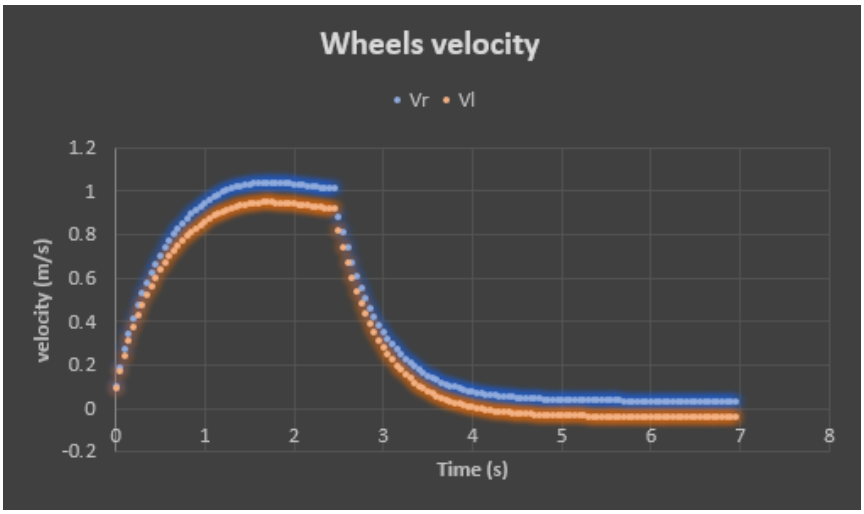

Figure 7.9.: Wheels velocity

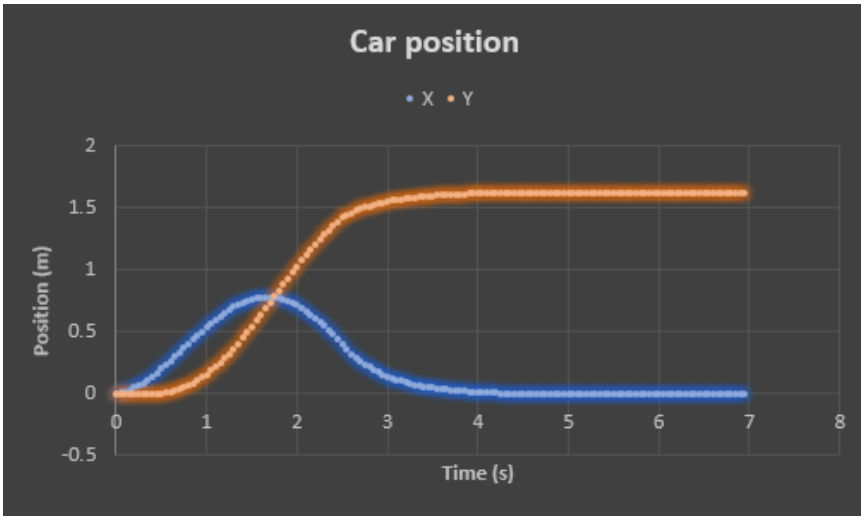

Figure 7.10.: Car position

In the figure 7.9 it is possible to see that at t=2.5 seconds the velocity of both wheels starts decreasing until they reach a value near 0 m/s. Even though the velocity of both wheels is not exactly 0 m/s, the values

are too small to break the inertia of the wheels, so the car effectively stops. In the figure 7.10 it is possible to see that the position of the car remains the same after the decrease of the velocity of the wheels, proving that the car is not moving.

7.1.2. Remote Vision Virtual Subsystem

In this section are presented the tests performed on the RVVS subsystem in the relevant domains.

7.1.2.1. Image Acquisition

The image acquisition system, implemented in Section 6.2.2.1, was tested out to assess its behaviour.

Firstly, it was selected the built-in webcam from host and attached to the guest VM. However, and despite extensive troubleshooting, the bypass suggested for web cameras[17] was not possible for a Mac OS host. Thus, a quick alternative was to flash a Linux OS onto a Storage Disk (SD) card and plug it to a Raspberry Pi 3 (available) and connect an external Universal Serial Bus (USB) camera to it (Fig. 7.11).

Figure 7.11.: Raspberry Pi 3 + Webcam testing setup

The deployment was performed using the Secure Shell (SSH) protocol to connect to the Raspberry Pi and execute the necessary commands on it, and using the Secure Copy (SCP) command that uses the Secure File Transfer Protocol (SFTP) protocol to transfer files between host and guest (Listing 7.1).

```
ssh pi@192.168.1.10 -v
scp -r webcam pi@192.168.1.10/Documents/
```

Listing 7.1: Deployment commands targetting Raspberry Pi

The web camera was then tested using the **lsusb** command and piped the output to a **.txt** file for further examination. Listing 7.2 contains an excerpt of this output where it can be observed the webcam model (**Cubeternet WebCam**) and the USB2.0 interface.

```
Bus 001 Device 007: ID 1e4e:0100 Cubeternet WebCam
Device Descriptor:
  bLength                18
  bDescriptorType         1
  bcdUSB               2.00
  bDeviceClass          239 Miscellaneous Device
  bDeviceSubClass         2
  bDeviceProtocol         1 Interface Association
  bMaxPacketSize0        64
  idVendor           0x1e4e Cubeternet
  idProduct          0x0100 WebCam
  bcdDevice            0.02
  iManufacturer           1 Etron Technologies
  iProduct                2 USB2.0 Camera
  iSerial                 0
  bNumConfigurations      1
```

Listing 7.2: Webcam information obtained via **lsusb** (excerpt)

However, this information by itself is not so useful. Additionally, and much more importantly, one wants to check the capabilities of the device and the supported formats. For that purpose it was used the **v4l2-ctl** utility (Listing 7.3). It can be observed that the only format supported is YUYV, which is a colour space based on one luminance component (Y') and two chrominance components, called U(blue) projection and V(red projection), respectively. The framerate is also fixed (30 fps), but with different resolutions. This is very limitative, as the newer webcams support MJPEG formats, easing video capture.

```
# determing devices connected and its nodes
raspberrypi:~$ v4l2-ctl --list-devices
bcm2835-codec-decode (platform:bcm2835-codec):
  /dev/video10
  /dev/video11
  /dev/video12

USB2.0 Camera: USB2.0 Camera (usb-3f980000.usb-1.4):
  /dev/video0
  /dev/video1

# determing device parameters for /dev/video0
```

```
pi@raspberrypi:~$ sudo v4l2-ctl -d /dev/video0 --get-parm
Streaming Parameters Video Capture:
  Capabilities     : timeperframe
  Frames per second: 30.000 (30/1)
  Read buffers     : 0

# determing image formats using v4l2-ctl
pi@raspberrypi:~$ v4l2-ctl --list-formats-ext
ioctl: VIDIOC_ENUM_FMT
  Type: Video Capture
  [0]: 'YUYV' (YUYV 4:2:2)
    Size: Discrete 640x480
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 352x288
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 320x240
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 176x144
      Interval: Discrete 0.033s (30.000 fps)
    Size: Discrete 160x120
      Interval: Discrete 0.033s (30.000 fps)
```

Listing 7.3: Webcam supported image formats, resolutions, and framerates

Effectively, it was executed the driver program for the webcam interface (Listing 6.20), but it reported the unsupported format as expected (Listing 7.4).

```
pi@raspberrypi:~/Documents/webcam$ ./webcam-main
Unable to set image format: Input/output error
```

Listing 7.4: Webcam driver program error: unsupported image format

Then, it was tried out the **UYVY422** format by modifying the following lines into Listing 6.20 (Listing 7.5)

```
/* Set format to 320x240 and Motion JPEG */
    wbc.setFormat(320, 240, V4L2_PIX_FMT_UYVY);
/* Start stream and write to file */
    wbc.startStream("output.yuvy");
```

Listing 7.5: Webcam driver program error: modifications to support **UYVY422** format

Although an image in the **UYVY422** format was obtained, **ffmpeg** requires tedious conversions between the packed format (**UYVY422**) to planar one (**JPG**). Thus, an online tool was used to convert this image [18] into **jpg** format. In Fig. 7.12 can be seen that an image was successfully acquired.

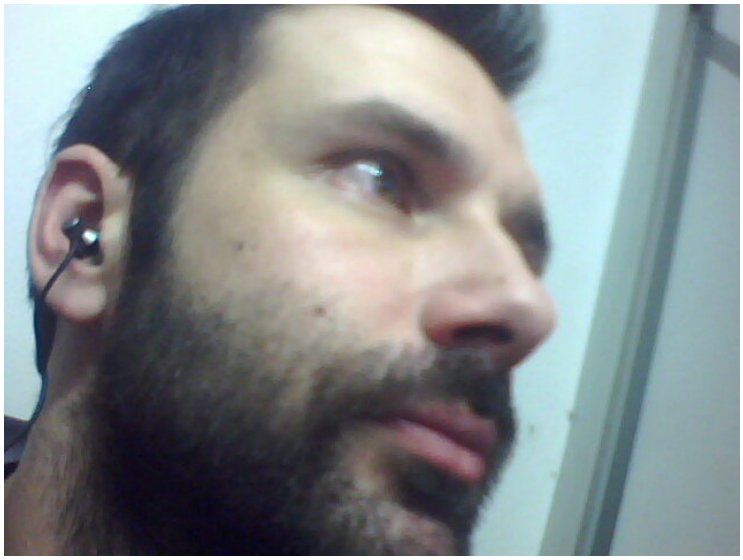

Figure 7.12.: Raspberry Pi 3 + Webcam test: success

However, this is a tedious process that could be avoided by using a different, newer, web camera supporting additional formats and resolutions, which limited severely the implementation and testing.

7.1.2.2. Wi-Fi

The Wi-Fi communications workflow, implemented in Section 6.2.2.2, was tested out to assess its behaviour, based on the client/server architecture in a concurrent execution scenario.

For this purpose, a small driver program was used (7.6), encapsulating each role on the network — client and server — in the respective thread and then execute them. The server thread creates a socket, binds to a port and address and listens for incomming connections. If this happens, it tries to accept the connection from a client, and tries to read any incoming data, prints it out and then exits. The client thread, on the other hand, tries to connect to the **localhost**, on IP address **127.0.0.1**, and if sucessful, sends a message to the server.

```
1 #include <iostream>
  #include <string>
  #include <list>
  #include <utility>
  #include <cstdio>
```

```
#include <map>

#include "main.hpp"
#include "Thread.hpp"
#include "WCOM_LL.hpp"

#define PORT_TEST 4545
#define BUFFER_SZ ((uint32_t) 64)

void tcp_socketServerTest(OS::Thread*) {
    WCOM::Error error;
    std::string error_str;
    WCOM::LL<WCOM::Protocol::TCP, WCOM::Role::SERVER> server(PORT_TEST);
    char buffer[30];

    // Serves to demonstrate that a pointer to the generic class can be used to access
        methods methods from the specialized classes
    WCOM::LL<>* server_ptr = &server;
    WCOM::LL<WCOM::Protocol::TCP, WCOM::Role::SERVER>* server_ptr2 = \
        static_cast<WCOM::LL<WCOM::Protocol::TCP,
                             WCOM::Role::SERVER>*>(server_ptr);

    std::cout << "FUN[TCP socketServerTest]: STARTED\n";

    // Create socket, bind and listen for connection
    error = server_ptr2->listenConnection();
    server_ptr->getLastError(error_str);
    std::cout << "SERVER   " << error_str;
    if (error) return;

    // Accept connection
    error = server.acceptConnection();
    server_ptr->getLastError(error_str);
    std::cout << "SERVER   " << error_str;

    // If there was an error during acceptance, return immediately.
    // The rest has been taken care of by the socket.
    if (error) return;

    while(1) {

        char temp_buffer[BUFFER_SZ];
```

```
        // Read string into temporary buffer to demonstrate how a message could be read
            continuously
        server_ptr->readStr(temp_buffer, BUFFER_SZ - 1);
        error = server_ptr->getLastError(error_str);
        std::cout << "SERVER    " << error_str;

        if (error)
            break;

        // Concatenate temporary string with permanent one
        strcat(buffer, (const char*)temp_buffer);

        // Check for reception of the message termination character
        if (strchr(buffer, '\n') != NULL) {
            // When finished, ptint the message and close the connection
            std::cout << "FUN[TCP socketServerTest]: Message received: "
                      << buffer;
            break;
        }
    }
    server.closeConnection();
    return;
}


void tcp_socketClientTest(OS::Thread*) {
    WCOM::Error error;
    std::string error_str;
    WCOM::LL<WCOM::Protocol::TCP, WCOM::Role::CLIENT> client(PORT_TEST);
    WCOM::LL<>* client_ptr = &client;

    char buffer[BUFFER_SZ] = "This is a message\n";

    std::cout << "FUN[TCP socketClientTest]: STARTED\n";

    std::string serv_addr = "127.0.0.1"; /**< localhost */

    // Open connection
    client.Connect(serv_addr, PORT_TEST);
    error = client_ptr->getLastError(error_str);
    std::cout << "CLIENT    " << error_str;

    if (error) return;
```

```
91      // Send message
        client_ptr->writeStr(buffer, sizeof(buffer));
        error = client_ptr->getLastError(error_str);
        std::cout << "CLIENT    " << error_str;
    }

    int main(int argc, char* argv[]) {

        using namespace std::chrono_literals;
        OS::Thread server_thread("Socket server test", tcp_socketServerTest);
101     OS::Thread client_thread("Socket client test", tcp_socketClientTest);

        std::cout << "\n---------------------------\n";
        std::cout << "Working...\n\n";

106     // Start server
        server_thread.run();
        // Wait for the server to configure. Anothr option would be to make
        // the client try to establish a connection indefinitely.
        std::this_thread::sleep_for(1s);
111     client_thread.run();

        // Synchronize the tasks
        server_thread.join();
        client_thread.join();
116     std::cout << "Joined!" << std::endl;

        return 0;
    }
```

Listing 7.6: Wi-Fi client/server driver program

Listing 7.7 presents the previous program's output, running on the Raspberry Pi. It can be observed that client and server exchange a message, thus, validating the implementation of the client/server model for the Wi-Fi communication.

```
1  pi@raspberrypi:~/Documents/wifi/sockets$ ./wcom

   ---------------------------
   Working...
```

```
FUN[TCP socketServerTest]: STARTED
SERVER   OK: Listening for connection
FUN[TCP socketClientTest]: STARTED
CLIENT   OK: Open connection
SERVER   OK: Acepted connection
SERVER   OK: Read string
SERVER   OK: Read string
SERVER   OK: Read string
SERVER   OK: Read string
FUN[TCP socketServerTest]: Message received: This is a message
Joined!
pi@raspberrypi:~/Documents/wifi/sockets$
```

Listing 7.7: Wi-Fi client/server driver program

7.1.3. Smartphone

7.1.3.1. Sensor Interaction

As referred in section 6.3.1.2, a ball movement application was made to test the accuracy of the linear acceleration values attained from the phone's accelerometer. One can observe the results in figures 7.13 and 7.14. In the **first case** (figure 7.13), the phone is raised on the left side but slightly downwards so the ball moves to the bottom right corner, as expected. On the **second case** presented (figure 7.14), the phone is also raised on the left side but this time slightly upwards making the ball to move from the initial position to somewhere near the top right corner. The linear acceleration values can be later used for generating the control commands of the rover since the **values were proven to be accurate**. As a way to test the code of the interaction with the rotation sensor, the values calculated were displayed on screen to perform a quick analysis, depicted in figure 7.15. One must notice the method used based in control percentage values since it allows the use of different rover components maintaining the way these values are calculated.

7.1.3.2. Bluetooth

After effectively changing the direction of a ball according to the accelerometer's state in the section 7.1.3.1, one could only need to merge its code with the Bluetooth connection setup, so that, instead of sending plain text messages, one can transfer the acceleration values.

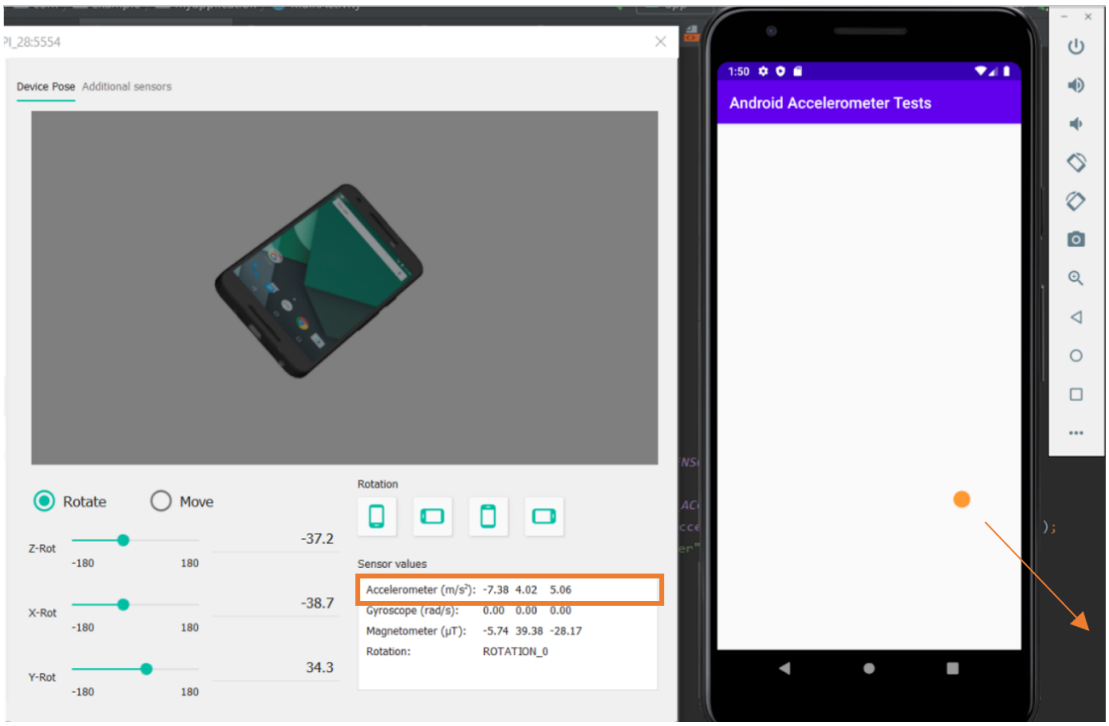

Figure 7.13.: Accelerometer ball movement tests - case 1

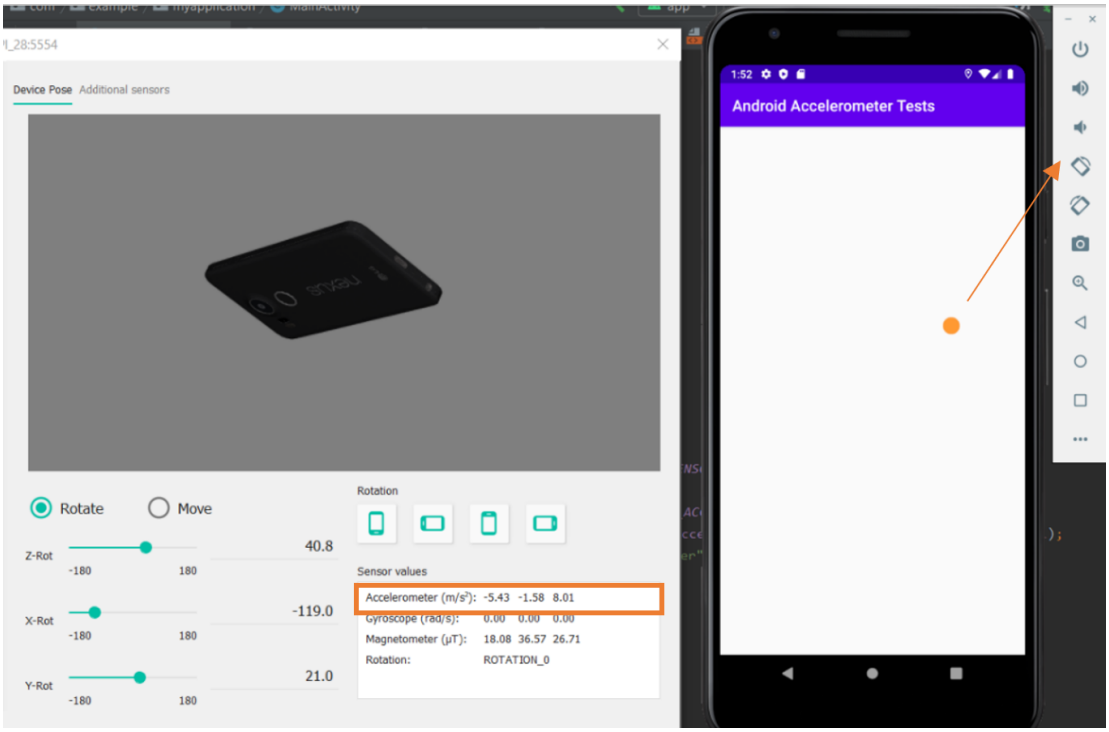

Figure 7.14.: Accelerometer ball movement tests - case 2

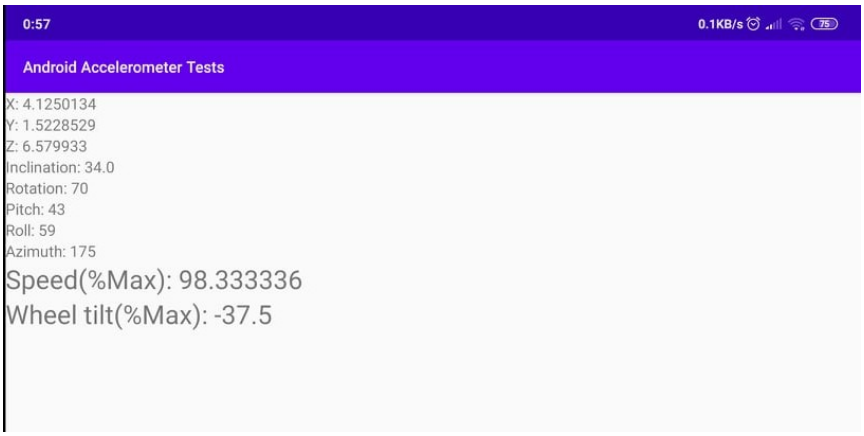

Figure 7.15.: Rotation sensor tests

7.1.3.3. Bluetooth: Smartphone-Smartphone

The first test was based on testing the Bluetooth connection setup when the interface was comprised of two distinct smartphones. With that in mind, the connection was successful and the devices were able to exchange messages, as expected. Since this test was performed using the code done in another course unit no further in-depth explanations will be presented in this section as the code was fully functional.

7.1.3.4. Wi-Fi

After the implementation of the Wi-Fi module forementioned in 6.3.3.1, one needed to perform tests in the connection between two smartphones and also between the smartphone and the RVVS. These tests will be discussed in the following sections.

7.1.3.5. Wi-Fi: Smartphone-Smartphone

For the tests related to the smartphone's wifi module, one tested the connection setup (figure 7.16) and the capability of message exchange (figure 7.17). Firstly, Wi-Fi must be enabled using the enable Wi-Fi button, accepting the request to turn it on. Next, one should press the Wifi Discover button, advancing to the subsequent screen, where a list of the available devices for connection is displayed. The green text on top of the screen refers to the current status of the application in terms of the wifi setup. When the user presses a list from the list, a request pop-up for Wi-Fi connection is presented. Accepting the request, the status changes to connected and the host and client are identified. Lastly, one tested sending and receiving messages from a smartphone to another smartphone by simply trying to press a WIFI MSG button to send a specific message, but due to deadline proximity, this feature wasn't functional and the Wi-Fi

smartphone-RVVS tests were abandoned.

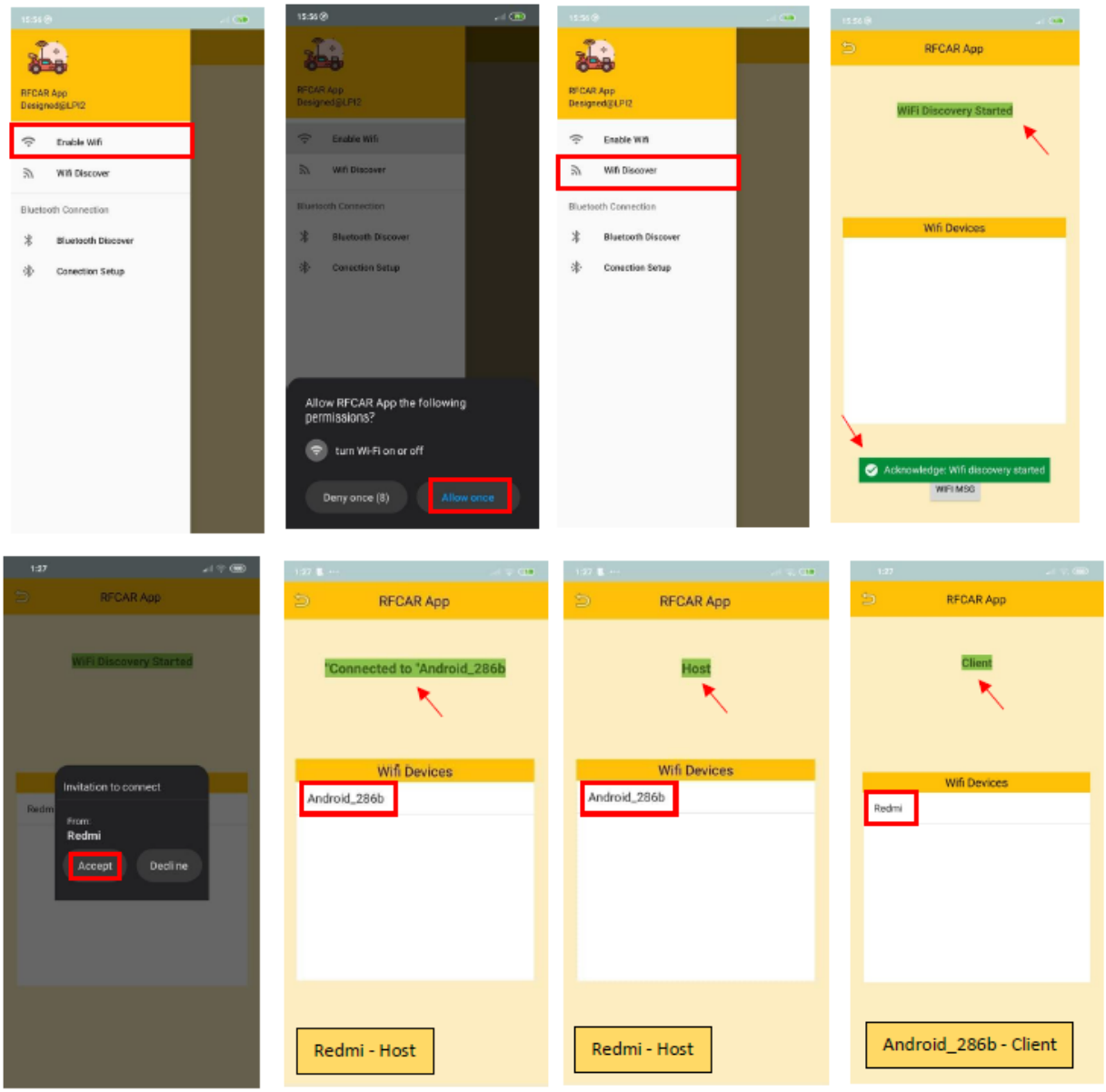

Figure 7.16.: Wi-Fi connection setup tests

7.1.3.6. User Interface

In the beginning, an app UI was initially thought out and made a first test design. When the user opens the program, the image of the RFCAR appears along with a built-in set of features. It is then explained the vehicle purpose as a navigation tool through inhospitable environments. Following that, on the top left

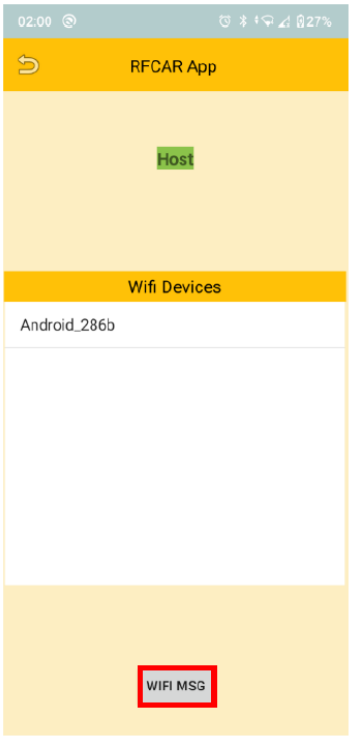

Figure 7.17.: Wi-Fi message exchange tests

corner, there's a button that when pressed it redirects to the main app activity allowing first-person vision on the controlled car just like a simulated car game, only this time in a real-life scenario. Some statistics, such as the transport position and its velocity, appear in front of full-screen video capture. From that point on, one can also choose to see the map and possibly the route traced along with some more detailed statistics of the remote control car. This first UI idea is depicted in figure 7.18. Later, some UI tests were made in terms of app functionality. One had to make sure the application behaved properly without crashing when, for example, the user intends to rotate the screen. Some activities' (app screens) orientation was fixed to ensure correct behaviour. The ones where that wasn't an issue, a version for landscape was created and tested (figure 7.19).

7.2. Integrated testing

Having tested every subsystem, comes the time to test the whole system as one.
An initial test was made with the **HC-05 Bluetooth module** inserted on the STM board. That approach only allows system engineers who have access to that module to fully test both Bluetooth client and server. However, to prove this new compound functionality, the device to be paired should also have Bluetooth

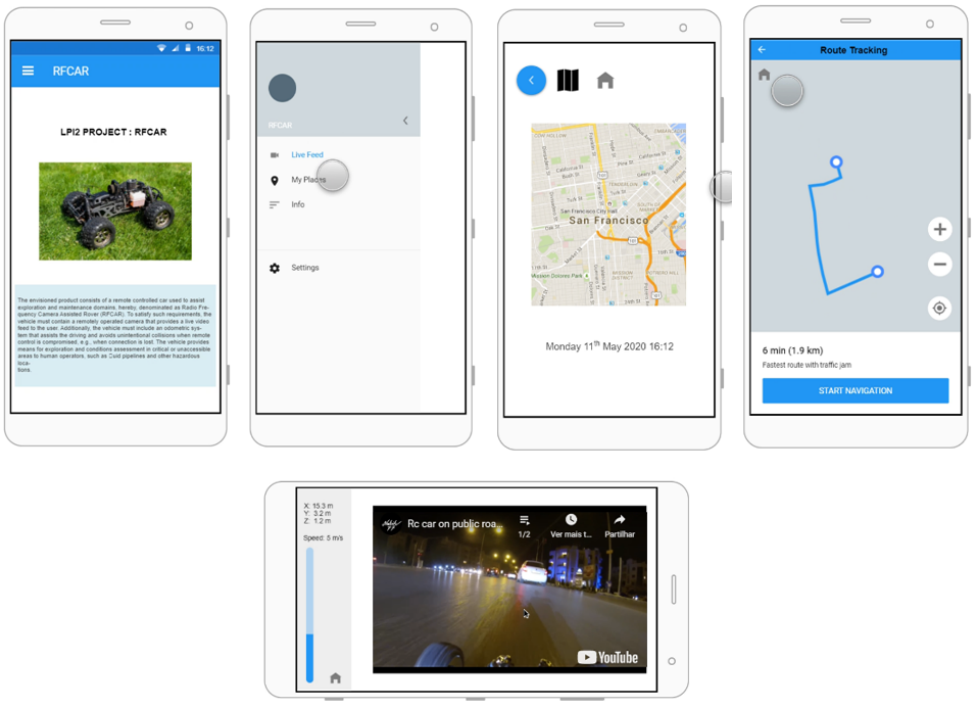

Figure 7.18.: First UI idea

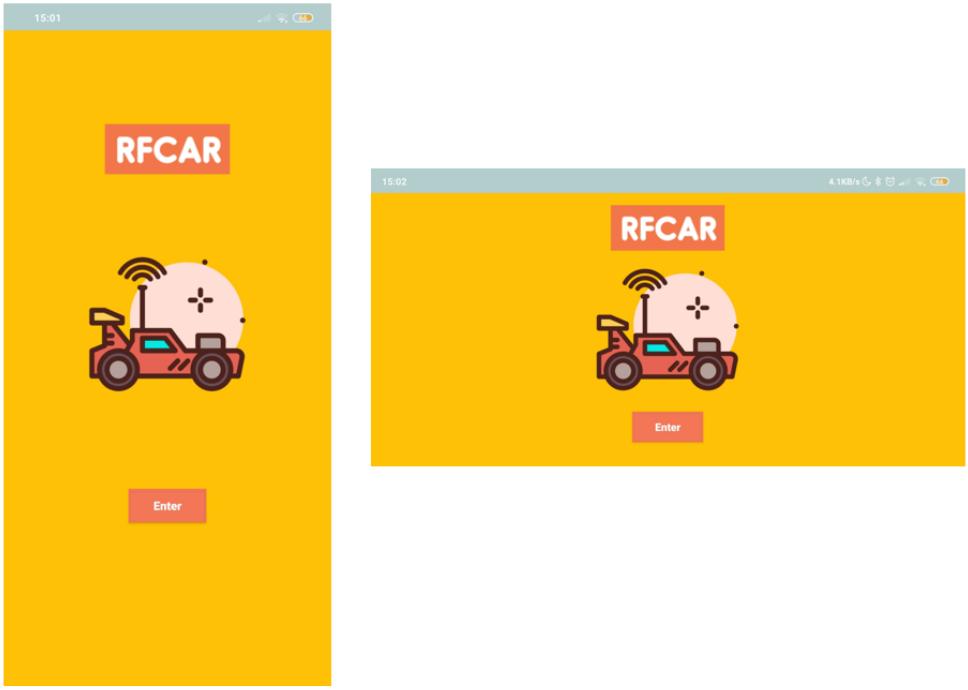

Figure 7.19.: UI orientation test example

drivers (setup) and specific hardware to use that protocol. However, despite both devices having the forementioned drivers and hardware for the connection to succeed, that didn't happen. After some research on the topic, one can assume that the problem was caused by **android version mismatching**. Meaning that the HC-05 only recognised older android versions (prior to 4.2). Fortunately, a new solution was found where one could run the server Bluetooth app in a virtual machine using, for example, COM6 port to communicate, and then redirect that port to another one (COM9) establishing the connection with the phone. In other words, the android app and bluetooth server app can be connected to different ports and still comunicate, due to the foresaid redirection represented in figure 7.20. As an advantage, this method requires less hardware resources.

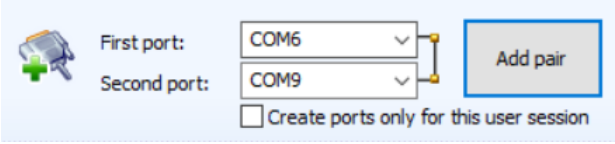

Figure 7.20.: Port redirection

After turning on Bluetooth on both android (ram) and PC devices (JOAO F), as depicted in figure 7.21, one could then open the android app and see in main menu its initial functionalities, figure 7.22. The app manually requests Bluetooth enabling if it isn't in the beginning.

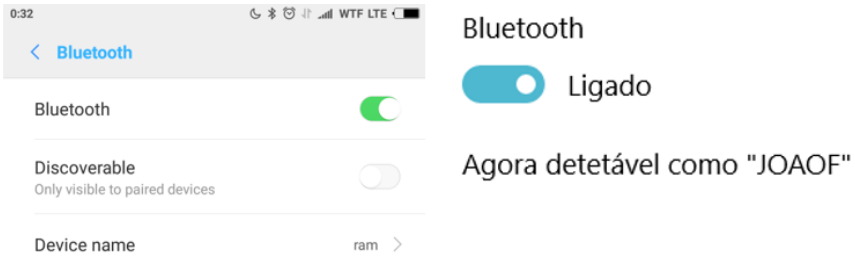

Figure 7.21.: Smartphone as "ram" and PC as "JOAOF" with Bluetooth enabled

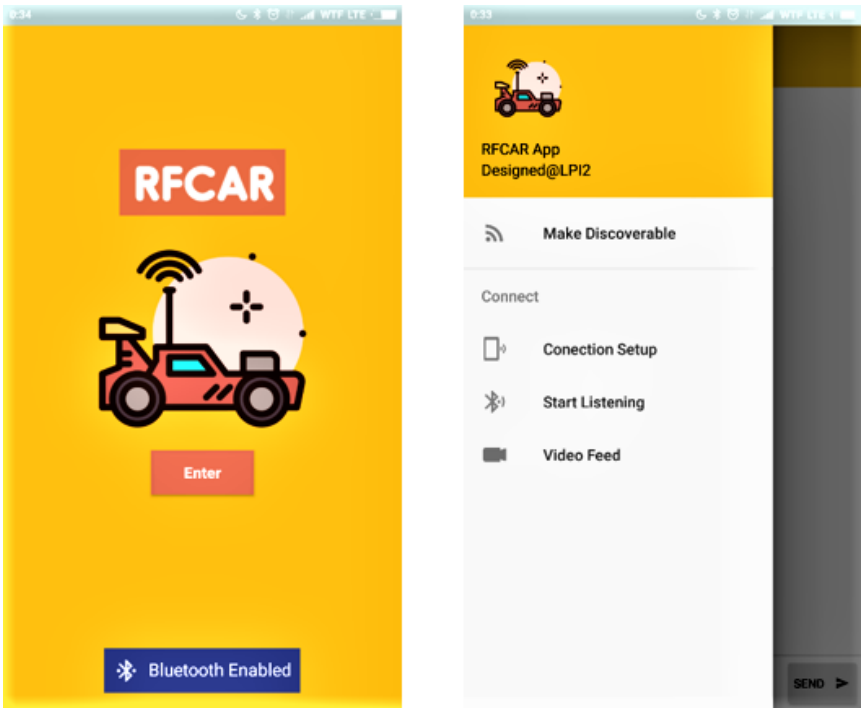

Figure 7.22.: RFCAR app: Initial and main activities, accordingly (**Test UI**)

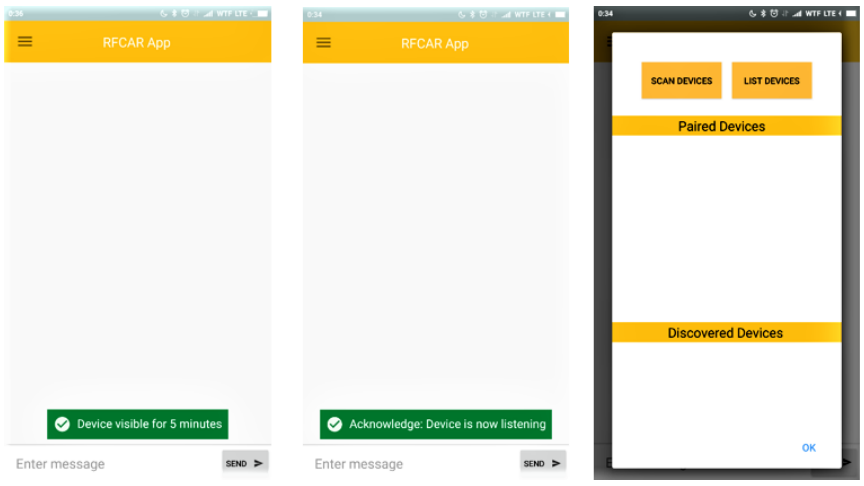

Figure 7.23.: RFCAR app: Discoverable, Start Listening and Connection Setup activities, respectively

To make an initial connection from square one, the smartphone should be discoverable and then be able to start listening, so that one could configure its connection setup. To do that, the options available in the main menu were clicked in that sequence, and when making it discoverable (even if only for 300 seconds) one should accept, figure 7.23.

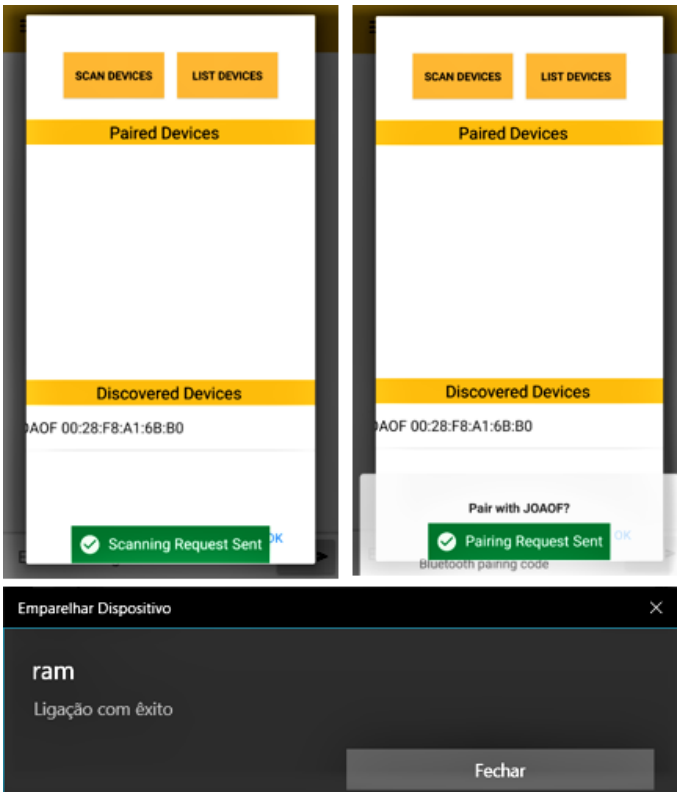

Figure 7.24.: RFCAR app and PC: Pressed Scan Devices button, target device pressed and paired successfull message on PC. A PIN number given by the PC must be inserted on app to conclude the pair phase

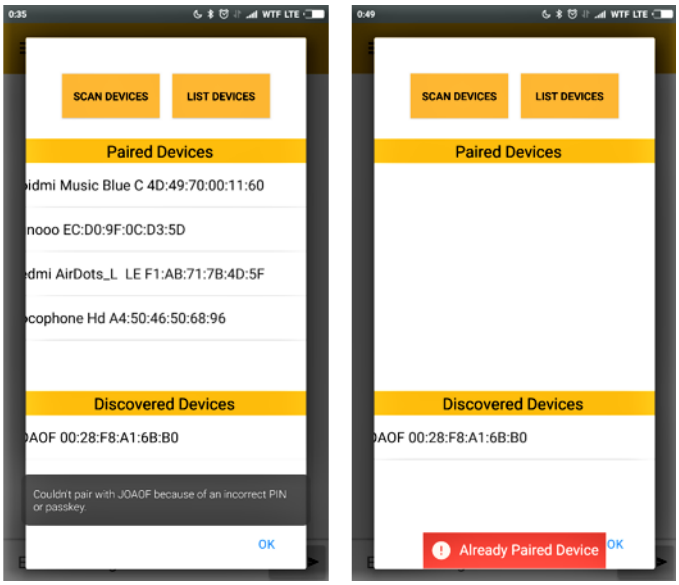

Figure 7.25.: RFCAR app: Pair failed examples: Wrong pin number or already paired device

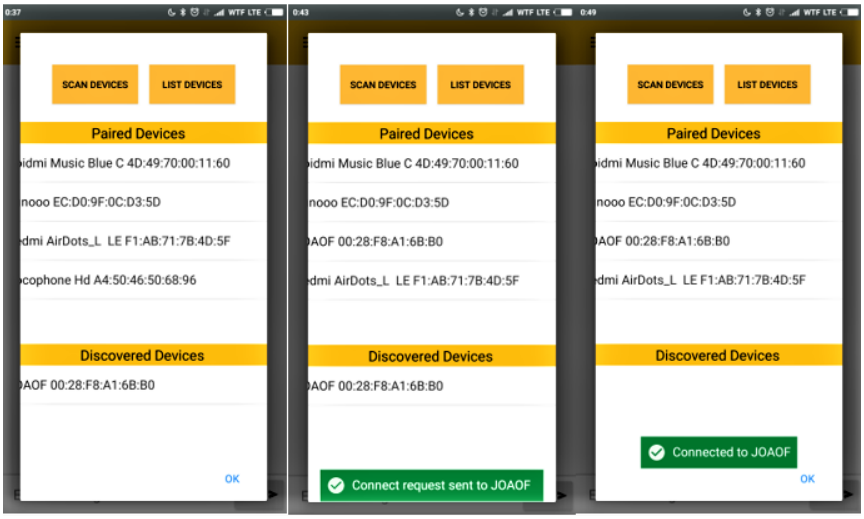

Figure 7.26.: RFCAR app: Connect phases: List devices pressed, target deviced pressed and successfull display of connection establish toast

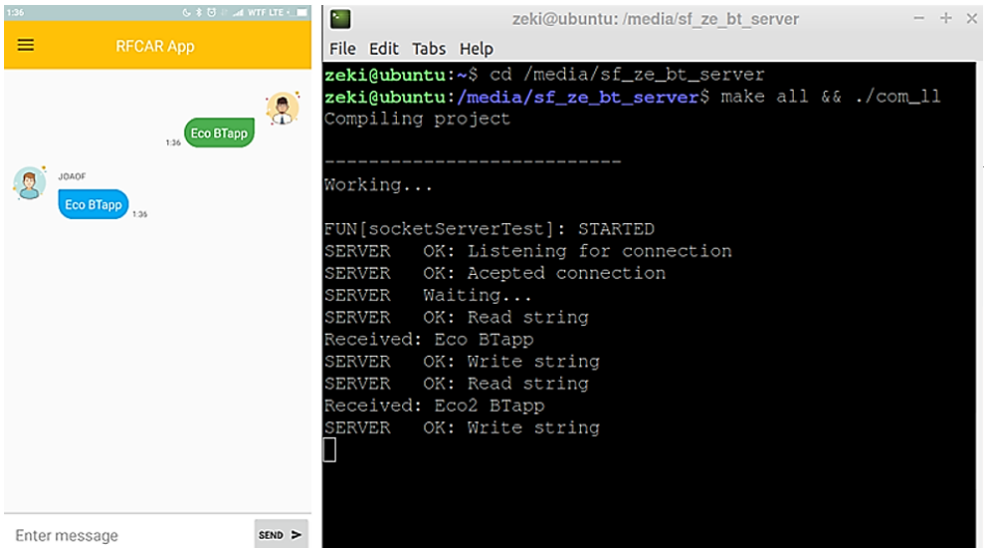

Figure 7.27.: RFCAR app and PC BT server: Successfull message exchanged from android to pc and android echo

When on the connection setup phase, the scan button was clicked to show the PC to connect as a discoverable device and following that, that same device detected was clicked to pair up with the smartphone. When the pairing phase is completed, one out of two outputs happen. The PC device pairs, as proven in last capture of figure 7.24, or doesn't as depicted in figure 7.25, as an example. Error messages would pop up whether the PIN introduced on the PC was mismatched or it was already paired in the first place, accordingly. On the former alternative, the connection was possible after pressing list devices button and clicking on the same target device as when pairing, sequence of events displayed in figure 7.26.

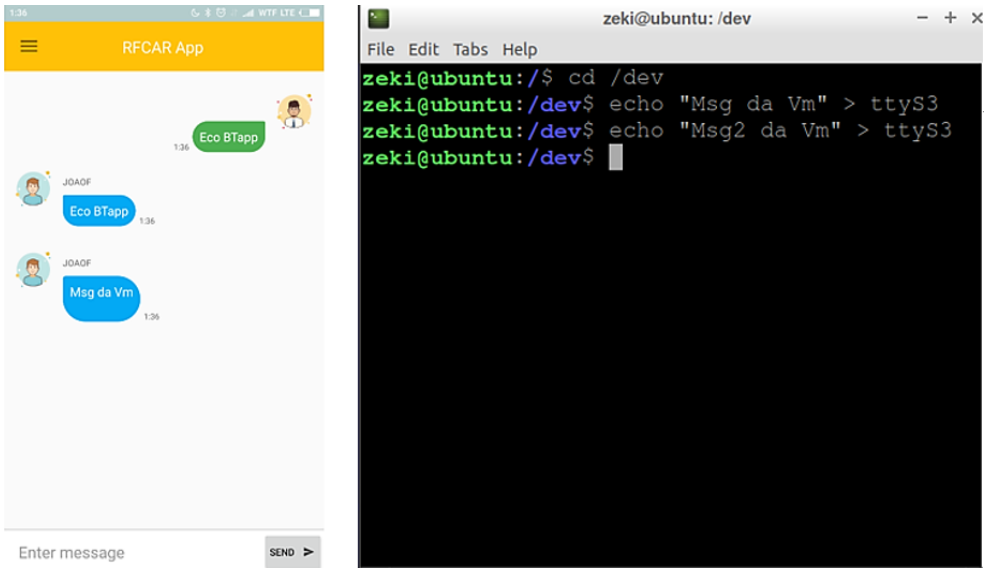

Figure 7.28.: RFCAR app and PC BT server: Successfull message exchanged from pc to android

Figure 7.29.: RFCAR app: It's possible to resend and receive more than once

Lastly, after opening output log terminals to receive app messages on the personal computer virtual machine guest operative system, on the principal menu, in the Enter message section, the messages were typed and then sent via clicking the adjacent send button. The message was then echoed on the phone and conveyed to the target (JOAO F) device, where they were displayed. Figure 7.27 show app and PC terminal points of view. Regarding figure 7.28, the message was written on the terminal and received on the smartphone, respectively. Figure 7.29 display final state on the app log after resending a different message as well as obtaining another one via the same source. Note that the PC terminal images already depicts all messages exchanged and show the initial command configuration needed for the test to happen.

On one hand, in the receiving terminal, the first command: cd /media/sf_ze_bt_server changed the directory to where the server app is located. The second: make all && ./com_ll compile all the source files and run com_ll.

On the other hand, in the sending terminal, the current directory was changed to /dev with the command: cd /dev to navigate to where virtual COM4 (ttyS3) so that messages could be sent through, as exemplified in the instruction: echo Msg da VM > ttyS3.

After properly testing the message exchange between the smartphone and the NVS one would simply need to start sending the control values (speed and wheel tilt percentages) periodically, receive the important alerts from the NVS and redirect them to the notification pop-ups created earlier. The results of these tests are represented in figure 7.30. Note that the video stream presented in figure 7.30 is merely used for test purposes as a way to envision a live rover feed.

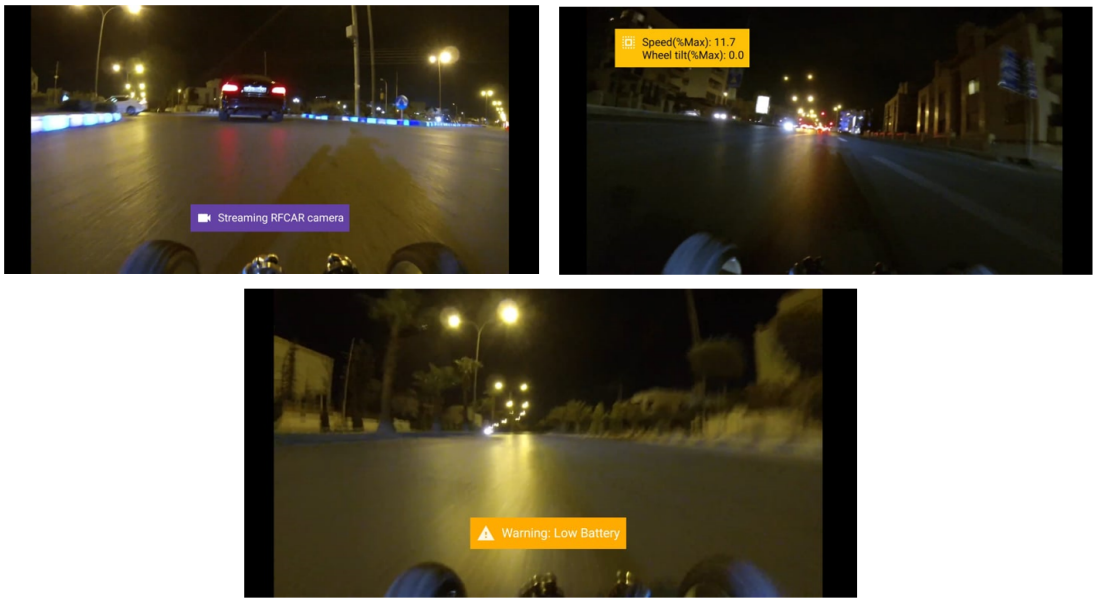

Figure 7.30.: Final product pop-up notification examples

7.3. Functionalities: Summary

The summary of the implemented functionalities is as follows:

- ☒ NVS [3/3]
  - ☒ Digital control of the vehicle: speed and direction
  - ☒ Obstacle avoidance
  - ☒ STM32 full-stack implementation
    - ☒ OS: threads, shared-memory, mutex.
    - ☒ IO: sensors, motors
    - ☒ Communication [2/2]:
      - ☒ Bluetooth: client and server
      - ☒ RS232: client and server
- ☒ Smartphone [3/3]
  - ☒ User Interface
    - ☒ Retrieving smartphone sensors data for simplified vehicle navigation (accelerometer/rotation sensor)
    - ☒ Video Feed
    - ☒ Notifications
  - ☒ Bluetooth: client and server
  - ☒ Wi-Fi: client and server
- ⊟ RVVS [2/3]
  - ☒ Communication [2/2]:
    - ☒ Wi-Fi: client and server
    - ☒ RS232: client and server
  - ⊟ Image Acquisition [1/2]
    - ☒ Image capture from webcam
    - ☐ Video capture and streaming
  - ☒ Telemetry

8. Verification and Validation

After testing the behaviour of the devised solution, its performance should now be tested and analysed in respect to the foreseen product specifications. Additionally, the product should be validated by an external agent to the development team to assess its overall suitability to its intended purpose. In this chapter, the verification and validation tests performed are presented and analysed.

8.1. Verification

In the verification phase, the product's performance is tested and verified its compliance to the foreseen specifications.

8.1.1. Correctness of the control algorithms

After the implementation of the control algorithms, the results obtained were compared with the output of the simulations.

In most cases, the output of the implementation and simulation are very similar, having only changed the stabilization time due to the use of the modified velocity algorithm. Even though in the implementation the car reaches the speed reference faster than in the simulations, the behavior of the car position wise is very similar, and both velocity graphics have the same wave form.

It was also possible to visualize that when the reference angle is not 0, in the implementation, the velocity of both wheels has a very small ripple at the same time. which did not occur in the simulations. These results are prensent in the test section 7.1.1.6.

In general, the implementation results were the expected, since the differences between what was obtained and what was simulated were very not significative.

8.1.2. Image Acquisition

In this section is verified the compliance between foreseen (see Section 3.1.5) and actual specifications (see Section 7.1.2.1).

As can be seen in this latter section, unfortunately, the available webcam is very old, supporting only the **uyvy422** format (luminance + chrominance). This prevents the straightforward video capture and its streaming to a remote streaming platform (e.g. **Youtube**) where it's ubiquitously available through a shareable link. A more convenient format would be the MJPEG.

Concerning the framerate, it can be that the webcam supports 30 fps only. However, it's actual value in dependent on system's load, but, as video capture was not possible, consequently, it is left undetermined.

For the same reason, and although several resolutions are available (640x320, 352x288, 340x240, 176x144, and 160x120), it was only possible to test out image acquisition at these resolutions, with success, but lacking the video capture tests, which is not critical, as it does not depend on dynamic conditions, only on a statically defined capability of the webcam.

8.1.3. Functionality

The end goal in this stage was to create a main system composed of subsystems that interacted with protocols like Bluetooth, Wi-Fi, GPRS and RS232 to assure the speed reference and wheel tilt commands reached the (virtual) rover. Succeeding, the rover should move accordingly to those commands altering its virtual coordinates. After the phase of integrated testing, Section 7.2, one can affirm that this goal was half met since the commands reach the virtual vehicle but only through one communication source (Bluetooth). The latter can also communicate back sending the alerts related to the status of the rover, once again resorting to Bluetooth. The final virtual environment block diagram is depicted in Fig. 8.1.

8.1.4. Communication

Considering subsystem communication the two parameters to verify were Reliability and Redundancy, since the Communication Range was reliant on a physical prototype test.

8.1.4.1. Reliability

Concerning Reliability, one needed to implement a system where the Wi-Fi dropped packets vs total packets ratio was monitored. However, since the system lacks the control message exchange through Wi-fi this goal is impossible to fulfil. Despite that, the Bluetooth control message exchange was implemented and

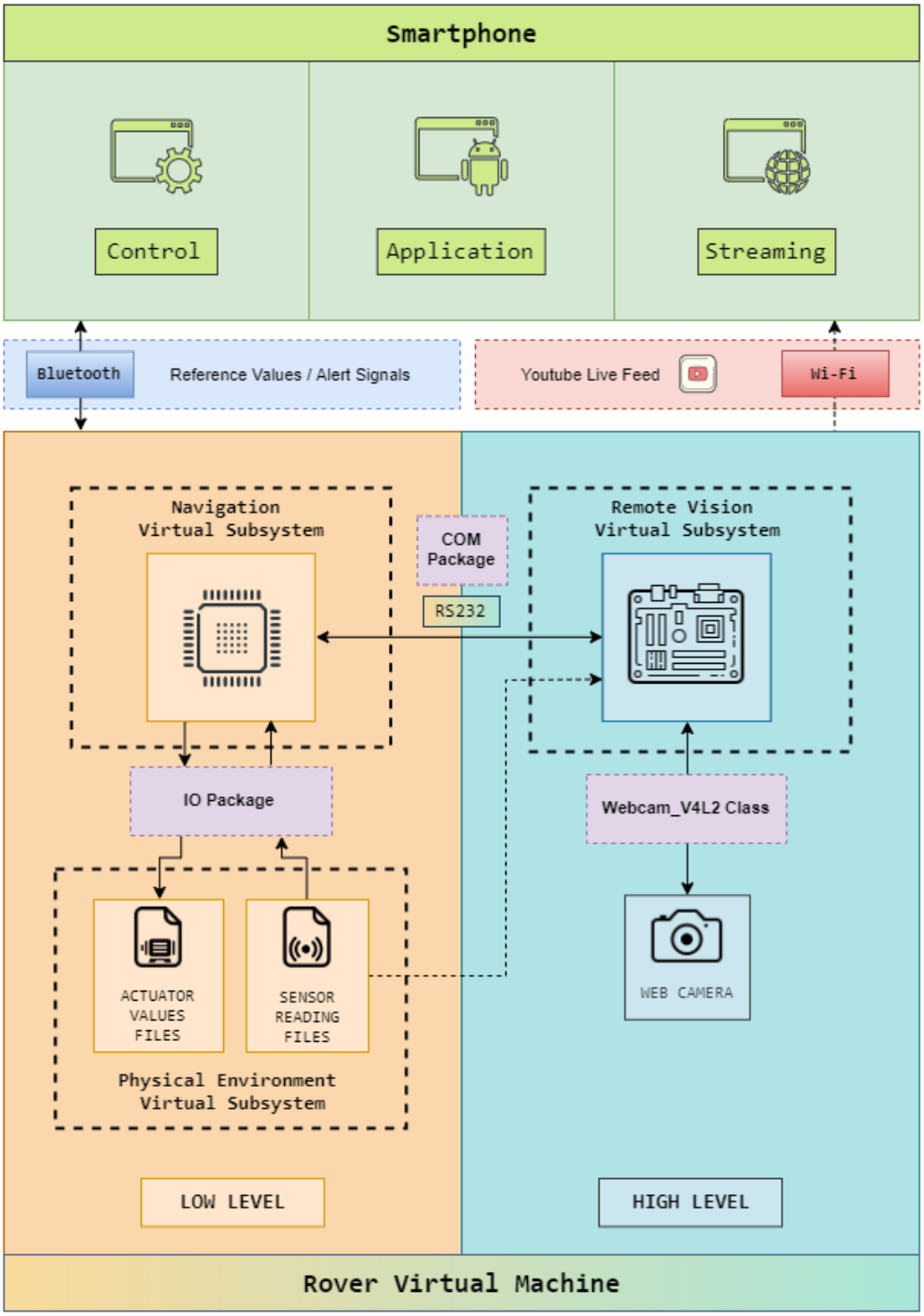

Figure 8.1.: Final virtual environment block diagram view

tested.As these tests were made the package loss was analysed and one can conclude there is little to no package loss when sending the commands from the smartphone to the NVS over Bluetooth since no message was lost. Despite not using a Wi-Fi, the Bluetooth protocol was still a reliable option.

8.1.4.2. Redundancy

Regarding redundancy, in the initial design (Fig. 4.1) it was established the Wi-Fi as the main communication and the Bluetooth as redundancy communication. Later, the main communication was changed to Bluetooth and, as aforementioned, there is no redundancy communication for sending commands to rover when the latter fails.

8.2. Validation

In this stage, the product is tested by an external agent to the development team to assess its overall suitability to its intended purpose. For that reason, the app was tested by a user outside of the workgroup. Instructions were given to him so that he could understand the concept to navigate through the user-friendly app and report any bugs or challenges faced when performing the following steps:

Step 1) Introduce the user to the last version of the android application: RFCAR app, after instructions given by one of the developers.

Step 2) When prompted by a system message pop up telling to turn on Bluetooth, turn on Bluetooth.

Step 3) Press Enter Button. Now, on the main menu, where a idle message, youtube icon and an arcade controller appears, press the button on the top left corner.

Step 4) Press the Bluetooth Discover button to allow the smartphone to be visible to other devices. Press Allow when prompted by the system message.

Step 5) Press the Connection Setup button on the lateral sliding tab.

Step 6) Press the Scan Devices button and wait until the Bluetooth enabled target device appears on Discovered Devices subsection and press it. If the target device to pair with doesn't appear, turn on its Bluetooth. A message to conclude the pair phase should appear on both device with a corresponding PIN. Certificate that the PINs match and then click allow and yes on both devices to conclude this stage.

Step 7) Press the List Devices button and wait again until the target device appears on Paired Devices subsection and press it to establish the connection.

Step 8) On the sliding tab, press Enable Wifi to turn it on. The device tries to automatically connect to a given Wifi network. If that happens, skip to step 10.

Figure 8.2.: The final version of the app is launched

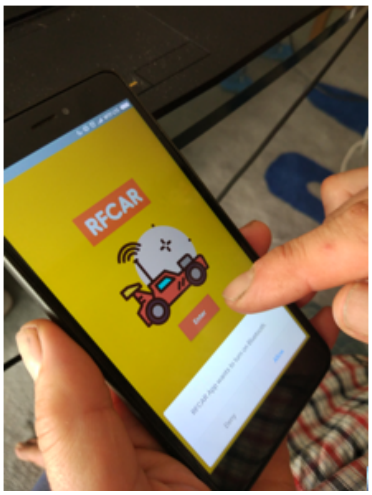

Figure 8.3.: Bluetooth is enabled, then Enter button is pressed

Step 9) Go to main menu and press the Wifi Discovery button and select the desired wireless network on the Wifi Devices section. Touch Wifi Msg to send an initial message to that network to initiate connection. Input the password if prompted.

Step 10) Touch the main menu. A full-screen youtube window and some accelerometer statistics emerges corresponding to speed and wheel tilt relative to its maximum values. Touch the play button and tilt the phone to see that the values are passing to the other Bluetooth device.

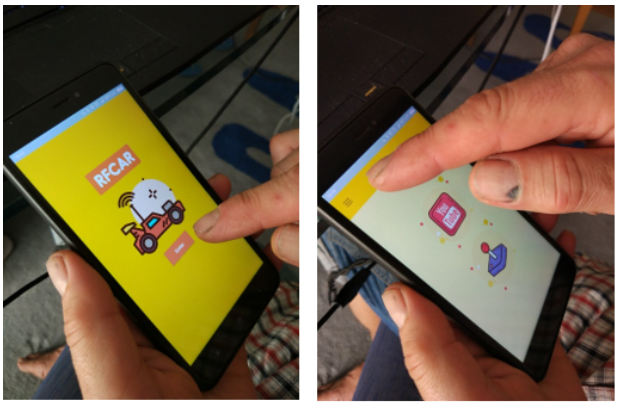

Figure 8.4.: On main menu, press the top left corner button to open options tab

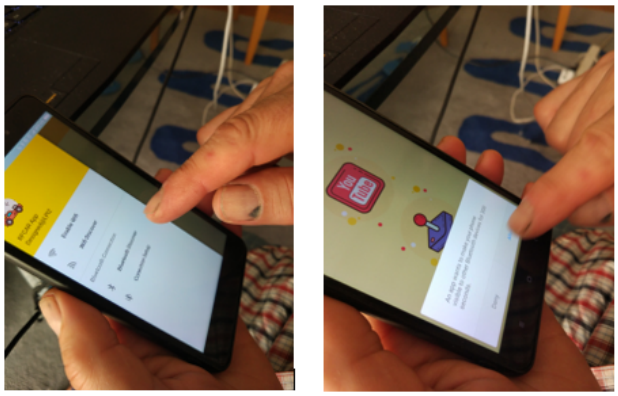

Figure 8.5.: Bluetooth Discover is pressed, turning the smartphone visible to other devices

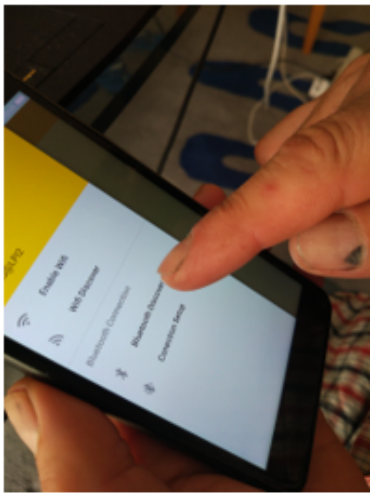

Figure 8.6.: Connection Setup is pressed to configure Bluetooth connection

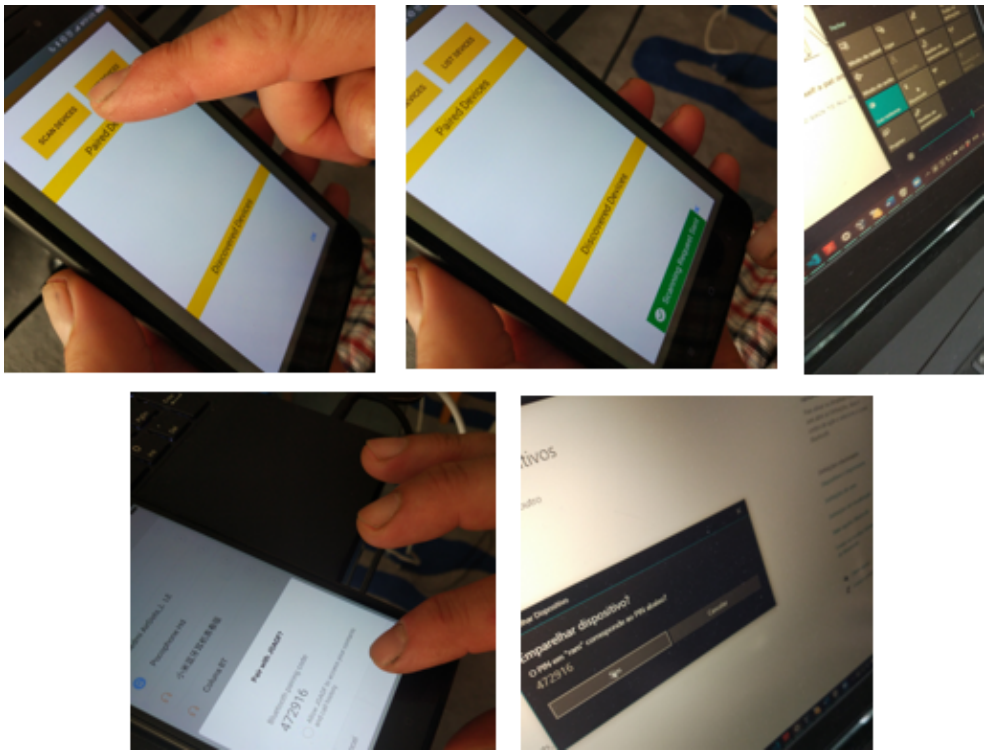

Figure 8.7.: Scan devices button is pressed, the desired target device to pair with is touched, and the pair phase concludes after the user verify that the PIN of the two devices matches and press yes on both messages displayed on each device

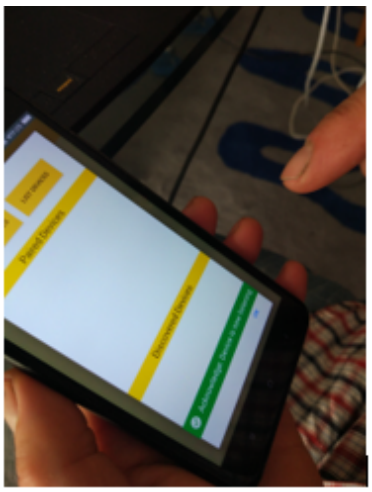

Figure 8.8.: A connect request is sent and shortly after accepted by the target device

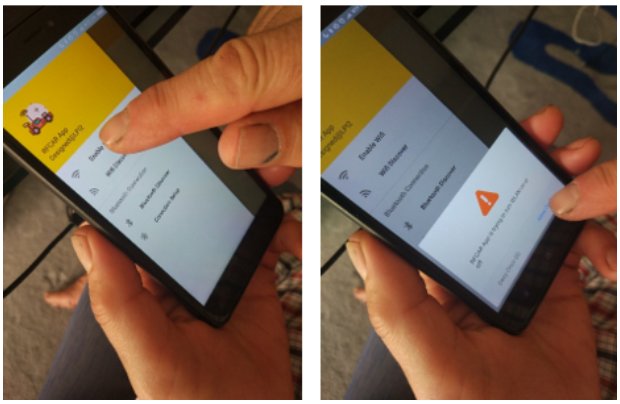

Figure 8.9.: Enable Wifi button is pressed

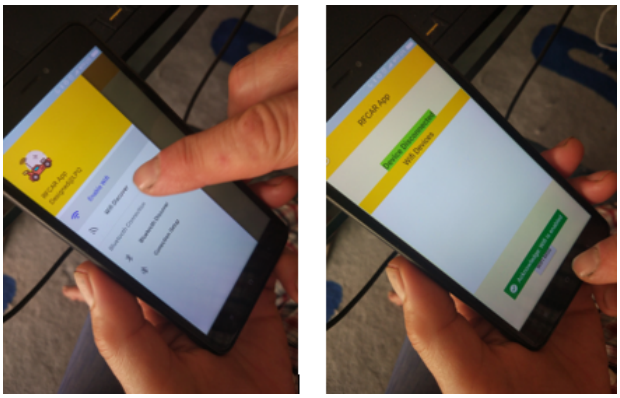

Figure 8.10.: The desired network is selected after pressing Wifi discover button

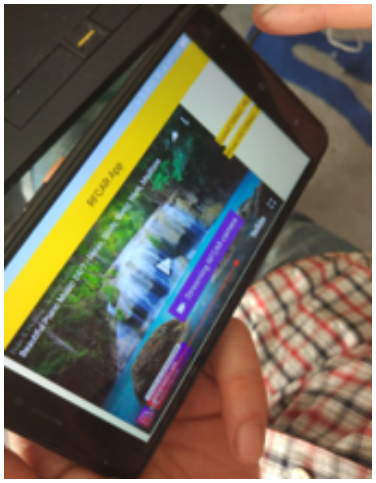

Figure 8.11.: The application sends accelerometer values to the desired device when phone tilts, and a video feed is enabled

9. Conclusion

In this chapter are outlined the conclusions and the prospect for future work regarding the RFCAR project.

9.1. Conclusions

The realization of this project went through several phases, following the waterfall methodology and top-down approach with two process iterations, modular and integrated.

In the analysis stage, the foreseen specifications were listed, as well as the envisioned tests for verification and validation of the product. Additionally, a preliminary design was sketched as a possible viable solution.

In the design phase, the considerations drawn in the analysis, combined with the initial design, were used to conceptualize a viable solution for the product materialization. The system as decomposed into smaller subsystems that was conducted by specialized teams, namely: Hardware control, smartphone, STM32, camera and Raspberry Pi.

The implementation phase started with a modular implementation of all the referred smaller subsystems and consecutive test. This phase always want hand to hand with the design since the functionalities didn't always work at first. After the modular implementation came te integrated implementation where all the smaller subsystem were combined in order to form the final product. Even though some tests were successful in the integrated implementation, others were not possible to accomplish.

After the implementation and respective tests, came the verification phase where the focus test the the foreseen specifications.

Finally, came the phase of product validation in which an external agent tested the product interface according to the instructions provided. The documentation was always a strong pillar in all the referred phases, and as such, was always updated every time any progress was made.

From a critical point of view, this project suffered a large restriction due the current situation the world is in, which hindered the interaction between the members of the group making it difficult to manufacture a prototype and to integrate the functionalities modularly implemented. As such, the base of the project was model-based design and simulation, which is a must-have tool for agile and cost-effective development of

products.

Despite the obstacles, this project can be considered a accomplishment due to the success of the android interface, multi platform communications, camera and control algorithm within a restricted timeline and budget, supported by the methodology used and the model-based design framework.

9.2. Prospect for Future Work

In the foreseeable future, an actual prototype could be implemented using the developed control algorithms, communications, camera and interface, as was the initial plan.

In order to make the system better, it could be used a different camera, with better frame rate, resolution and range and more image formats. Also, redundancy could be added to communication making the system more stable and less fault-prone.

To create a better environment for the user, some upgrades could be made such as, finish video feed interface, add Global Positioning System (GPS) tracking to assess the position of the car at all times, and more rigorous odometry that does not force the vehicle's stoppage when facing an obstacle, but allows it to circumvent it.

Finally, the prototype could be tested in different real environments, as envisioned.

Bibliography

[1] Ian Sommerville. Software process models. ACM computing surveys (CSUR), 28(1):269–271, 1996.

[2] Michael A Cusumano and Stanley A Smith. Beyond the waterfall: Software development at microsoft. 1995.

[3] Bernd Bruegge and Allen H Dutoit. Object-Oriented Software Engineering Using UML, Patterns and Java-(Required), volume 2004. Prentice Hall, 2004.

[4] Albert S Huang and Larry Rudolph. Bluetooth essentials for programmers. Cambridge University Press, 2007.

[5] E Bryan Carne. A professional's guide to data communication in a TCP/IP world. Artech House, 2004.

[6] Geoff Sanders, Lionel Thorens, Manfred Reisky, Oliver Rulik, and Stefan Deylitz. GPRS networks. Wiley Online Library, 2003.

[7] Gary R Wright and W Richard Stevens. Tcp/ip illustrated. vol. 2: The implementation. tii, 1995.

[8] Michael Kerrisk. The Linux programming interface: a Linux and UNIX system programming handbook. No Starch Press, 2010.

[9] M David Hanson. The client/server architecture. In Server Management, pages 17–28. Auerbach Publications, 2000.

[10] Client/server model. URL https://www.ibm.com/support/knowledgecenter/en/SSAL2T_8.1.0/com.ibm.cics.tx.doc/concepts/c_clnt_sevr_model.html. accessed: 2020-07-02.

[11] DICK AUTOR BUTTLAR, Bradford Nichols, Dick Buttlar, Jacqueline Farrell, and Jackie Farrell. Pthreads programming: A POSIX standard for better multiprocessing. " O'Reilly Media, Inc.", 1996.

[12] Yocto project. URL https://www.yoctoproject.org/. accessed: 2020-07-02.

[13] Video for linux 2 (v4l2) headers in linux kernel code. URL https://github.com/raspberrypi/linux/blob/rpi-4.4.y/include/uapi/linux/videodev2.h#L468. accessed: 2020-07-02.

[14] Motion sensors: Android developers. URL https://developer.android.com/guide/topics/sensors/sensors_motion.

[15] Googlearchive. googlearchive/android-bluetoothchat. URL https://github.com/googlearchive/android-BluetoothChat.

[16] Youtube android player api | google developers. URL https://developers.google.com/youtube/android/player.

[17] Webcam passthrough for virtualbox. URL https://docs.oracle.com/en/virtualization/virtualbox/6.0/admin/webcam-passthrough.html. accessed: 2020-07-02.

[18] Convertio: online image converter. URL https://convertio.co/pt/jpg-converter/. accessed: 2020-07-12.

Appendices

A. Project Planning — Gantt diagram

In Fig. A.1 is illustrated the Gantt chart for the project, containing the tasks' descriptions.

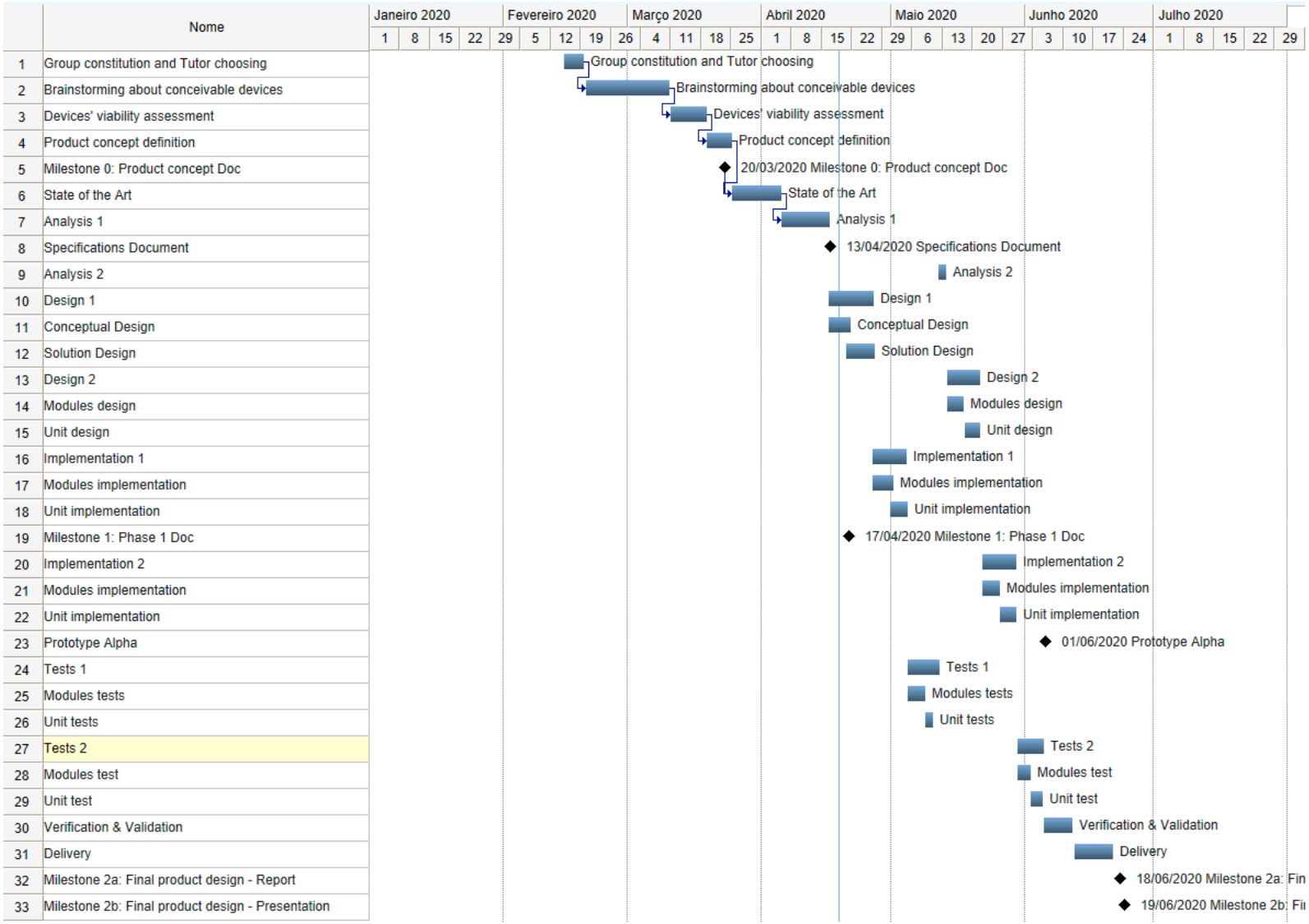

Figure A.1.: Project planning — Gantt diagram

B. Online Repository — GitHub

In Fig. B.1 is illustrated the online repository used to organize the project.
https://github.com/LPI2-DreamTeam/RFCAR

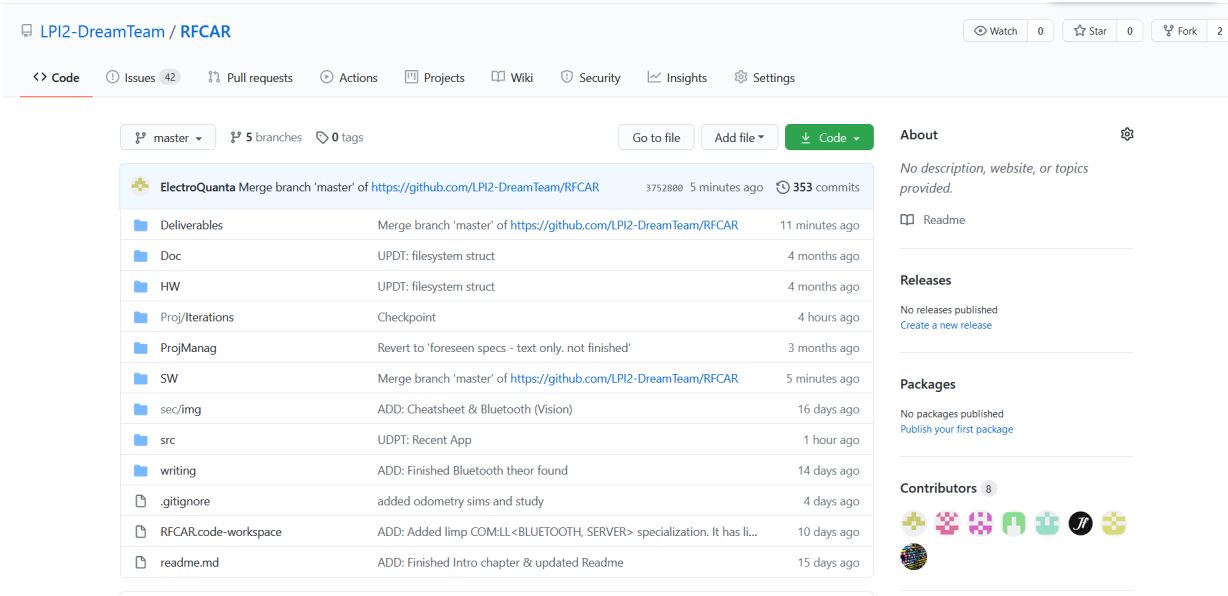

Figure B.1.: Online Repository — GitHub

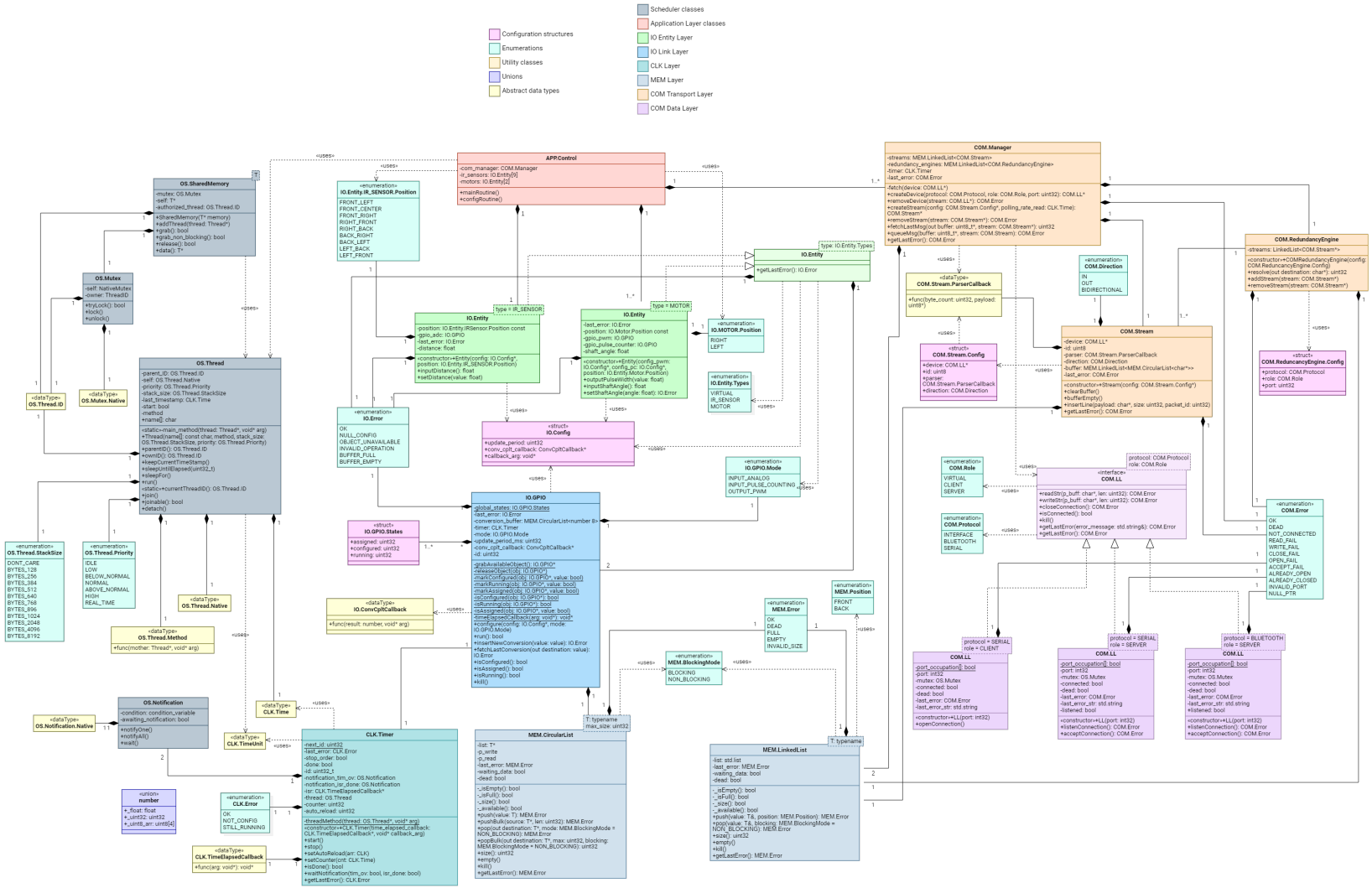

Figure B.1.: Navigation subsystem class diagram augmented

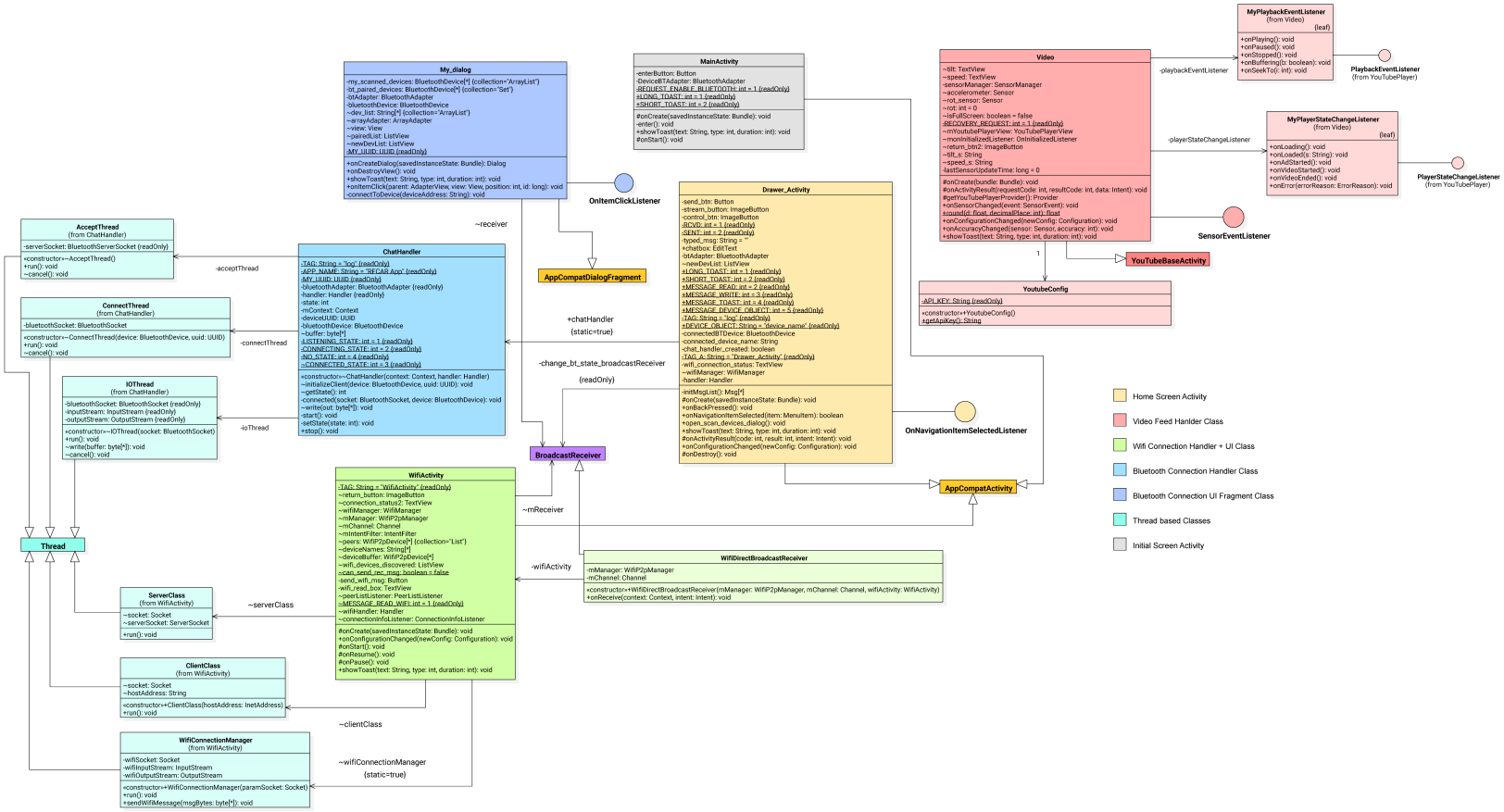

Figure B.1.: Smartphone class diagram augmented

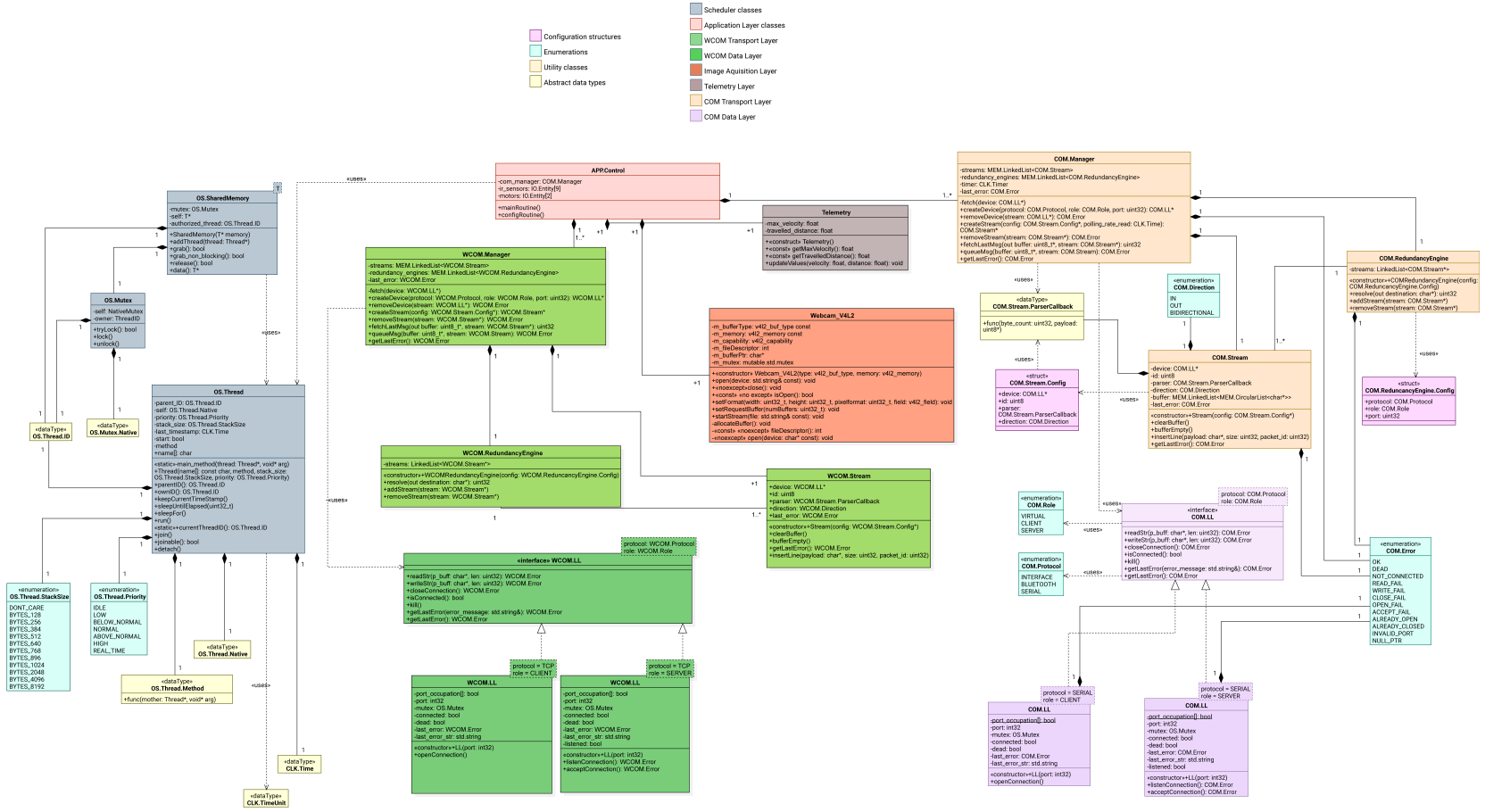

Figure B.1.: Vision Class Diagram augmented