

UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN**



**GRADO EN INGENIERÍA DE TECNOLOGÍAS Y
SERVICIOS DE TELECOMUNICACIÓN**

TRABAJO FIN DE GRADO

**DESIGN AND IMPLEMENTATION OF A MOTOR
CONTROL SYSTEM BASED ON micro-ROS**

MARÍA GARCÍA PEROTE

2023

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN



GRADO EN INGENIERÍA DE TECNOLOGÍAS Y
SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DESIGN AND IMPLEMENTATION OF A MOTOR
CONTROL SYSTEM BASED ON micro-ROS

Author

MARÍA GARCÍA PEROTE

Tutor

ALEJANDRO GÓMEZ MOLINA

Co-tutor

ÁLVARO GUTIÉRREZ MARTÍN

2023

Abstract

Control systems are an important part of our daily lives due to the large number of applications in which they are integrated, from automated vehicles to medical systems. They are part of more complex robotic systems and their main functionality is to regulate the behaviour of other devices to achieve a certain result.

These systems must meet stringent constraints and requirements while trying to reduce the likelihood of error. However, the communication between the principal functional blocks of these systems presents a challenge, which results in a long development time even for experts.

Therefore, the aim pursued by the current BSc Thesis is to develop a standard software platform based on ROS architecture, which will be scalable and will facilitate the design process of control systems for developers.

For this purpose, a motor control system has been developed and implemented and a simple functional application has been integrated to work on it. Thanks to PID controllers and the use of micro-ROS, this will allow the user to control and drive the motors to a desired position.

Keywords: embedded systems, middleware, microcontrollers, motor, controller, ROS

Resumen

Los sistemas de control constituyen una parte importante de nuestro día a día debido a la gran cantidad de aplicaciones en las que aparecen integrados, desde vehículos automáticos hasta sistemas médicos. Forman parte de sistemas robóticos más complejos y su principal funcionalidad es la de regular el comportamiento de otros dispositivos para conseguir un cierto resultado.

Estos sistemas deben de cumplir con estrictas restricciones y requerimientos a la vez que tratar de reducir las probabilidades de que se produzca algún error. Sin embargo, la comunicación entre los bloques funcionales principales de estos sistemas supone todo un desafío, lo que se traduce en un largo tiempo de desarrollo incluso para los más expertos.

Por tanto, el objetivo que persigue el actual Trabajo de Fin de Grado es desarrollar una plataforma estándar de software basada en la arquitectura ROS, que sea escalable y facilite a los desarrolladores el proceso de diseño de los sistemas de control.

Para ello, se ha desarrollado e implementado un sistema de control de motores y se ha integrado una sencilla aplicación funcional que trabaje sobre este. Gracias a controladores PID y al uso de micro-ROS, esto permitirá al usuario controlar y dirigir los motores a la posición deseada.

Palabras clave: sistemas embebidos, middleware, microcontroladores, motor, controlador, ROS

Acknowledgements

I would like to thank my supervisor, Alejandro Gómez, for all the dedication and support that he has offered me during the development of this thesis. Thank you for your time, enthusiasm, advice and patience, and for introducing me to a field of engineering in which I will undoubtedly continue to deepen. None of this would have been possible without you. I would also like to thank Álvaro Gutiérrez for the opportunity he gave me to develop a project that I would like and that would be a challenge for me, so that I could feel satisfied putting an end to my career.

There are many people who have accompanied me on this journey and their support and affection have been essential for me to get this far. Since I arrived in Madrid I have met wonderful people who have become more like family than friends, and I would like to thank them for their affection and words of encouragement that have helped me so much. I would also like to thank my life partner for all the love, support and trust he has given me, for encouraging me and giving me the strength I needed to fight for everything I want to achieve and which makes me happy.

Finally, I would like to thank my family for being the fundamental pillar of my life, but especially my parents and my brother, for their unconditional love and support and for believing in me even when I had stopped to believe. I get emotional just thinking about that 12-year-old girl who wanted to be a telecommunications engineer has finally fulfilled her dream.

Grandpa, I did it. I dedicate this to you.

Contents

Abstract	v
Resumen	vii
Acknowledgements	ix
Contents	xi
Figure Index	xiii
Table Index	xv
List of acronyms	xvii
1 Introduction and goals	1
1.1 State of the art	1
1.1.1 Embedded systems	1
1.1.2 Microcontroller	2
1.1.2.1 Peripherals	3
1.1.2.1.1 Timers	3
1.1.2.1.2 PWM	4
1.1.2.1.3 Rotary encoder	5
1.1.2.1.4 Ethernet	6
1.1.3 H-bridge	7
1.1.4 Controllers	8
1.1.5 ROS	9
1.1.5.1 micro-ROS	10
1.2 Motivations and goals	11
1.3 Document structure	12
2 Hardware architecture	13
2.1 Introduction	13
2.2 Requirements	13
2.3 Hardware selection	14
2.3.1 Microcontroller	14
2.3.2 Motor control board	15
2.3.2.1 H-bridge	17
2.3.3 Motor	17

2.4	Complete hardware system diagram	18
3	Software	21
3.1	Introduction	21
3.2	Requirements	21
3.3	Middleware	22
3.3.1	Timers	22
3.3.2	PWM	23
3.3.3	Encoders	25
3.3.4	Controller	26
3.3.5	Third party middlewares	28
3.3.5.1	FreeRTOS	28
3.3.5.2	LwIP	29
3.4	ROS2	29
3.4.1	micro-ROS	29
4	Testing	31
4.1	PWM middleware	31
4.2	Encoder middleware	32
4.3	PID middleware	32
4.4	LwIP middleware	33
4.5	ROS middleware	33
4.6	Functional application	34
5	Conclusions	37
5.1	Conclusions	37
5.2	Future improvements	38
	Bibliography	39
A	Ethical, social, economic and environmental aspects	43
A.1	Introduction	43
A.2	Description of relevant impacts related to the project	43
A.2.1	Ethical impact	43
A.2.2	Social impact	43
A.2.3	Economic impact	44
A.2.4	Environmental impact	44
B	Project budget	45
C	User manual	47
C.1	Requirements	47
C.2	Install dependencies	47
C.3	Starting XRCE-DDS agent	48
C.4	Compiling the project and running the application in Python	49

List of Figures

1.1	Basic structure of an embedded system [4].	2
1.2	Different examples of duty cycles - 50%, 20% and 80% [8].	5
1.3	Directions of a quadrature encoder [8]	6
1.4	Switching characteristic definition [13].	7
1.5	Structure of a control system [14].	8
1.6	The ROS ecosystem [17].	9
1.7	Communication between nodes in ROS [18].	10
1.8	Comparison between ROS2 and micro-ROS architecture [21].	11
2.1	Conceptual diagram of hardware requirements.	14
2.2	NUCLEO STM32F746ZG development board [24].	15
2.3	X-NUCLEO-IHM04A1 expansion board [25].	16
2.4	Connection of two bidirectional DC motors [25].	16
2.5	Motor selected (a) with its encoder (b) [30].	18
2.6	Complete hardware system diagram.	19
3.1	Timer callback representation [38].	23
3.2	PID feedback control scheme [40].	26
3.3	PID controller block diagram [42].	27
4.1	PWM test on the oscilloscope.	31
4.2	Encoder test on the oscilloscope.	32
4.3	Agent connection log.	34
4.4	Application topic list.	34
4.5	Platform for functional application.	35
4.6	Python application diagram.	35
4.7	Application workflow.	36
4.8	Data received from the application.	36
C.1	Capture of the IP configuration that needs to be modified.	48
C.2	Step 1: Open the downloaded project.	50
C.3	Step 2: Import the project.	50
C.4	Step 3: Change in debugger configuration.	51
C.5	Step 4: Compiling and debugging options.	51
C.6	Capture of the <i>rgt</i> console.	52

Table Index

1.1	Features of each timer category [8].	4
2.1	Truth table of an H-bridge [26].	17
2.2	Hardware connections.	19
3.1	Communication topics.	30
B.1	Human resources costs.	45
B.2	Costs of materials.	46
B.3	Total cost.	46

List of Acronyms

API:	Application Programming Interface.
A-D:	Analog-to-Digital.
ARP:	Address Resolution Protocol.
ARR:	AutoReload Register.
CCRx:	Capture/Compare Register.
CPR:	Counts Per Revolution.
CPU:	Central Processing Unit.
CSMA/CD:	Carrier Sense Multiple Access with Collision Detection.
CPR:	Counts Per Revolution.
DC:	Direct Current.
DHCP:	Dynamic Host Configuration Protocol.
DDS:	Data Distribution Service.
DMA:	Direct Memory Access.
DMOS:	Double-Diffused MOSFET.
DNS:	Domain Name System.
D-A:	Digital-to-Analog.
EEPROM:	Electrically Erasable Programmable ROM.
FPGA:	Field Programmable Gate Array.
HAL:	Hardware Abstraction Layer.
HSE:	High Speed External.
HSI:	High Speed Internal.
HW:	Hardware.
IC:	Integrated Circuit.

ICMP:	Internet Control Message Protocol.
IEEE:	Institute of Electrical and Electronics Engineers.
LAN:	Local Area Network.
LED:	Light Emitting Diode.
LwIP:	Lightweight Internet Protocol.
MAC:	Media Access Control.
MAN:	Metropolitan Area Network.
MCU:	Microcontroller Unit.
MII:	Media Independent Interface.
MOSFET:	Metal Oxide Semiconductor Field-Effect Transistors.
P2P:	Peer-to-Peer.
PHY:	Physical.
PID:	Proportional Integral Derivative.
POSIX:	Portable Operating System Interface.
PWM:	Pulse-Width Modulation.
PSC:	Prescaler.
RAM:	Random Access Memory.
RLC:	ROS Client Library.
ROM:	Read-Only Memory.
RMII:	Reduced Media Independent Interface.
RMW:	ROS Middleware Interface.
ROS:	Robot Operating System.
RTOS:	Real-Time Operating Systems.
TCP/IP:	Transmission Control Protocol/Internet Protocol.
TCM:	Tightly-Coupled Memory.
UDP:	User Datagram Protocol.
UART:	Universal Asynchronous Receiver-Transmitter.
UDP:	User Datagram Protocol.
UTP:	Twisted Pair.
XRCE-DDS:	eXtremely Resource Constrained Environment Data Distribution System.

Chapter 1

Introduction and goals

Control systems are an important part of our daily life and they are integrated into more complex robotic systems. The environment is continually changing and, in most cases, is unpredictable. This makes it challenging to predict the timing, location and nature of disturbances due to the infinite number of possible variations. Therefore, creating systems capable of handling all these changes is currently unachievable, but the development of control systems that can autonomously adjust their conduct to external changes is within reach. Their main functionality consists on addressing or regulating the behaviour of other devices to achieve a desired result, which entails meeting strict requirements while reducing the likelihood of error [1].

Their relevance lies in the large number of applications in which they appear. They belong, not only to automated vehicles, humanoids robots or airplane controllers, but also to biomedical systems in charge of controlling respiratory or cardiovascular variables, among others. This means that the performance and efficiency of control systems is of paramount importance as they contribute to the well-being and safety of people.

The communication between the principal functional blocks in these systems is something difficult to manage and it requires a long development time nowadays. Thanks to the emerging software tools, such as ROS (Robot Operating System), which is an open-source framework that facilitates the integration of complex robot systems, it is possible to offer a standard software platform to developers, easing this design process. This is the main objective pursued by the current final degree project.

1.1. State of the art

1.1.1. Embedded systems

Embedded systems are computer systems based on microprocessors or microcontrollers that are designed to execute a dedicated function, frequently with real-time computing constraints, which means that a delay meeting deadlines might lead to a complete system failure. Moreover, they are required to operate autonomously for extended

periods of time, so they are designed to be efficient, reliable and cost-effective [2].

These systems can operate as standalone devices but they are usually part of high-level systems, and they are composed of both hardware and software components, closely related to each other. Embedded systems combined with the stringent timing requirements [3] mentioned can boost a wide range of services, from regulating traffic lights or monitoring plane takeoffs to regulating our blood pressure.

The basic structure that composes an embedded system is made up of different parts as it is shown in the block diagram of Figure 1.1.

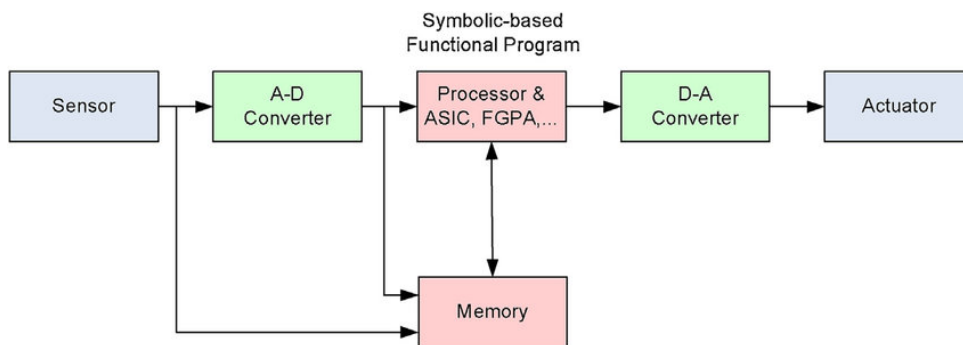


Figure 1.1: Basic structure of an embedded system [4].

- **Sensor:** it converts the physical magnitude to an electrical signal and stores the measured quantity to the memory.
- **A-D converter:** A-D stands for Analog-to-Digital. It converts the analog signal sent by the sensor into a digital one.
- **Processor:** data is processed by this part to determine the output and store it in memory.
- **D-A converter:** it stands for Digital-to-Analog and transforms the digital data given to the processor from digital to analog.
- **Actuator:** this component is in charge of comparing the output received with the expected one.

1.1.2. Microcontroller

A microcontroller (also called MCU or Microcontroller Unit) is an integrated circuit (IC) which constitutes a very small computer, also known as a chip, designed to perform specific tasks, commonly, in embedded systems. It is used to control the behaviour of the system, collect data from sensors and provide output signals to actuators based on the information gathered [5].

Microcontrollers are driven by synchronous sequential logic circuits and their states

change only at discrete timing regulated by a clock signal. Furthermore, its speed is determined by its clock frequency, which represents the number of cycles the microcontroller can complete in one unit of time, usually measured in megahertz (MHz) or gigahertz (GHz). This is directly related to the performance of the microcontroller and how fast it can execute instructions and perform tasks efficiently and reliably.

It is also important to mention that microcontrollers contain thousands of transistors, which are electronic components used to control and amplify electrical signals, on a single chip. This has been achieved thanks to the development of integrated circuit technology and as Moore's law states, the number of transistors of a microprocessor continues to double in every 18 months [6].

The main parts of a microprocessor are the following: a processor, a memory (RAM, ROM or EEPROM), an internal oscillator and peripherals of different types, from input/output ports whose main use is to interface with the external world, to timers, pulse width modulation (PWM) nodes or serial communication interfaces such as UART. Regarding the processor, it is a CPU (Central Processing Unit) which controls all the processes taking place in the MCU and executes instructions stored in memory, that can be either ROM (Read-Only Memory) or EEPROM (Electrically Erasable Programmable Read-Only Memory). The internal oscillator functions as the MCU's core clock and its purpose is to monitor the internal processes [7]. We will discuss peripherals in more depth in the next subsection of this project.

1.1.2.1. Peripherals

Microcontroller peripherals are hardware modules that are embedded into the microcontroller and offer an interface with the external world. Peripherals can be used to carry out specific tasks, such as generating analog signals, interacting with external devices or capturing input signals. The availability and flexibility of peripherals can be a key factor in choosing a microcontroller for a particular application.

The most important peripherals for this project are described hereafter:

1.1.2.1.1. Timers

Timers are hardware components which behave as *free-running* counters that work with a counting frequency that is a fraction of their source clock [8]. They are typically included as part of a microcontroller's on-chip peripherals and they are used for a wide range of purposes, such as measuring the frequency of external events, generating waveforms on their outputs or implementing real-time systems with accurate timing requirements.

To explain the timer architecture, it is important to take into account that an STM32 microcontroller shall be used in this project. These timers include interconnection

features, thereby enabling several of them to be combined and synchronized. There are different types of them: basic timers (which feature 16-bit counters), general purpose timers (16/32-bit counters), advanced timers, low power timers, and so on. Timers mainly differ in the number of inputs and outputs they have, in their resolution and in some functional features (some of them can be programmed in up/down counting mode or in direct memory access (DMA) mode). The most relevant features of each timer category are depicted below in Table 1.1:

Table 1.1: Features of each timer category [8].

Timer Type	Counter resolution	Counter type	DMA	Channels	Complimentary channels	Synchronization	
						Master	Slave
Advanced	16-bit	up, down and center aligned	Yes	4	3	Yes	Yes
General purpose	16/32-bit	up, down and center aligned	Yes	4	0	Yes	Yes
Basic	16-bit	up	Yes	0	0	Yes	No
1-channel	16-bit	up	No	1	0	Yes (OC signal)	No
2-channels	16-bit	up	No	2	0	Yes	Yes
1-channel with one complementary output	16-bit	up	Yes	1	1	Yes (OC signal)	No
2-channel with one complementary output	16-bit	up	Yes	2	1	Yes	Yes
High-resolution	16-bit	up	Yes	10	10	Yes	Yes
Low-power	16-bit	up	No	1	0	No	No

1.1.2.1.2. PWM

Pulse-width modulation (PWM) is a type of technique that can be implemented in microcontrollers by using a timer in a specific mode. It is a modulation used in electronics to control the amount of power delivered to a device or component by generating several pulses with different duty cycles in a given period of time or frequency. A duty cycle is the percentage of one period of time in which a signal is active [8]. DC (Direct Current) motors, which are electric motors that convert electrical into mechanical energy [9], respond exclusively to DC signals because these motors rely on the consistent flow of current in one direction to generate the magnetic fields needed for their operation, avoiding instant changes. For that reason, in the presence of a high-frequency signal, the motor's response is limited to its DC component, commonly known as the average voltage.

Expression 1.1 shows the mathematical relation between duty cycle and average voltage, where D is the duty cycle, T_{on} is the time the signal is active, V_{max} is the peak voltage of the signal and V_{avg} is the average voltage.

$$D = \frac{T_{on}}{Period} \times 100\% \quad (1.1)$$

$$V_{avg} = V_{max} \times D \quad (1.2)$$

However, these applications can be divided into two main categories: on the one hand, by using a PWM signal to control the voltage applied to the motor, it is possible to effectively control the amount of energy supplied to a load (the output voltage and hence the current) and, on the other hand, to encode a message on a carrier wave (a square wave, for instance). PWM is widely used in microcontrollers and other digital circuits and has many applications, including controlling the speed and direction of rotation of DC motors, regulating the brightness of LEDs, and so on.

Figure 1.2 depicts that if the duty cycle of the PWM signal is increased, the average voltage supplied to the motor increases too, and therefore its speed.

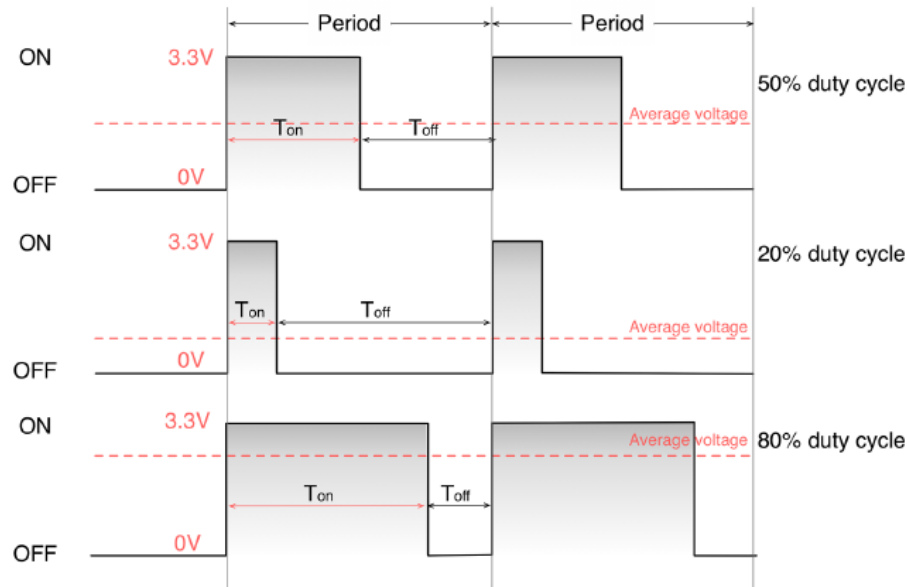


Figure 1.2: Different examples of duty cycles - 50%, 20% and 80% [8].

In comparison to other techniques, PWM has several advantages for controlling DC motors. For instance, it is more efficient than using a resistor to control the voltage because the PWM signal switches the voltage on and off rapidly, reducing the amount of power wasted as heat. PWM control is also useful for applications that need smooth speed modifications since it enables precise and accurate speed control.

1.1.2.1.3. Rotary encoder

An encoder is a device used to measure the position or speed of a rotating shaft or linear motion. Rotary encoders generate signals that are used to measure changes in the angular position, which it is used to determine the speed, the position and the direction of an object.

Rotary encoders are a type of incremental encoders which provide a series of pulses that are generated cyclically as the shaft rotates, indicating the direction and amount of movement relative to a reference point [8]. These incremental rotary encoders are the most widely employed in the industry due to its capability to provide signals that can be simply interpreted to provide motion related information like velocity [8].

These encoders use two quadrature outputs, labeled as A and B, which are 90 degrees out of phase with each other. The direction of the motor is determined by the phase relationship between the two signals, with the leading phase indicating the direction of rotation. Whether phase A leads phase B or phase B leads phase A determines the direction of the motor, forward or backward, as shown in Figure 1.3:

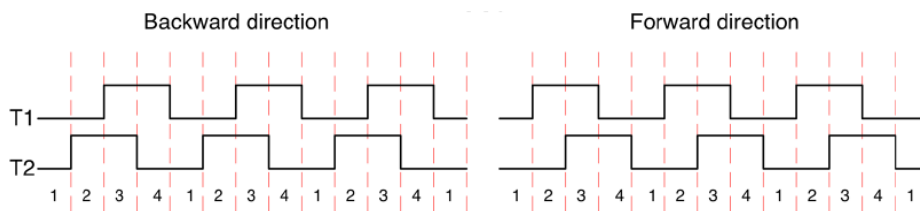


Figure 1.3: Directions of a quadrature encoder [8]

It is also important to point out that by counting the number of pulses generated on both channels over one revolution of the shaft, the encoder can determine the number of counts per revolution (CPR), thus measuring its angular resolution. Higher resolution encoders can detect smaller angular changes and provide more precise position feedback.

1.1.2.1.4. Ethernet

The Ethernet standard is a set of specifications and protocols that define the physical and data link layer standards, commonly, for wired Local and Metropolitan Area Networks (LANs and MANs, respectively). This defines the physical (PHY) layer standards for transmitting data between devices on a network, including the format of data packets, the mechanisms for addressing and error detection and correction. It was commercially introduced in 1980 and first standardized as IEEE 802.3.

Using a Media Access Control (MAC) protocol, Ethernet LAN operation is specified for certain speeds ranging from 1 Mb/s to 400 Gb/s [10]. It works as an interface of, for instance CPUs or FPGAs, for data processing and communicating with the PHY chip. The PHY layer is responsible for transmitting the data received from the MAC layer as an analog signal over the physical network interface, using copper (twisted pair (UTP) and coaxial) or fiber-optic cables.

With the aim of connecting the MAC with the PHY layer, two different types of

interfaces can be used, MII (Media Independent Interface) or RMII (Reduced Media Independent Interface). MII uses a 4-bit wide data bus to transmit and receive data at a rate of 25 MHz, allowing for a maximum data rate of 100 Mbps, and also includes several control signals to manage the data flow and detect errors. Meanwhile, RMII is a more compact version which uses a 2-bit wide data bus and operates at a rate of 50 MHz, allowing the same maximum data rate but using fewer pins and less board space [11].

Commonly, the Ethernet data link uses the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol, which helps to avoid collisions when data from different devices is transmitted on the network simultaneously.

1.1.3. H-bridge

In electronics, sometimes there are heavy loads incorporated into a circuit that are not possible to be powered and driven directly from a microcontroller system. In these cases, an H-bridge driver is used and the microcontroller is responsible for controlling it with a high degree of efficiency. The most common scenario is when there are motors or other high-power devices.

It is called a bridge because it consists of four switching elements, which are usually power transistors or MOSFETs, arranged in two pairs and placed in parallel with the load, controlling the current flow through it [12]. The timing and duration of the switch transitions can be adjusted in order to optimize the load's performance, as it is shown below in Figure 1.4:

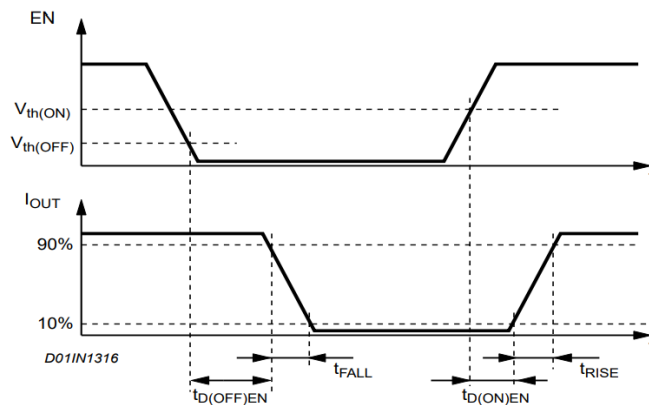


Figure 1.4: Switching characteristic definition [13].

In the current project it has been decided to use an H-bridge. This kind of bridge is typically used to control the direction and speed of DC (Direct Current) motors, enabling bidirectional motor driving by activating one of the diagonally-opposed switch pairs. This means that motors can be driven forward or backward, which

provides greater control and flexibility in areas such as robotics or electric vehicles. H-bridges also allow the use of pulse-width modulation to control the average voltage applied to the motor, which reduces power consumption and increases efficiency.

1.1.4. Controllers

A controller is a device or component that manages or regulates the behavior of a system by adjusting its output signal based on feedback from sensors or other sources of information, using control loops. They are widely used in several applications such as automation or robotics.

As it is shown below in Figure 1.5, the basic structure of a control system that uses a controller is composed of the following main blocks:

- **Controller:** it receives the error signal from the error detector (or the input signal when the loop starts) and determines the appropriate control action to be taken, applying control algorithms to minimize the error.
- **Plant:** it is referred to the motor and is the part of the system that is being controlled. The plant takes the actuating signal from the controller and produces the output action.
- **Feedback elements:** these are the sensor and the feedback path that provides information about the actual value of the magnitude measured of the motor. The feedback signal is fed back to the error detector, which allows the controller to make adjustments as necessary.
- **Error detector:** it compares the desired or reference value of the magnitude to be measured with the actual value and calculates the error signal, which represents the difference between both measures.

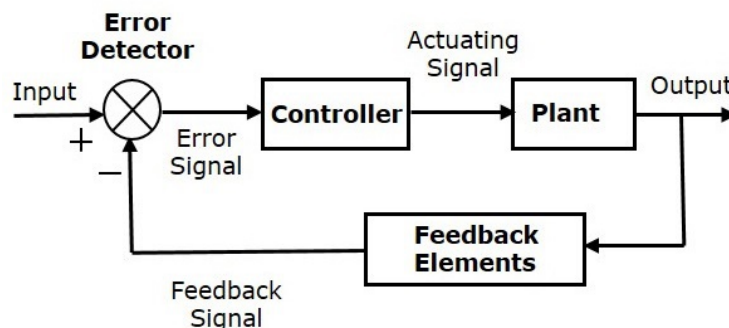


Figure 1.5: Structure of a control system [14].

There are different types of controllers, but this BSc Thesis focuses on the position and the speed ones.

The position control of a DC motor is crucial in applications for precision control systems [9]. The purpose of a position controller is to drive the motor to a desired setpoint, generating a control signal, with high accuracy and stability using feedback. The control signal is typically a voltage or current signal that is sent to a motor or other actuator to drive the device being controlled. On the other hand, speed controllers work in the same way but trying to achieve a specific speed.

1.1.5. ROS

The Robot Operating System (ROS) is an open-source framework that provides a collection of libraries and tools for developing robot software, including drivers, algorithms, and hardware abstraction layers [15]. ROS facilitates the integration of complex robot systems and simplifies the development of new robot applications. It also provides a peer-to-peer (P2P) communication system through its publish-subscribe messaging paradigm. For these reasons, ROS is widely used in the robotics community for both research and commercial applications and supports a variety of robot platforms, such as autonomous vehicles.

However, due to problems such as lack of security and no real-time support, ROS2 was eventually created [16]. One of the key changes in ROS2 is the adoption of a new communication middleware, called Data Distribution Service (DDS), which is an API (Application Programming Interface), and provides better performance, reliability and interoperability. DDS enables different components of a robot system to communicate with each other, regardless of programming language or machine. The figure below (Figure 1.6) represents the different blocks that conform the framework:



Figure 1.6: The ROS ecosystem [17].

It is vital to understand the main components that enable communication in this system. These are:

- **Nodes:** individual programs that perform specific tasks and are able to communicate with each other through the communication infrastructure.
- **Topics:** channels over which nodes can publish messages and subscribe to receive messages.
- **Messages:** data structures that nodes use to communicate with each other over topics.

In a ROS system, nodes can communicate with each other in four different ways, shown in Figure 1.7:

- **Publish-subscribe messaging:** nodes can publish messages on a specific topic, and other nodes can subscribe to that topic to receive the messages.
- **Service request-response:** this is an asynchronous form of communication where a node can request a service from another one, which will then provide a response.
- **Parameters:** they are used for storing data that can be accessed by several nodes, which allows changes in the behavior and settings of the nodes.
- **Actions:** they are a form of communication that provide a way for nodes to execute complex behaviors in a coordinated manner adding support for feedback.

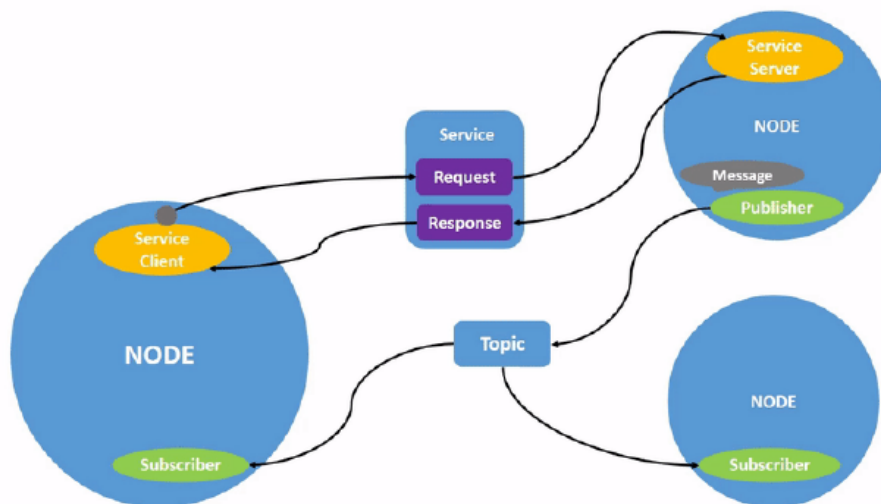


Figure 1.7: Communication between nodes in ROS [18].

1.1.5.1. micro-ROS

Micro-ROS is a version of ROS designed for microcontrollers and other resource-constrained devices which provides a lightweight framework that can be deployed on small embedded devices, reducing costs and increasing development efficiency [19].

It uses the same programming model and communication infrastructure as ROS, but its implementation is optimized for extremely constrained computational resources. Micro-ROS allows ROS2 APIs to be brought to microcontrollers, so that developers can benefit from most relevant, major ROS 2 concepts [20], simplifying the transfer of advanced software to the application level.

As it is pictured in Figure 1.8, micro-ROS reuses the libraries of ROS2. The layers that have been maintained are the ROS Client Library (RCL), the ROS Middleware Interface (RMW) and also, the RCLCPP, which is a C++ abstraction layer on top of the RCL [21]. It is important to point out that the RMW implementation is based on a library called Micro XRCE-DDS, which comes from the ROS2's library, XRCE-DDS (eXtremely Resource Constrained Environment Data Distribution System). This relies on a client-server architecture, which connects low-resource devices (XRCE Clients) to a server (XRCE Agent), acting as a bridge between the clients and the DDS network.

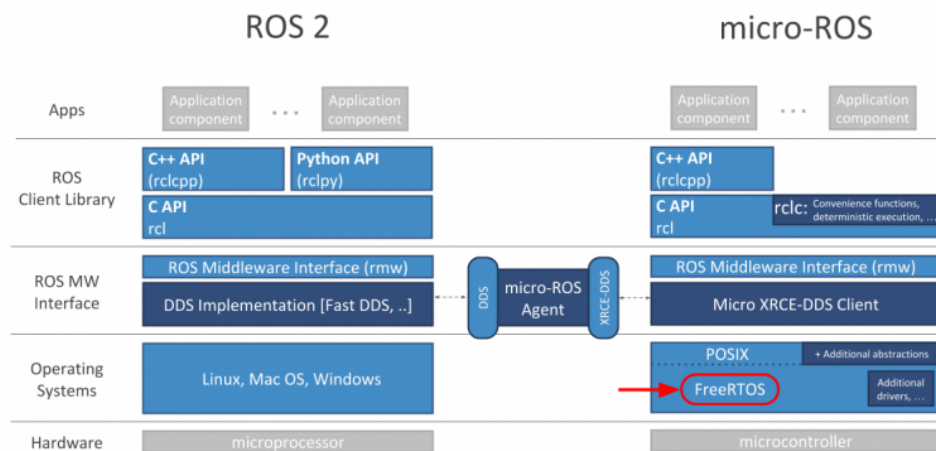


Figure 1.8: Comparison between ROS2 and micro-ROS architecture [21].

Moreover, one of the main features of Micro-ROS is its support for different real-time operating systems (RTOS), such as FreeRTOS, which will be also important in the development of this final degree project.

1.2. Motivations and goals

Taking into account the difficulties in developing motor control systems and managing the communication between their principal functional blocks, the aim of this BSc Thesis is to carry out the design and implementation of a motor control system based on the Robot Operating System, ROS2.

In order to reach this goal, the creation of this control system embedded in a microcontroller will be essential. It will use ROS2 as the communication and control interface, so that anyone with just a few software requirements can control it from their computers. In this way, a scalable, reusable and user-friendly system will be created, which may be implemented as a powerful tool in engine control laboratories.

With the purpose of achieving the objectives set and developing the project in more organized means, the following steps are defined:

- State of the art of control systems.
- Analysis of hardware requirements.
- Analysis of software requirements.
- Design and implementation of the embedded software.
- Unit testing.

1.3. Document structure

This manuscript has been structured in five main sections.

In this first chapter an introduction of the main important aspects related with this project is described, along with a theoretical framework and the motivations and goals of its development.

Chapter 2 describes the hardware selected including a description of the different blocks and the relation among them.

Chapter 3 contains an explanation of the software design developed to facilitate the fulfillment of this project, describing both the requirements gathered and the steps taken to achieve its implementation.

Chapter 4 is focused on the testing process. It describes the tests performed to check the functionality of the project, at both hardware and software levels.

Finally, Chapter 5 describes the results and the conclusions drawn from the thesis, as well as future improvements.

Chapter 2

Hardware architecture

2.1. Introduction

Hardware (HW) refers to the physical and tangible components which integrate a computer or electronic system. Thanks to the interoperability with the firmware, which is embedded into the hardware to control it, and the software that runs programs on top of it, it is capable of running a computing system [23]. Hardware is not only present in computers and electronic devices, but also extends its functionalities to cars, phones, cameras and so on. Furthermore, it is important to notice that the performance of any computer system will depend to a large extent on the characteristics of its hardware components.

The architecture implemented in the current project is explained in this chapter. It is based on the interconnection between motors, encoders, a motor control board and a microcontroller development board.

2.2. Requirements

The development of this BSc Thesis has required a prior analysis of the needs found in the field of control systems and an approach to the potential applications or functionalities to be implemented. As the aim is to facilitate the design process for software developers in control systems, the following technical requirements have been gathered to be taken into account:

- The system must be able to control at least two motors simultaneously.
- It is required that the motor is fitted with an encoder, to allow an easy control of its speed and position.
- The motor selected must be able to rotate in both directions, depending on the polarity of its power supply.
- The chosen board must have enough timers to connect both motors and encoders, and set up the respective PWM signals.

- The power stage must be equipped with two H-bridges to control the motors.
- The microprocessor must have sufficient available memory to allow the use of micro-ROS.
- Ethernet connection is required to communicate with the computer.

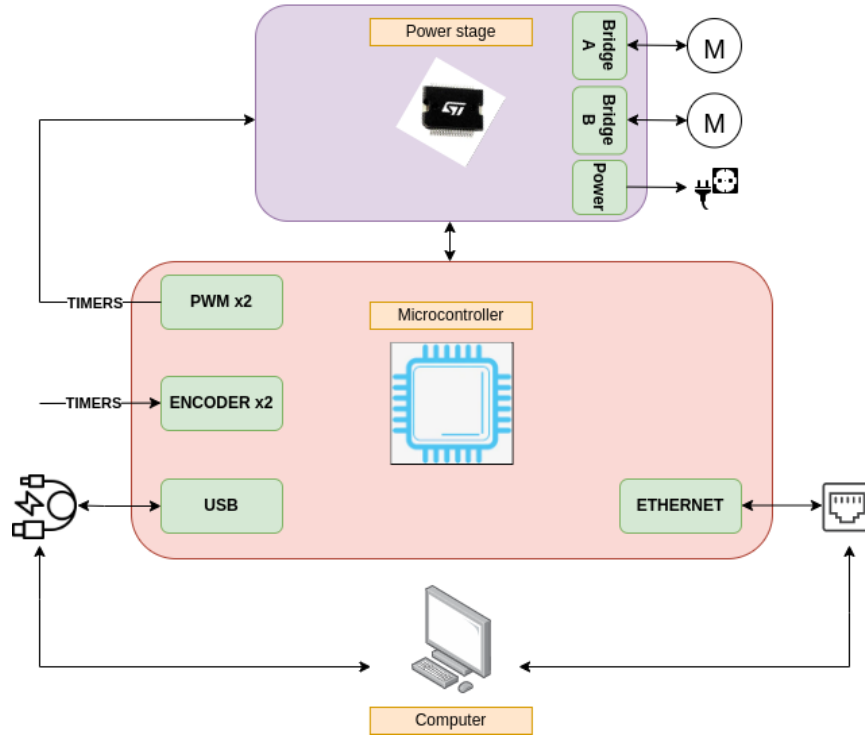


Figure 2.1: Conceptual diagram of hardware requirements.

2.3. Hardware selection

In order to cover the control and connectivity needs of this BSc Thesis, both a NUCLEO STM32F746ZG development board and an X-NUCLEO-IHM04A1 motor control board have been used, whose main features are described below.

2.3.1. Microcontroller

The STM32F746xx microcontroller (see Figure 2.2) is based on the high-performance ARM Cortex-M7 32-bit RISC core and is capable of running at up to 216 MHz clock speed with a power supply from 1.7 to 3.6 V. This device also integrates an extensive range of enhanced I/Os and a variety of on-chip peripherals, including ADCs, DACs, thirteen general-purpose 16-bit timers (including two PWM timers for motor control),

two general-purpose 32-bit timers and communication interfaces such as SPI, I2C, UART, USB, Ethernet, and CAN [24].

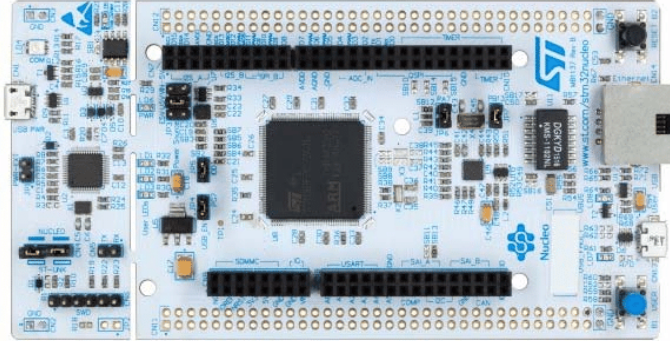


Figure 2.2: NUCLEO STM32F746ZG development board [24].

It also incorporates high-speed embedded memories with a Flash memory up to 1 MB, 320 KB of SRAM (including 64 Kbytes of Data TCM (Tightly-Coupled Memory) RAM for critical real-time data), 16 KB of instruction TCM RAM (for critical real-time routines) and 4 KB of backup SRAM available in the lowest power modes [24].

It has been decided to use this development board due to several factors: it allows working at a suitable clock frequency for the correct operation of the system, its low cost and its numerous timers, which will be needed to generate the PWM signals and for the encoders of the motors. But mainly, because it offers Ethernet connectivity, a fundamental characteristic to develop the software of the project.

In addition, the use of an ST family board provides the possibility of using an efficient, intuitive development environment with a wide range of functionalities, from configuring the pins, peripherals and middleware to facilitate software development to generating the code.

These features make the STM32F7 microcontroller family suitable for a wide range of applications such as motor drive and application control, medical equipment and industrial applications.

2.3.2. Motor control board

The X-NUCLEO-IHM04A1 (see Figure 2.3) is a dual brush DC motor drive expansion board that provides an affordable but robust solution for driving one to four DC motors. Some of the key features [25] of this expansion board are:

- It is compatible with STM32 Nucleo boards and with the Arduino UNO R3.
- It is able to drive dual-bipolar DC or quad-unipolar DC motors.
- A voltage range of 3.3 V to 5 V is required to power the logic circuitry.
- The supply voltage of both motors supports a voltage from 8 V to 50 V DC.
- Its motor phase current is up to 2.8 A RMS (Root Mean Square).

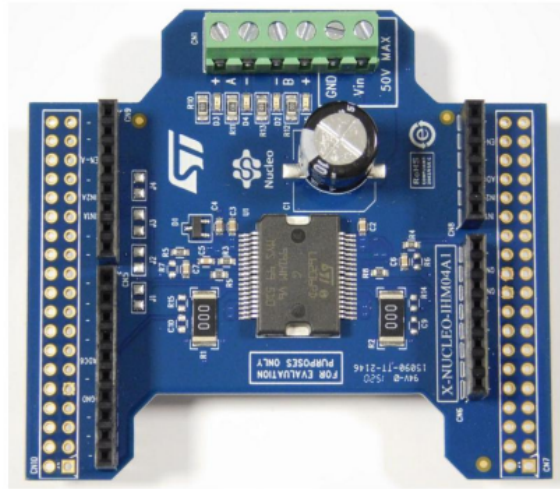


Figure 2.3: X-NUCLEO-IHM04A1 expansion board [25].

In particular, to develop this BSc Thesis the connection and control of two independent bidirectional DC motors simultaneously will be essential. This is achieved by connecting both motors and the power supply to the Nucleo expansion board, as shown in Figure 2.4.

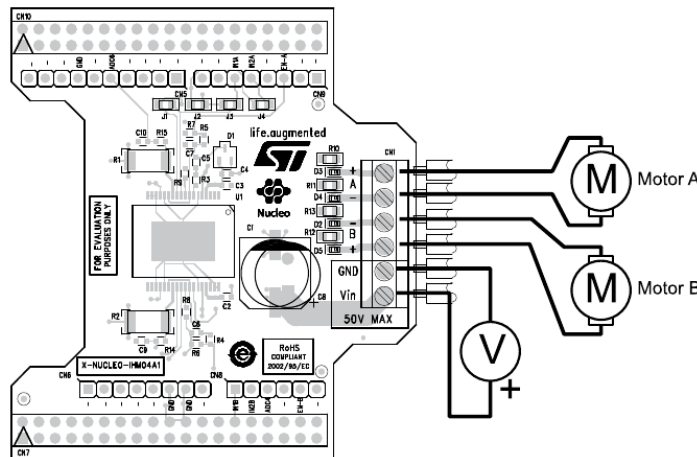


Figure 2.4: Connection of two bidirectional DC motors [25].

2.3.2.1. H-bridge

One of the particularities of the X-NUCLEO-IHM04A1 is that it is equipped with a power stage and an H-bridge. The power stage is responsible for amplifying the control signal received and supplying the current necessary to power the DC motor. As it is explained in Section 1.1.3, this kind of bridge is typically used to control the direction and speed of DC motors and it allows the use of PWM to control the average voltage applied to the motor.

It is based on the L6206 device which is a DMOS (Double-Diffused MOSFET) dual full bridge which integrates two independent power MOS full bridges [26]. Regarding its operation, the inputs (IN1 and IN2) of both bridges (A and B) can be configured at high or low level by looking at its truth table (see Table 2.1), depicted hereafter:

Table 2.1: Truth table of an H-bridge [26].

Inputs			Outputs	
EN	IN1	IN2	OUT1	OUT2
L	X ⁽¹⁾	X ⁽¹⁾	High Z ⁽²⁾	High Z ⁽²⁾
H	L	L	GND	GND
H	H	L	Vs	GND
H	L	H	GND	Vs
H	H	H	Vs	Vs

1. X = don't care.

2. High Z = high impedance output.

Depending on the set input values, the motor will react in different ways (represented in OUT1 and OUT2):

- If the enable (EN) is deactivated, the motor does not move.
- Once the enable is activated, the motor will move only if the inputs of the IN1 and IN2 signals are different (one at HIGH and the other at LOW). According to which one is at high level, the motor will rotate in one or another direction.
- Finally, if both input signals are set to the same level, a short circuit will be triggered in the power stage and the motor will not rotate.

2.3.3. Motor

A DC motor is an electromechanical system that converts electrical energy into mechanical rotational motion by creating a magnetic field powered by direct current. There are different types but the one chosen is a brushed DC motor. It incorporates two key components: a stator, which is the stationary part and contains permanent

magnets, and a rotor, which is the rotating part [27].

Brushed DC motors operate through the interaction of a rotating coil and the magnets surrounding it. The rotation of the coil causes the contact between the commutator and the brush to change, altering the current flow in the coil. This mechanism allows the motor to generate motion [28].

By varying the voltage applied to the motor, it is possible to control its speed and apply control techniques, such as PWM. Moreover, simply by shifting the polarity of the power supply, DC motors can quickly change their rotational direction. They are relatively simple to use, reduce power consumption and their size is perfect for applications where space and weight constraints apply. However, they also have some disadvantages [29], such as: they require higher maintenance, are generally less efficient than brushless DC motors and are not suitable for high-power applications.

One of the reasons why brushed DC motors are often preferred over other types of motor is their ability to precisely control their speed, thus facilitating position control, which is the main reason why they have been selected for the development of the BSc Thesis.

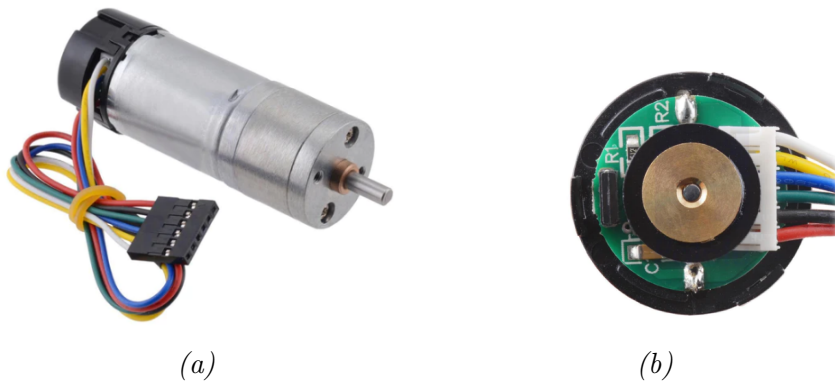


Figure 2.5: Motor selected (a) with its encoder (b) [30].

Specifically, a Pololu Metal Gearmotor with a rotary encoder has been chosen, shown in Figure 2.5, whose nominal voltage to operate optimally is 12 V [30]. This encoder provides a series of pulses that are generated cyclically as the shaft rotates, indicating the direction and amount of movement with respect to a reference point. It has six color-coded wires: two for powering the motor, two for powering the encoder and lastly, two for connecting the encoder to each of the channels.

2.4. Complete hardware system diagram

Finally, after analysing the requirements of the project and selecting the electronic elements that will be part of it, they will be interconnected, according to the

indications of each datasheet respectively, as illustrated in Figure 2.6.

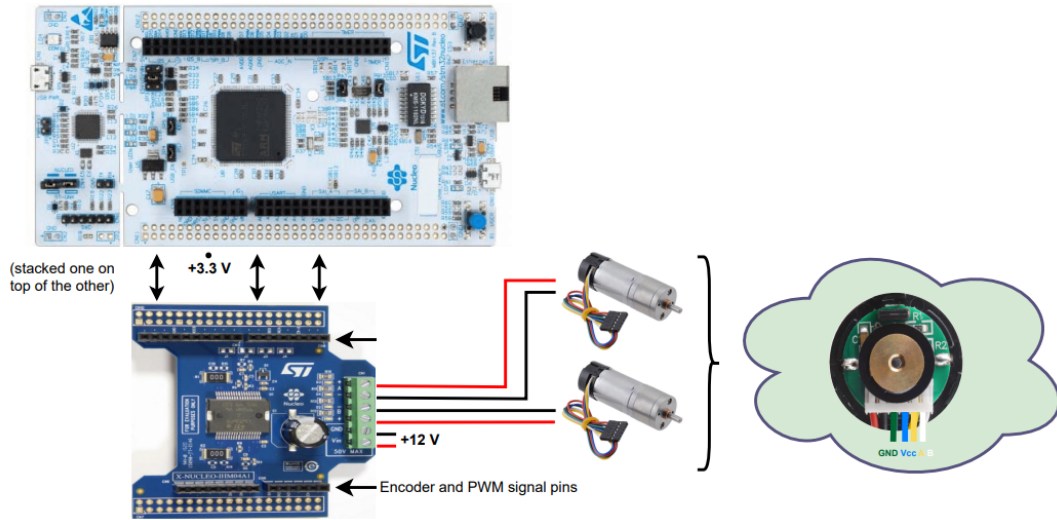


Figure 2.6: Complete hardware system diagram.

One of the advantages of using these boards is that they are designed to fit together with compatible pin headers and connectors, so to connect the STM32F7 microcontroller to the X-NUCLEO expansion board, the boards are simply stacked together. Thus, the functionality of the matching pins is the same.

In addition, it is important to consider that each of the encoders and PWM signals are configured with different timers of the microcontroller for their correct operation. This will be explained in more detail in the next chapter, but schematically the following connections have been made:

Table 2.2: Hardware connections.

GPIO	PERIPHERALS	FUNCTION
PF15	-	EN-A
PF5	-	EN-B
PE9	PWM_0 TIM1	Bridge A CH1
PE11	PWM_0 TIM1	Bridge A CH2
PA3	PWM_1 TIM2	Bridge B CH4
PA5	PWM_1 TIM2	Bridge B CH1
PB4	ENCODER_0 TIM1	Bridge A CH1
PB5	ENCODER_0 TIM1	Bridge A CH2
PD12	ENCODER_1 TIM2	Bridge B CH1
PD13	ENCODER_1 TIM2	Bridge B CH2

Finally, it is important to highlight that the pins configured for the PWM must

match the input pins of the different bridges ($IN1_A$, $IN2_A$, $IN1_B$, $IN2_B$), so we jumper PE9 to PE11 to match $IN2_A$ and PA5 to PC0, coinciding with $IN2_B$. The PE11 and PA3 pins correspond correctly to $IN1_A$ and $IN1_B$, respectively.

Chapter 3

Software

3.1. Introduction

According to Leon J. Osterweil [31], software is not only code to be executed on a computer, but also is a compound of different specifications, designs and results, a large collection of constraints and relations, which lead to the creation of different programs required by the users.

Broadly speaking, software is a set of instructions, data or programs used to execute specific tasks on electronic devices. It provides the data needed by computers for their operation and meets users' needs. Unlike hardware, software it's non-tangible and its main function is to handle physical entities, commonly with reduced resources and capabilities, in a reliable and efficient way [32]. However, they are closely related to each other, especially when it comes to making an embedded system work.

The development software platform used in this BSc Thesis is known as STM32CubeIDE. The STM32 software development platform does not only provide the necessary drivers to control their peripherals (timers, PWM, encoders, etc.), but also integrates a Hardware Abstraction Layer (HAL), which is an official library to develop STM32 applications. The HAL will be used in the functions of the specific middleware implemented.

In this chapter, all the concepts related with the design and implementation of the different necessary software blocks will be described in detail.

3.2. Requirements

The development of this BSc Thesis has required a prior analysis of the needs found in the field of control systems. In the same way as in the hardware chapter, and with the intention of facilitating the design process for software developers, the following technical requirements have been gathered:

- The software implemented must offer a scalable and easily understandable code for the user.

- The software must be able to implement the configuration for both motors (PWM and encoder signals) and control them in position or speed.
- The software must implement two parallel PID [33] (Proportional Integral Derivative) controllers to drive each motor individually.
- FreeRTOS [34] must be used to create different tasks to control the motor system. It is important to consider that controllers and communication processes must be implemented as individual tasks to ensure real-time execution of the application.
- The use of ROS2 [15] communications is essential in order to achieve greater flexibility in terms of interoperability, integration and scalability of the system.
- It is necessary to implement lwIP [35] as the TCP/IP stack for embedded system communications, in order to reduce memory usage.

3.3. Middleware

Middleware is a software layer that lays between the application and the driver layers. It enables connectivity between two or more devices or components of a distributed network, improves interoperability and removes complexities in the development of applications, offering developers a uniform interface for support [36].

3.3.1. Timers

As mentioned in previous chapters, timers behave like counters whose operation depends on their source clock. The STM32 family of microcontrollers can be synchronised using two different clock sources: an internal RC oscillator or HSI (High Speed Internal) or an external dedicated crystal oscillator or HSE (High Speed External) [8]. In this project, the HSE was preferred to the HSI because it is a much more accurate clock signal. However, it has a longer start-up time and higher power consumption, but these are not a problem in this case. The crystal oscillator makes a scale from 25 MHz, which is the input frequency, to 216 MHz, which is the maximum timer frequency.

The microcontroller has different timers (Basic Timers, General Purpose Timers and Advanced Timers) that can be configured for several purposes, but they all share the same structure [8]. The three main parameters are:

- **Prescaler (PSC):** it is in charge of dividing the timer clock by any integer between 1 and 65535.
- **Counter Mode:** it defines the counter direction of the timer, but the one set is `TIM_COUNTERMODE_UP`.
- **Counter Period:** it sets the maximum value to which the timer will be able to count, after which the count will be restarted. This parameter is also known as ARR (AutoReload Register).

Finally, it is important to explain how the counters of these timers work in order to understand how to set the timer's counting period. A timer typically counts up from zero to a specified value, which cannot be higher than the maximum unsigned value for its resolution (for instance, a 16-bit timer overflows when the counter reaches 65535 and the counter is then reset) [37], but it can also count down. This value is the ARR. As it can be seen in Figure 3.1, once the counter reaches the ARR value, the timer interrupt is triggered resulting in an interrupt request (IRQ).

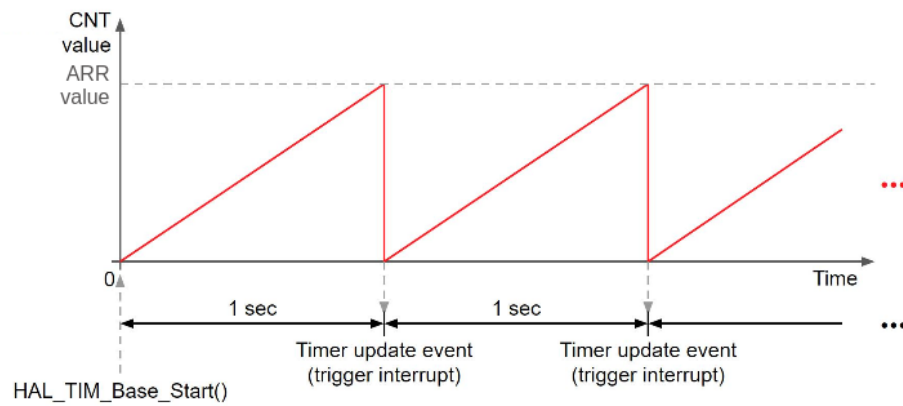


Figure 3.1: Timer callback representation [38].

In the next sections, with the aim of configuring the middleware implemented, different timers have been chosen with specific configuration.

3.3.2. PWM

To generate the PWM signals, thanks to the development environment, it is possible to carry out specific configurations for each peripheral according to the user's needs. Therefore, in order to simplify the complexities that this process may entail and reduce the software development time for programmers, different libraries have been implemented to configure the necessary parameters and call specific functions from an intuitive interface.

The goal is to program both timers (one for each PWM signal) with a time base so that the frequency of these signals will be 50 kHz. This choice is made because there are several fundamental frequencies in the transfer function of the motor and it is intended to eliminate the higher frequencies with a low-pass filter. It is important to note that since it is impossible to implement an ideal low-pass filter, the D/A conversion of the control signal and the continuous signal will not give an exact reconstruction of the sampled signal [39].

The configuration of the timers included in the implemented middleware functions is

as follows:

1. Two timers, *TIM1* and *TIM2*, have been chosen to control the PWM signals of *BRIDGE A* and *BRIDGE B*, respectively. These timers share their configuration with the only exception that the channels chosen in the first case have been *CH1* and *CH2*, and *CH1* and *CH4* in the second case, due to the availability of the pins on the board. We configure these channels in the PWM generation option.
2. The next step is to configure the counter settings. First, the counter mode is set to *up*. Then the value of the prescaler (PSC) will be 0 so that no clock division is produced, using its maximum resolution. Finally, the value of ARR is calculated as shown in Equation 3.2 [37]:

$$ARR = \frac{f_{TIM}}{f_{PWM} \cdot (PSC + 1)} - 1 \quad (3.1)$$

$$ARR = \frac{216MHz}{50kHz} - 1 = 4319 \quad (3.2)$$

3. To generate the variable frequency signals, the output comparison mode of the timers is used. Depending on the desired duty cycle (DC), a certain value for the CCRx (Capture/compare register) is calculated for this function (see Equation 3.4 [37]). Thus, the CCRx value is set on one timer channel and on the other channel it is set to 0, so that the movement is generated either forwards or backwards.

$$Duty_cycle(\%) = \frac{CCRx}{ARR} \cdot 100 \quad (3.3)$$

$$CCRx = \frac{DC \cdot ARR}{100} \quad (3.4)$$

4. Finally, it is important to note that it is necessary to create a function to activate the pins that match the enable pins of the bridge indefinitely.

All of the above configuration is simplified and abstracted into the functions shown in Code 3.1. In all of them, the user must specify the jumper (A or B) as a parameter, depending on which one is being configured. The headers of these functions are the following:

```

1  /** This function starts the appropriate timers in PWM mode and set
    enables to default.*/
2  void PWM_TIMn_Start (PWM_Bridge_t bridge);
3  /** This function enables the PWM mode in the bridge selected by the
    user.*/
4  void PWM_Enable(PWM_Bridge_t bridge);
5  /** The function changes the duty ratio for Timer module in PWM mode.*/
6  void PWM_TIMn_Set_Duty (PWM_Bridge_t bridge, float duty_cycle);
7  /** The function gets the duty ratio for Timer module in PWM mode.*/
8  float PWM_TIMn_Get_Duty (PWM_Bridge_t bridge);
9  /** This function sets the PWM to 0.*/
10 void PWM_TIMn_Stop(PWM_Bridge_t bridge);

```

Code 3.1: PWM middleware headers.

The source code is implemented in the *pwm.c* and *pwm.h* files allocated in the source and include folders.

3.3.3. Encoders

In the same way as in the previous section, the goal pursued is to simplify the configuration of the rotary encoders. The microcontroller chosen holds specific peripherals to capture the incoming pulses on the input channel of the timer.

It is important to take into account the characteristics of the motors used, both the reduction and the CPR (Counts Per Revolution), which are listed in their datasheet.

The configuration of the timers employed is as follows:

1. Two timers, *TIM3* and *TIM4*, have been chosen to configure the encoder signals of *BRIDGE A* and *BRIDGE B*, respectively. We configure these timers in the Encoder mode option, which will combine *CH1* and *CH2* for their set up.
2. Next, the counter settings are configured. The counter mode is set to up and the value of the PSC to 0.
3. The encoder mode is configured with the option *Encoder Mode TI1 and TI2*, to set the *X4* mode so that the TIMx_CNT register is updated on every edge of both channels. This option doubles the capture frequency [8].
4. The ARR value is set to the number of pulses per lap, which is obtained from Equation 3.5. By doing so, when the CNT register counts up to the value in ARR, an entire revolution will have been completed and an update event triggers an interrupt.

$$Pulses_per_revolution = CPR \cdot reduction \quad (3.5)$$

5. It is necessary to activate global interruptions for these timers. The timer interrupt routine will update the number of revolutions made by the motor, which will then be used to calculate its position (see Equation 3.6). The counter of revolutions shall increase or decrease depending on the direction of movement of the rotary encoder.

$$Position(rad) = revolutions + \frac{CNT}{ARR} \cdot 2\pi \quad (3.6)$$

As in the case of PWM, in all the implemented functions, the user must specify the jumper (A or B) as a parameter, depending on which one is being configured. The headers of these functions are shown in Code 3.2:

```

1 /** This function starts the appropriate timers in Encoder mode and
   sets CNT.*/
2 void Encoder_TIMn_Start(PWM_Bridge_t bridge);
3 /** This function gives us the position of the motor's encoder.*/
4 float Encoder_GetPosition(PWM_Bridge_t bridge);
5 /** This function sets a reset state, to be able to know what is the
   initial state of the encoder.*/
6 void Encoder_Set_Reset(PWM_Bridge_t bridge);
7 /** This function stops appropriate Timer module in Encoder mode.*/

```

```
8 void Encoder_TIMn_Stop(PWM_Bridge_t bridge);
```

Code 3.2: Encoder middleware headers.

The source code is implemented in the encoder.c and encoder.h files allocated in the source and include folders.

3.3.4. Controller

As previously explained in Section 1.1.4., a controller is a component that is used to control the behaviour of a system. The objective to be achieved is to drive the motor to a desired position, for which a position controller will be required.

The control architecture adopted is a PID (Proportional Integral Derivative) controller. It is a third order controller with a proportional, integral and derivative block in direct loop (see Figure 3.2).

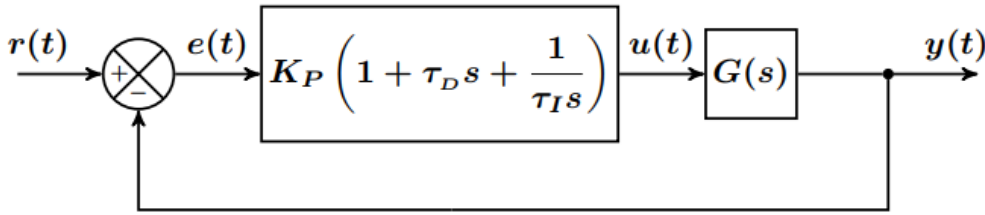


Figure 3.2: PID feedback control scheme [40].

An analysis for a position control system is performed for a motor modelled as an order 2 system [41], whereby $G(s)$, which corresponds to its transfer function, would be as follows (see Equation 3.7):

$$G(s) = \frac{K}{s(s+p)} \quad (3.7)$$

This controller calculates an output signal based on its three terms: a proportional term (K_P), which provides an output proportional to the error signal, an integral term (τ_I), that integrates the error and takes into account the accumulated error over time and a derivative one (τ_D), which delivers an output proportional to the rate of change of the error with respect to time [33]. These three terms relate to error, accumulation of error and change in the error respectively. Its corresponding transfer function and error function are as follows [40]:

$$H_{PID}(s) = \frac{KK_P\tau_D \left(s^2 + \frac{s}{\tau_D} + \frac{1}{\tau_D\tau_I} \right)}{s^2(s+p) + KK_P\tau_D \left(s^2 + \frac{s}{\tau_D} + \frac{1}{\tau_D\tau_I} \right)} \quad (3.8)$$

$$H_{e,PID}(s) = \frac{s^2(s+p)}{s^2(s+p) + K K_P \tau_D \left(s^2 + \frac{s}{\tau_D} + \frac{1}{\tau_D \tau_I} \right)} \quad (3.9)$$

The aim of implementing this controller is to be able to control the position of the motor so that by setting an angle, it will rotate to that position. In addition, the motor must offer resistance to being rotated manually, so that if it is moved from the set position, it will be able to return to it. To this end, functions are implemented that will simplify this configuration process. Functions are created to determine the PID parameters as well as to calculate the error signal generated by the system (see Equation 3.10). The block diagram in Figure 3.3 shows the action loop of the controller, where the input is the desired position and the output is the duty cycle that will be applied to the PWM signals to achieve the movement of the motor. The error signal is calculated as the difference between the desired position (set point) and the input received by the controller.

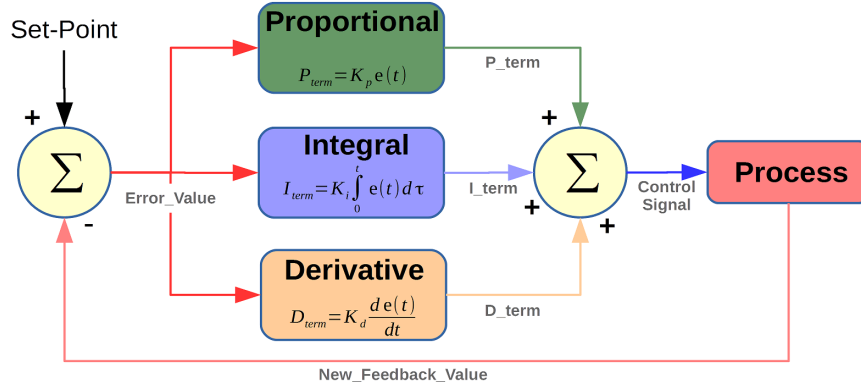


Figure 3.3: PID controller block diagram [42].

$$u[k] = K_P \left(e[k] + \tau_D \cdot \frac{(e[k] - e[k-1])}{T} + \frac{T}{\tau_I} \cdot \sum_{i=0}^k e[i] \right) \quad (3.10)$$

Occasionally, control systems experience a situation where the controller's output keeps accumulating, even when the system is not able to respond due to saturation or constraints [43]. This typically involves the integral term, which may continue to integrate and accumulate the error signal over time. This produces an abrupt jump that will break the feedback loop, causing instability, which is called *windup* [44]. The idea is to implement a method that prevents the integral action of the system from continuing to grow once it is in the saturation limits, so that the controller is able to act directly without having to wait for the accumulated error to be fully discharged.

The headers of the functions implemented are shown in Code 3.3:

```

1 /** This function sets the parameters for the PID controllers.*/
2 void PID_set_parameters (PWM_Bridge_t bridge, float kp, float kd, float
   ki, float windup_limit);
3 /** This function establishes the initial setpoint that the PID will
   try to achieve.*/

```

```

4 void PID_setPoint (PWM_Bridge_t bridge, float position);
5 /** This function executes the PID controller.*/
6 float PID_apply(PWM_Bridge_t bridge, float input);
7 /** This function sets the period of the controller in ms.*/
8 void PID_set_period(PWM_Bridge_t bridge, float period);
9 /** This function resets the parameters for the PID controllers.*/
10 void PID_reset(PWM_Bridge_t bridge);

```

Code 3.3: PID controller middleware headers.

The source code is implemented in the `pid_controller.c` and `pid_controller.h` files allocated in the source and include folders.

3.3.5. Third party middlewares

3.3.5.1. FreeRTOS

FreeRTOS is an open source real-time operating system (RTOS) designed to perform multiple tasks at the same time for time-critical applications on microcontrollers, meeting certain deadlines [34]. It is under a MIT license, which is a software license that gives permission for users to reuse code for any purpose, and it is available for more than 40 architectures [34].

The scheduler used in FreeRTOS must guarantee meeting hard real-time constraints, which means that its behaviour must be predictable. This is achieved by allowing the user to assign a priority to each thread of execution, commonly known as a task, so that the scheduler knows which thread to execute first [45]. Its main features include, among others, interrupt handling, synchronisation (via mutex, semaphores and so on) and high portability.

To implement FreeRTOS in this BSc Thesis, the option provided by the CubeIDE environment has been enabled. Then, two tasks have been created, one for each motor, which have been assigned the same priority, and another task related to LwIP, which will be explained in the next section, with a higher priority.

In these tasks, the motor peripherals are activated and the parameters of the PID controller are configured for the first time. It is important to understand that for the proper functioning of the system, the controller input must be the position sampled with the motor encoder and the output, the duty cycle applied to the PWM. The middleware functions implemented to carry out the PID controller tasks are contained in `PID_tasks.c` and `PID_tasks.h` and their headers are as follows:

```

1 /** This function enables the PID tasks.*/
2 void PIDtask_enable(PWM_Bridge_t bridge);
3 /** This function disables the PID tasks.*/
4 void PIDtask_disable(PWM_Bridge_t bridge);
5 /** This function returns true if the PID tasks are enabled or false if
   not.*/
6 bool PIDtask_isEnabled(PWM_Bridge_t bridge);

```

```

7 /** This function defines the PID task for PWM_BRIDGE_A.*/
8 void PID1task();
9 /** This function defines the PID task for PWM_BRIDGE_B.*/
10 void PID2task();

```

Code 3.4: PID tasks (FreeRTOS) middleware headers.

3.3.5.2. LwIP

LwIP is a Lightweight Internet Protocol which offers a simple and open-source implementation of the TCP/IP (Transmission Control Protocol/Internet Protocol) stack for embedded systems [35]. It consumes minimal system resources and memory, making it suitable for microcontroller-based systems with tens of kilobytes of free RAM and room for around 40 kB of ROM/flash memory [35]. Otherwise, internet connectivity would not be an option as the typical TCP/IP stack is too resource-intensive. Its main characteristics are [46]:

- Implement several TCP/IP protocols such as: IPv4, IPv6, ICMP (Internet Control Message Protocol), UDP (User Datagram Protocol), TCP y ARP (Address Resolution Protocol), among others.
- Includes support for the DHCP (Dynamic Host Configuration Protocol) and DNS (Domain Name System) clients.
- It offers a socket API (Application Programming Interface) for application development, used in the POSIX (Portable Operating System Interface) sockets specification. This improves the compatibility between different operating systems and facilitates the development of software for programmers.

In the current final degree project it has been necessary to activate LwIP to establish an Ethernet connection (TCP/IP) between the ST board and the computer from which ROS is running. For this purpose, in the development environment, Ethernet is first enabled in RMII mode with global interrupts. Next, LwIP is enabled and instead of using the support code provided by CubeIDE, another configuration that better fits our project needs will be used based on the source code of the *NomadRehab* project available on GitHub [47]. In the settings, ICMP, DNS, UDP and TCP are activated as protocols and an address is specified for the board, as well as a gateway for the further connection to the computer.

3.4. ROS2

3.4.1. micro-ROS

The last step is to create an application in microROS which, by using all the generated code libraries and thanks to the connection of the board to the computer, allows the user to change the position of the motors.

For this purpose, a topic and a subscriber have been generated for each motor and also, a common timer for both. In this way, the subscriber listens to the position requests received from the user, the publisher constantly publishes the current position of the motor and the timer sends a signal to publish the new messages. The message type used to set the position of the motor and to get it is *Float_32*. In addition, another subscriber is generated for each motor to allow the user to enable or disable the desired PID controller. To this end, a *Boolean* message is used.

Table 3.1: Communication topics.

Topic	Data type	Description
/motor1/set_pos	std_msgs/msg/float32	Set motor position BRIDGE_A
/motor1/get_pos	std_msgs/msg/float32	Get motor position BRIDGE_A
/motor2/set_pos	std_msgs/msg/float32	Set motor position BRIDGE_B
/motor2/get_pos	std_msgs/msg/float32	Get motor position BRIDGE_B
/motor1/enable	std_msgs/msg/bool	Enable PID task BRIDGE_A
/motor2/enable	std_msgs/msg/bool	Enable PID task BRIDGE_B

To configure the environment it will be necessary to change and include different dependencies. MicroROS offers different transport options, but since in this project the LwIP middleware has been implemented, which contains its own socket library, no further configuration is required for UDP transport. MicroROS is a static library and although the memory it needs is allocated at compilation time, it is also necessary to allocate a heap of memory to the publisher and subscribers at runtime. This memory is known as dynamic memory and the implemented functions come from the *NomadRehab* project [47].

Finally, a microROS task is created and it will be executed in parallel with the FreeRTOS tasks, explained in Section 3.3.5.1., which handle the PID controllers. The functions created are located in `microros_app.c` and `microros_app.h`.

Chapter 4

Testing

Once the project configuration is finished, the next steps to be taken will focus on testing that everything works correctly. The testing process is essential to detect errors in the configuration and operation in order to develop a fully functional project. Therefore, this section will be divided into different parts, testing that each implemented software block works independently but also together and inserting evidence that corroborates it.

The debugging tool used in the project has been the SEGGER J-LINK debugger, which is available for installation on STM32 boards, and provides higher speed and better software support than ST-LINK [48], which is the default debugger.

4.1. PWM middleware

Firstly, the PWM signals will be tested to ensure that they are generated correctly on the pins configured for this purpose and that the middleware implemented works as expected. In order to do this, the duty cycle values will be set for both motors and the oscilloscope will be used to visualise these signals and check that the module is working as expected.



Figure 4.1: PWM test on the oscilloscope.

When testing the pins configured as PWM, Figure 4.1, only one of the channels is activated at 3.3 V, leaving the other one at 0 V. As explained in previous sections, this configuration makes the motor capable of moving in one direction. In addition, the function which sets the duty cycle also works correctly. This can be verified by the fact that when the duty is set to 50%, the signal is activated for half of one period and set to 0 for the other half. Furthermore, it is important to highlight that this function has also been tested by changing the duty cycles.

4.2. Encoder middleware

The next step is to test that the encoder is working properly. To this end, the first step is to check that the signals read on both output channels correspond to what would be expected from a quadrature encoder. As it is depicted in Figure 4.2, square signals are generated in both channels 90 degrees out of phase.

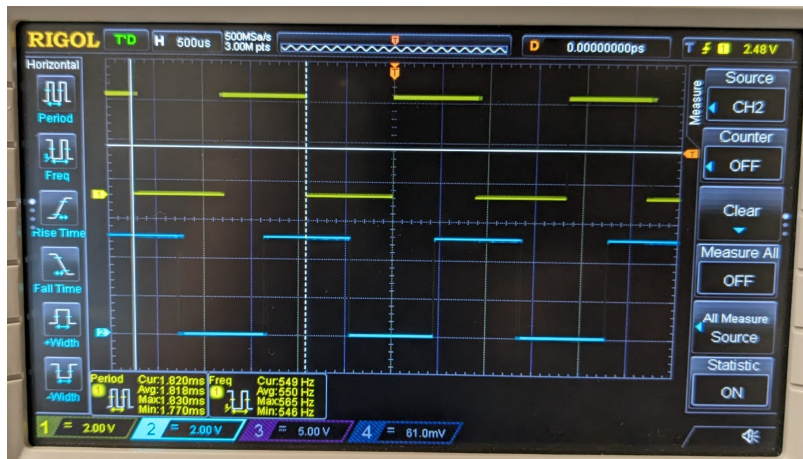


Figure 4.2: Encoder test on the oscilloscope.

Furthermore, thanks to the debug tool it has been verified that the different middleware functions implemented work as expected in both motors. First, it was checked that the respective CNT registers varied as the motor moved, increasing or decreasing the counter value depending on the direction of rotation. It was then verified that the IRQ defined in the code jumped when the ARR value was reached. Finally, it was observed that the final calculated position value corresponds to the rotation made by the motor.

4.3. PID middleware

Once the operation of the PWM and the encoder had been checked, the PID controller was tested. It has been verified in both motors that once the controller is activated it is difficult to move the encoder manually, and if it does move, it applies a rotating

force in the opposite direction to correct this displacement. In addition, it has been proved using different input values that the output values generated are as expected. Therefore, the motor moves correctly towards the indicated position thanks to the implemented functions, generating a small error that is calculated at each iteration.

4.4. LwIP middleware

Once the hardware part has been tested, it will be checked that the interconnection of the board with the computer via ethernet and the implementation of LwIP middleware are correct. This has not been straightforward because the configuration generated by CubeIDE was not the correct one. The problem was due to a misassigned pin and it was noticed because the PHY was configured correctly but no network traffic was detected in Wireshark. Wireshark is a network traffic analysis tool which can be used to check that the connection with ethernet is successful and that data packets are being exchanged.

The first test has consisted of obtaining an IP address through the DHCP protocol. Once obtained, IP addresses for both the board and the computer have been assigned and configured in the same range. The software creates a socket on port 8888, which is the one chosen for the ethernet configuration, and it is possible to verify with Wireshark that there is network traffic. Nevertheless, it can also be verified by establishing the connection to the micro-ROS agent as will be explained in the next section (see Figure 4.3).

4.5. ROS middleware

The last step is to test the micro-ROS application. It must be verified that the connection between it, the XRCE-DDS agent and ROS2 is performed correctly. To carry out this part of the test, it is necessary to have previously integrated micro-ROS on the ST microcontroller, the necessary packages for its implementation and the agent to which it will be connected to communicate within the ROS2 environment (with anyone who subscribes to the published topics).

First of all, it is necessary to verify that the connection with the agent through port 8888, which is where the ethernet connection was configured, is working. It can be observed in Figure 4.3 that the packets sent and received through ethernet between the board and the computer are handled through UDP requests.

```

maria@maria-HP-Laptop-15-da0xxx:~/Documents/GitHub/ControlMotorROS/software/firmware$ sudo MicroXRCEAgent udp4 --port 8888 -v6
[1687281816.236349] info | UDPv4AgentLinux.cpp | init | running... | port: 8888
[1687281816.237089] info | Root.cpp | set_verbose_level | logger setup | verbose_level: 6
[1687281834.760106] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x00000000, len: 24, data:
0000: 00 00 00 00 01 10 00 50 52 43 45 01 00 01 0F 4A F0 88 16 01 00 FC 01
[1687281834.760461] info | Root.cpp | create_client | create | client_key: 0x4AF08816, session_id: 0x81
[1687281834.760503] info | SessionManager.hpp | establish_session | session established | client_key: 0x4AF08816, address: 192.168.2.100:12020
[1687281834.760788] debug | UDPv4AgentLinux.cpp | send_message | [** <<UDP>> **] | client_key: 0x4AF08816, len: 19, data:
0000: 81 80 00 00 04 01 08 00 00 58 52 43 45 01 00 01 0F 00
[1687281836.778921] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x4AF08816, len: 40, data:
0000: 81 80 00 00 01 07 20 00 0A 00 01 01 03 00 00 11 00 00 00 01 00 00 09 00 00 00 70 6F 73 69
0020: 74 69 6F 6E 00 00 00 00
[1687281836.790540] info | ProxyClient.cpp | create_participant | participant created | client_key: 0x4AF08816, participant_id: 0x000(1)
[1687281836.790868] debug | UDPv4AgentLinux.cpp | send_message | [** <<UDP>> **] | client_key: 0x4AF08816, len: 14, data:
0000: 81 80 00 00 05 01 06 00 00 0A 00 01 00 00
[1687281836.791811] debug | UDPv4AgentLinux.cpp | send_message | [** <<UDP>> **] | client_key: 0x4AF08816, len: 13, data:
0000: 81 80 00 00 0A 01 05 00 01 00 00 00 80
[1687281836.793364] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x4AF08816, len: 13, data:
0000: 81 80 00 00 0A 01 05 00 01 00 00 00 80
[1687281836.795791] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x4AF08816, len: 80, data:
0000: 81 80 01 00 01 07 48 00 00 05 00 02 02 03 00 00 3A 00 00 00 12 00 00 72 74 2F 6D 6F 74 6F 72
0020: 31 2F 73 65 74 5F 70 6F 73 00 00 01 1E 00 00 00 73 74 64 5F 6D 73 67 73 3A 3A 6D 73 67 3A 3A 64
0040: 64 73 5F 3A 3A 46 6C 6F 61 74 33 32 5F 00 00 01
[1687281838.215852] info | ProxyClient.cpp | create_topic | topic created | client_key: 0x4AF08816, topic_id: 0x000(2), participant_id: 0x000(1)
[1687281838.216017] debug | UDPv4AgentLinux.cpp | send_message | [** <<UDP>> **] | client_key: 0x4AF08816, len: 14, data:
0000: 81 80 01 00 05 01 06 00 00 00 00 02 00 00
[1687281838.216067] debug | UDPv4AgentLinux.cpp | send_message | [** <<UDP>> **] | client_key: 0x4AF08816, len: 13, data:
0000: 81 80 00 00 0A 01 05 00 02 00 00 00 80
[1687281838.218418] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x4AF08816, len: 13, data:
0000: 81 80 00 00 0A 01 05 00 02 00 00 00 80
[1687281838.219451] debug | UDPv4AgentLinux.cpp | recv_message | [==> UDP <==] | client_key: 0x4AF08816, len: 24, data:
0000: 81 80 02 00 01 07 10 00 00 0C 00 04 04 03 00 00 02 00 00 00 00 00 00 00 01
[1687281838.219846] info | ProxyClient.cpp | create_subscriber | subscriber created | client_key: 0x4AF08816, subscriber_id: 0x000(4), participant_id: 0x000(1)

```

Figure 4.3: Agent connection log.

Additionally, it is verified that the session is established and the topics defined in the middleware functions, as well as the publishers and subscribers, are created. Finally, to verify the integration of the environment, a setup must be performed from the terminal where the user is working. Then, it is necessary to check whether the topics, whose functionalities were described in Section 3.4.1., have been correctly registered (see Figure 4.4).

```

maria@maria-HP-Laptop-15-da0xxx:~/Documents/GitHub/ControlMotorROS/software/firmware/ControlMotorTF6_uROS$ source /opt/ros/foxy/setup.bash
maria@maria-HP-Laptop-15-da0xxx:~/Documents/GitHub/ControlMotorROS/software/firmware/ControlMotorTF6_uROS$ source /opt/microros_ws/install/setup.bash
maria@maria-HP-Laptop-15-da0xxx:~/Documents/GitHub/ControlMotorROS/software/firmware/ControlMotorTF6_uROS$ ros2 topic list
/motor1/enable
/motor1/get_pos
/motor1/set_pos
/motor2/enable
/motor2/get_pos
/motor2/set_pos
/parameter_events
/rosout

```

Figure 4.4: Application topic list.

From the *rqt* console provided by ROS2, which is a software framework that implements several tools in a graphical user interface [49], it is possible to subscribe to topics and set positions so that coordinated movements will be sent to both motors.

4.6. Functional application

Finally, once it has been verified that all the middleware modules work as expected independently and also together, the testing process is taken a step further. For this purpose, a structure where both motors can be mounted and connected to the ST board and the control board, as explained in the hardware section (see Figure 2.6), has been used. The platform used was part of a MSc thesis [50] and it has been adapted for its use in this project, as shown below:

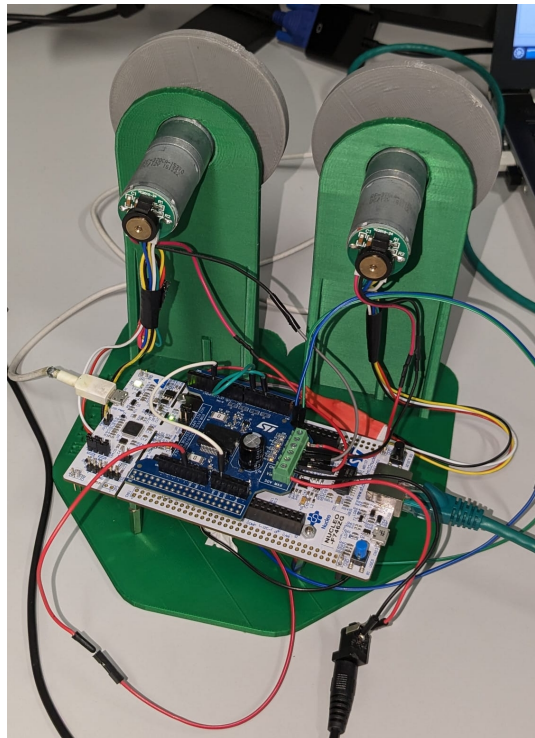


Figure 4.5: Platform for functional application.

Once the hardware part is ready, a simple Python application was developed to demonstrate that the project can be scalable in the future and can be implemented in much more complex control applications. The aim of this application was to establish one motor as the master and the other as the slave, so that if the master is moved by hand, the other motor will follow it making the same movement. A diagram of the application structure is shown in Figure 4.6.

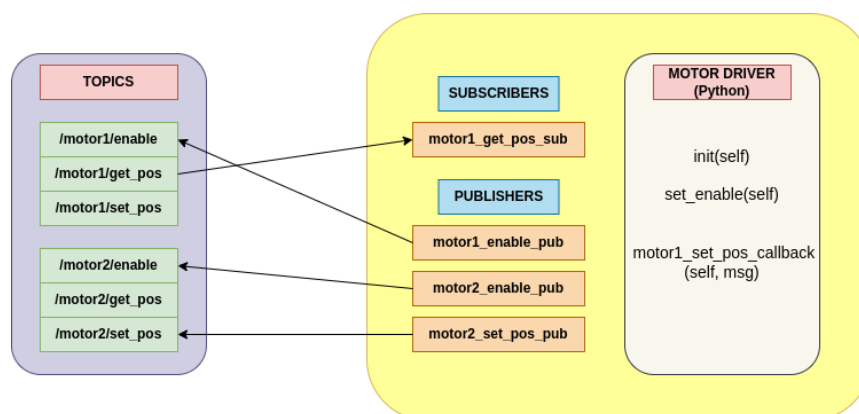


Figure 4.6: Python application diagram.

To achieve this, the motor to be moved manually shall be the one subscribed to obtain its position, and the slave shall be the one to publish this position through its

movement (see Figure 4.7). Only the PID controller of the motor acting as a slave will be enabled.

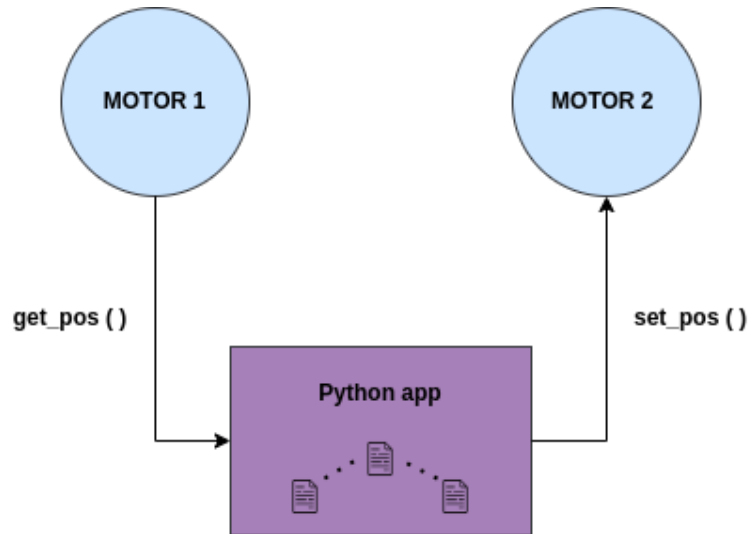


Figure 4.7: Application workflow.

For instance, if the master motor is moved half a revolution, the terminal where the application is running will show the new position plus its error, as shown in Figure 4.8:

```

[INFO] [1687287206.765174615] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287206.858927693] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287206.952810990] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287207.046603708] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287207.140107965] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287207.233960707] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287207.327637254] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287207.421543257] [motor_controller]: Position motor 1: 3.460846
[INFO] [1687287207.515160752] [motor_controller]: Position motor 1: 3.460846
  
```

Figure 4.8: Data received from the application.

Chapter 5

Conclusions

5.1. Conclusions

The aim of this BSc Thesis was to develop a standard software platform that would facilitate the design process of control systems. In order to do this, it was necessary to carry out an exhaustive analysis of the objectives that were expected to be achieved during the development time available. This required a prior study of the requirements and operation of the hardware and software implemented.

From a technical perspective and as a result of the decisions adopted during the project, which have had a direct impact on the results, it has been possible to develop and implement a fully functional motor control system based on microROS. The project has been approached with a particular focus on its reusability and scalability, trying to bring users and developers a solution that is user-friendly and adaptable to the possible changes that might be required in future configurations.

Regarding the implementation of ROS, it has been noticed both analytically and functionally that it provides multiple advantages, offering greater flexibility in terms of interoperability with other systems and ease of integrating different components. In addition to having a wide range of tools and libraries, ROS is an open source framework, which means that it allows code reuse. These improvements reduce development time, allowing several people to work simultaneously on the same project, whilst fostering the creation of more scalable control systems.

Personally, the execution of this project has given me the opportunity to get deeper into the field of control systems and to understand the great importance of their implementation in applications that are part of our daily lives, such as medical systems or plane controllers. It has also provided a broader vision of the development of a complete engineering project, from the first ideas to the resolution of problems before the project's culmination.

5.2. Future improvements

The aim of this BSc Thesis was to achieve a motor control system that works under ROS. During its development and subsequent testing process, several aspects for improvement have been detected that could be included in future versions of this project. Furthermore, as mentioned previously, embedded systems are part of a wide range of electronic systems and devices that belong to our daily lives, so the future lines of development are numerous. These improvements are listed below:

- Carry out an analysis of the motor's behaviour, subsequently developing its experimental modelling that will allow its configuration parameters to be adjusted, aiming to achieve an approximation to an ideal behaviour that guarantees its correct operation.
- Implement other types of more complex controllers, such as PID-D, which is a PID with a derivative on the parallel branch. This would allow, among other things, the elimination of step disturbances, achieving a much more stable and precise control system.
- In terms of developing possible applications, there is a wide range of possibilities. They could range from applications that control linear and rotational pendulums to applications that control game tables such as air hockey or a table football.
- Create an application, with its corresponding graphical interface, from which the system could be controlled in a user-friendly way. In addition, added functionalities could be included, such as graphing the behaviour of the motor in relation to how its parameters have been adjusted. This improvement would bring the project closer to replicating a virtual control laboratory.

Bibliography

References

- [1] Richard C. Dorf and Robert H. Bishop. *Modern control systems*. Pearson, Hoboken, thirteenth edition edition, 2016.
- [2] What is an Embedded System? Definition and FAQs | HEAVY.AI. URL: <https://www.heavy.ai/technical-glossary/embedded-systems>.
- [3] Insup Lee, Joseph Y.-T. Leung, and Sang H. Son. *Handbook of Real-Time and Embedded Systems*. CRC Press, July 2007.
- [4] Hassen Dorrah, Walaa Gabr, and Mohamed Elsayed. Derivation of Symbolic-based Embedded Feedback Control Stabilization Expressions with Experimentation. 2018:427–441, 12 2018. doi:10.1016/j.jesit.2018.02.003.
- [5] Ligo George. What is a Microcontroller ? How does it work ?, March 2020. URL: <https://electrosome.com/microcontroller/>.
- [6] Archil Avaliani. Quantum Computers, May 2004. arXiv:cs/0405004. URL: <http://arxiv.org/abs/cs/0405004>.
- [7] Everything You Need to Know About Microcontrollers | RS. URL: <https://uk.rs-online.com/web/content/discovery/ideas-and-advice/microcontrollers-guide>.
- [8] Carmine Noviello. *Mastering STM32*. Leanpub, 2018.
- [9] Myo Maung Maung, Maung Maung Latt, and Chaw Myat Nwe. DC Motor Angular Position Control using PID Controller with Friction Compensation. *International Journal of Scientific and Research Publications (IJSRP)*, 8(11), November 2018. URL: <http://www.ijsrp.org/research-paper-1118.php?rp=P837923>, doi:10.29322/IJSRP.8.11.2018.p8321.
- [10] IEEE Standards Association. URL: <https://standards.ieee.org0>.
- [11] MII and RMII Routing Guidelines for Ethernet, July 2019. URL: <https://resources.pcb.cadence.com/blog/2019-mii-and-rmii-routing-guidelines-for-ethernet>.

- [12] Essential Electronics - The H-bridge Motor Controller | Toshiba Electronic Devices & Storage Corporation | Europe(EMEA). URL: https://toshiba.semicon-storage.com/eu/semiconductor/design-development/innovationcentre/articles/tcm0587_TB67H450.html.
- [13] L6206 - DMOS dual full bridge driver - STMicroelectronics. URL: <https://www.st.com/en/motor-drivers/l6206.html>.
- [14] Walid Isaed. Close loop speed control of DC Motor with SCADA system by using arduino and labVIEW. 05 2018.
- [15] ROS: Home. URL: <https://www.ros.org/>.
- [16] M. A. B. Robotics. Legged robots: ROS2, March 2022. URL: <https://mab-robotics.medium.com/legged-robots-ros2-6051f9c907cd>.
- [17] ROS: The ROS Ecosystem. URL: <https://www.ros.org/blog/ecosystem/>.
- [18] Understanding nodes — ROS 2 Documentation: Foxy documentation. URL: <https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html>.
- [19] Getting Started with micro-ROS: Core and Advanced Tutorials – FIWARE, June 2020. Section: Tech. URL: <https://www.firmware.org/2020/06/16/getting-started-with-micro-ros-core-and-advanced-tutorials/>.
- [20] Introduction to Client Library, June 2023. URL: https://micro.ros.org/docs/concepts/client_library/introduction/.
- [21] micro-ROS on FreeRTOS, September 2020. URL: <https://www.freertos.org/2020/09/micro-ros-on-freertos.html>.
- [22] micro-ROS. URL: <http://www.ofera.eu/index.php/micro-ros>.
- [23] Hardware, November 2020. URL: <https://www.techopedia.com/definition/2210/hardware-hw>.
- [24] ST. *ARM®-based Cortex®-M7 32b MCU+FPU, 462DMIPS, up to 1MB Flash/320+16+ 4KB RAM, USB OTG HS/FS, ethernet, 18 TIMs, 3 ADCs, 25 com itf, cam & LCD-speed CAN transceiver*, 2016. Rev 4.
- [25] ST. *Dual brush DC motor driver expansion board based on L6206 for STM32 Nucleo*, 2015. Rev 1.
- [26] ST. *DMOS dual full bridge driver*, 2014. Rev 2.
- [27] Everything You Need To Know About DC Motors | RS. URL: <https://ie.rs-online.com/web/generalDisplay.html?id=ideas-and-advice/dc-motors-guide>.
- [28] How do brushed DC motors work? The need for regular maintenance explained | ASPINA. URL: <https://us.aspina-group.com/en/learning-zone/columns/what-is/013/>.

- [29] Mat Dirjish. What's The Difference Between Brush DC And Brushless DC Motors?, February 2012. URL: <https://www.electronicdesign.com/technologies/components/electromechanical/article/21796048/electronic-design-whats-the-difference-between-brush-dc-and-brushless-dc-motors>.
- [30] Pololu - 9.7:1 Metal Gearmotor 25Dx63L mm HP 12V with 48 CPR Encoder. URL: <https://www.pololu.com/product/4842>.
- [31] L.J. Osterweil. What is software? pages 261–273, 09 2008. doi:<https://doi.org/10.1007/s10515-008-0031-y>.
- [32] What is Software? Definition, Types and Examples. URL: <https://www.techtarget.com/searchapparchitecture/definition/software>.
- [33] Tarun Agarwal. PID Controller : Working, Types, Advantages & Its Applications, December 2013. URL: <https://www.elprocus.com/the-working-of-a-pid-controller/>.
- [34] Nikhil Agnihotri. What is FreeRTOS? URL: <https://www.engineersgarage.com/what-is-freertos/>.
- [35] Nikhil Agnihotri. What is Lightweight Internet Protocol (LwIP)? URL: <https://www.engineersgarage.com/light-weight-internet-protocol-arduino-esp-embedded-controllers/>.
- [36] What is Middleware? (And How Does it Work?). URL: <https://www.talend.com/resources/what-is-middleware/>.
- [37] ST. STM32L4 timers. URL: https://www.st.com/content/ccc/resource/training/technical/product_training/c4/1b/56/83/3a/a1/47/64/STM32L4_WDG_TIMERS_GPTIM.pdf/files/STM32L4_WDG_TIMERS_GPTIM.pdf/jcr:content/translations/en.STM32L4_WDG_TIMERS_GPTIM.pdf.
- [38] Getting Started with STM32 - Timers and Timer Interrupts. URL: <https://www.digikey.com/en/maker/projects/d08e6493cefa486fb1e79c43c0b08cc6>.
- [39] Félix Monasterio-Huelin, Álvaro Gutiérrez, and Blanca Larraga. Modelado de un motor DC, 2023. URL: <http://www.robolabo.etsit.upm.es/asignaturas/seco/apuntes/modelado.pdf>.
- [40] Blanca Larraga. Diseño, 2023. URL: <http://www.robolabo.etsit.upm.es/asignaturas/seco/transparencias/diseno.pdf>.
- [41] Félix Monasterio-Huelin, Álvaro Gutiérrez, and Blanca Larraga. Diseño, 2023. URL: <http://www.robolabo.etsit.upm.es/asignaturas/seco/apuntes/design.pdf>.
- [42] The Engineering Concepts. PID Controller - What-is-PID-controller-How-it-works?, November 2018. URL: <https://www.theengineeringconcepts.com/pid-controller/>.

- [43] Muniru Olajide Okelola, David Oluwagbemiga Aborisade, and Philip Adesola Adewuyi. Performance and Configuration Analysis of Tracking Time Anti-Windup PID Controllers. *Jurnal Ilmiah Teknik Elektro Komputer dan Informatika*, 6(2):20–29, January 2021. URL: <http://journal.uad.ac.id/index.php/JITEKI/article/view/18867>, doi:10.26555/jiteki.v6i2.18867.
- [44] Félix Monasterio-Huelin. Estudio del controlador PID., 2010. URL: http://www.robolabo.etsit.upm.es/asignaturas/seco/apuntes/2015-2019/controlador_pid.pdf.
- [45] Why RTOS and What is RTOS? URL: <https://www.freertos.org/about-RTOS.html>.
- [46] lwIP: Overview. URL: https://www.nongnu.org/lwip/2_1_x/index.html.
- [47] Alejandro Gómez Molina. Nomadrehab. URL: <https://github.com/Robolabo/NomadRehab/tree/main/Software>.
- [48] Reemplazando el firmware ST-Link por J-Link en las placas de desarrollo de ST | B105 lab, November 2017. URL: <https://elb105.com/reemplazando-el-firmware-st-link-por-j-link-en-las-placas-de-desarrollo-de-st/>.
- [49] rqt - ROS Wiki. URL: <http://wiki.ros.org/rqt>.
- [50] Alejandro Gómez Molina. Design and implementation of a high-performance hardware platform for driving motor control systems., 2022. URL: http://www.robolabo.etsit.upm.es/publications/TFM/TFM_AlejandroGomezMolina2.pdf.
- [51] María García Perote. Controlmotorros. URL: <https://github.com/Robolabo/ControlMotorROS.git>.

Appendix A

Ethical, social, economic and environmental aspects

A.1. Introduction

When developing an engineering project, it is of vital importance to take into consideration different ethical, social, economic and environmental aspects. This ensures that the regulations for the practical application of engineering are complied with and that engineers are aware of the impact that their projects can cause, as well as taking responsibility when it comes to mitigating the damage caused by them. Therefore, an analysis of the main impacts will be carried out.

A.2. Description of relevant impacts related to the project

A.2.1. Ethical impact

The pace of growth in digitisation and the implementation of robotic systems has gone from gradual to abrupt, raising social and ethical concerns. These doubts are based on users' mistrust of the security and efficiency of the developed systems. As explained, this project is intended to free up the workload of developers so that they can focus on improving the efficiency and performance of the systems. In addition, all the software developed is based on and supported by other open source resources, respecting intellectual property rights above all.

A.2.2. Social impact

As mentioned before, control systems form a very important part of our lives as they appear in a wide range of devices and applications, from washing machines to medical systems. This implies that the social impact of the project will be particularly relevant in this analysis.

This BSc Thesis focuses on the development of a standard software platform that facilitates the design of control systems for developers. The aim is to reduce the long development time involved in creating these systems so that developers can invest more time in improving their performance and efficiency, which is of high importance to contribute to the well-being and safety of people and to enhance user satisfaction. Furthermore, by integrating it with a ROS architecture, the developed middleware will be open source.

A.2.3. Economic impact

There is an increasing demand for more agile systems that are able to adapt to constant changes in the environment and to regulate the behaviour of other devices. In terms of economic impact, due to the far-reaching impact that robotics and digitalisation are having on all sectors, the subject of this project could be targeted for investment in the technology and industrial sector. Because it is mainly software development, the costs of the project are not limiting in terms of feasibility and scalability for future enhancements.

A.2.4. Environmental impact

The creation of electronic devices generally has a negative environmental impact, both in terms of energy consumption and greenhouse gas emissions. However, as this is a project that focuses on software development, the ecological impact that can be derived from it is minimal. In addition, the idea is that it is scalable, both at software as well as hardware level, and the electronic devices can be reused in future lines of development, thus reducing the carbon footprint.

Appendix B

Project budget

- **Personal:** To determine the hourly cost of employees, the average salary of a junior engineer (who could be a telecommunications or electronics engineer) and a project manager (who, in this case, is an engineer) has been taken into consideration. The estimations presented in Table B.1 are based on a gross annual salary of 28,000 € for the junior engineer and 38,000 € for the project manager, the social security costs paid by the company (which are around 35%) and the hours spent on the project. It has also been taken into account that, as the maximum working day in Spain is 40 hours per week, the average monthly working hours are 160 hours.

Table B.1: Human resources costs.

	Hourly rate (€)	Hours	Total (€)
Project manager	26.72	30	801.56
Engineering student	19.69	360	7,087.50
TOTAL			7,889.06

- **Costs of material resources:** For material resources, the price is calculated taking into account the depreciation, calculated as the price of the product divided by the months of useful life (see Table B.2).

Table B.2: Costs of materials.

	Lifetime (years)	Uds.	Cost (€)	Depreciation (€/month)	Time of use (months)	Total (€)
Personal computer	5	1	800.00	13.33	9	120.00
STM32F746ZG	6	2	53.33	0.74	9	6.66
X-NUCLEO-IHM04A1	6	2	31.60	0.44	9	3.95
Motor with encoder	8	2	50.00	1.04	6	6.25
Oscilloscope	7	1	2,000.00	23.81	3	71.43
Digital multimeter	7	1	50.00	0.60	2	1.19
Laboratory equipment	3	1	25.00	0.69	9	6.25
TOTAL						215.73

Finally, the project costs are summed up and the total cost of the project is calculated after taxes, with the addition of 21% IVA. Moreover, an estimation has been made to obtain a 20% profit from the project.

	Cost
Human resources costs	7,889.06 €
Material costs	215.73 €
Subtotal costs	8,104.79 €
Benefits	1,620.96 €
Subtotal	9,725.75 €
IVA	2,042.41 €
Total	11,768.16 €

Table B.3: Total cost.

Appendix C

User manual

This manual is dedicated to the user in order to facilitate the understanding of the steps to be followed to make use of the system.

C.1. Requirements

There are a number of software requirements that must be met in order to make use of the developed control system:

- Download the GitHub repository where the project is hosted [51].
- Linux Operating System (OS) on the computer, preferably Ubuntu 20.04, which is the version used for the development of the project.
- CubeIDE software installed, preferably version 1.10.1 to avoid bugs.
- Make a setup of the environment to support the use of ROS2 in the project.
- Python installed to run the example application that controls the system.

C.2. Install dependencies

The project repository is located on Github and can be downloaded using the following command at the terminal:

```
1 git clone https://github.com/Robolabo/ControlMotorROS.git
```

First, it is necessary to move to the micro-ROS folder of the downloaded project and once the *.sh* files are given execution permissions, run the *create_firmware.sh* script as follows:

```
1 cd ControlMotorROS/software/firmware/ControlMotorTFG_uros/Middlewares/  
   Third_Party/microROS  
2 chmod +x *.sh  
3 ./create_firmware.sh
```

Next, in the same folder there is an executable script, *setup_uros.sh*, that automates the download of ROS2, the agent used and the necessary dependencies. It is executed thus:

```
1 sudo -s source setup_uros.sh
```

The ROS distribution installed and with which all the configurations of the project have been made was foxy, but it could be changed to another one in case of using another version of Ubuntu. This configuration is hosted in the *setup_uros.sh* script.

```
1 export ROS_DISTRO=foxy
```

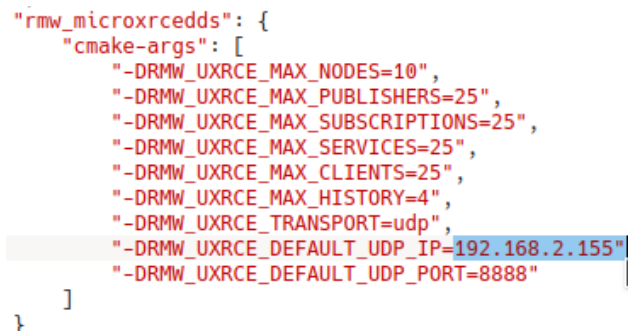
On each new terminal that is opened, it will be necessary to setup the ROS2 environment in order to be able to use its packages. The necessary commands are the following, but the script *setup_uros.sh* and *create_firmware.sh* have been modified so that they are configured and it is only necessary to execute the second command:

```
1 export ROS_DISTRO=foxy
2 source /opt/ros/foxy/setup.bash
3 source /opt/microros_ws/install/setup.bash
```

C.3. Starting XRCE-DDS agent

The first step to start the agent is to move to the micro-ROS folder and modify the *colcon.meta* files to configure the IP field (see Figure C.1). This is done in order to set a fixed IP of the computer to establish UDP communication when connecting the ST board. The IP set must correspond to the IP of the computer from which the agent is running.

```
1 cd ControlMotorROS/software/firmware/ControlMotorTFG_uros/Middlewares/
   Third_Party/microROS
```



```

"rmw_microxrcedds": {
  "cmake-args": [
    "-DRMW_UXRCE_MAX_NODES=10",
    "-DRMW_UXRCE_MAX_PUBLISHERS=25",
    "-DRMW_UXRCE_MAX_SUBSCRIPTIONS=25",
    "-DRMW_UXRCE_MAX_SERVICES=25",
    "-DRMW_UXRCE_MAX_CLIENTS=25",
    "-DRMW_UXRCE_MAX_HISTORY=4",
    "-DRMW_UXRCE_TRANSPORT=udp",
    "-DRMW_UXRCE_DEFAULT_UDP_IP=192.168.2.155",
    "-DRMW_UXRCE_DEFAULT_UDP_PORT=8888"
  ]
}

```

Figure C.1: Capture of the IP configuration that needs to be modified.

Alternatively, the following command can be run from the terminal:

```
1 ./configure_firmware.sh -t udp -i 192.168.2.155 -p 8888 #The project
   has been configured on that IP and on port 8888
```

After this step, it is necessary to run a command again to compile the ROS2 environment but with the computer's IP set this time.

```
1 ./build.sh -f -o ../
```

To start the communication with the board it is necessary to start the XRCE-DDS agent, which will act as a bridge between ROS2 and micro-ROS. The package that enables its launch is found with the dependencies downloaded from the repository. As the communication protocol used in the project is UDP to allow the ethernet connection, the agent is launched as follows on port 8888:

```
1 MicroXRCEAgent udp4 --port 8888 -v6
```

Once this is done, by executing the above command it is possible to see that the communication is done correctly by connecting the board and the ethernet cable (see Figure 4.3).

In addition, once the main program is running, you can test if the configured topics are registered correctly and if the name of the topic is known, you can listen directly to what it publishes from the terminal.

```
1 ros2 topic list #List registered topics
2 ros2 topic echo /topic_name #Command to listen the topic selected
```

In this project, the IP has been configured as static for simplicity but it could also work with DHCP by changing the LwIP settings to implement automatic IP address configuration.

C.4. Compiling the project and running the application in Python

The development of the project has been carried out using the CubeIDE software version 1.10.1, which must be downloaded from their website.

As shown in the pictures, it is necessary to import the project into the programme and then change the debug settings used. First, the downloaded project is opened (see Figure C.2) and imported into the CubeIDE environment (see Figure C.3).

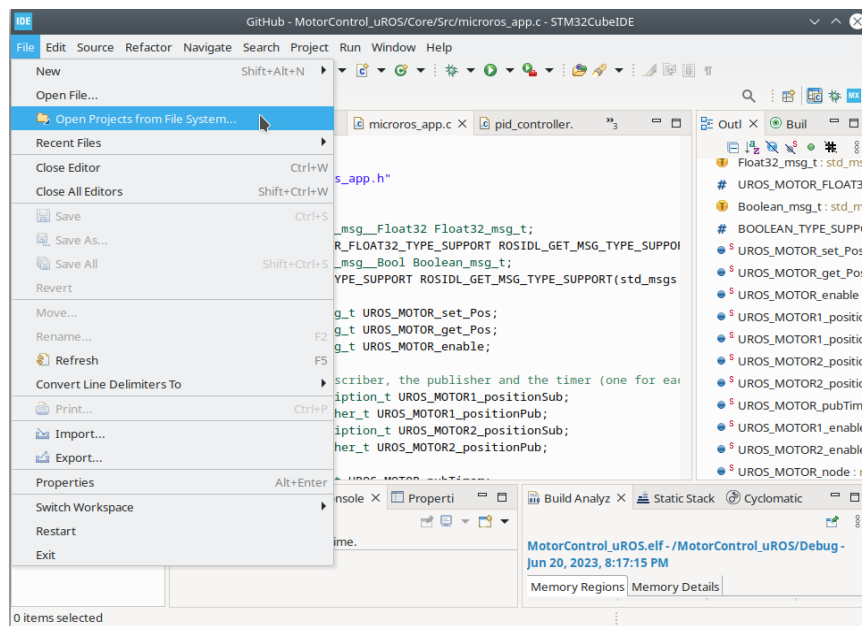


Figure C.2: Step 1: Open the downloaded project.

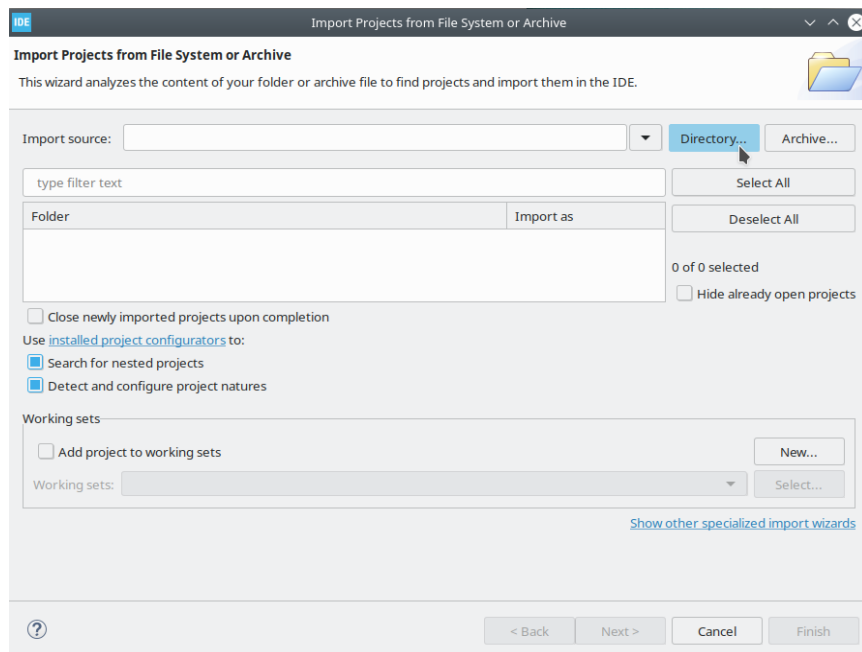


Figure C.3: Step 2: Import the project.

The debug probe configuration is then changed in the debug settings (see Figure C.4).

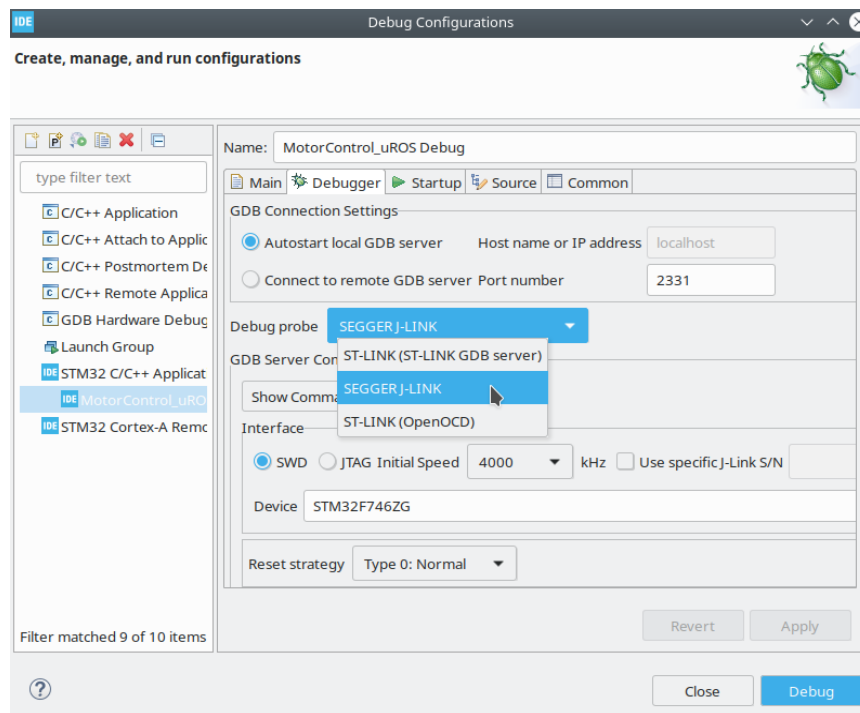


Figure C.4: Step 3: Change in debugger configuration.

As shown in Figure C.5, there are different options for compiling and debugging the project.

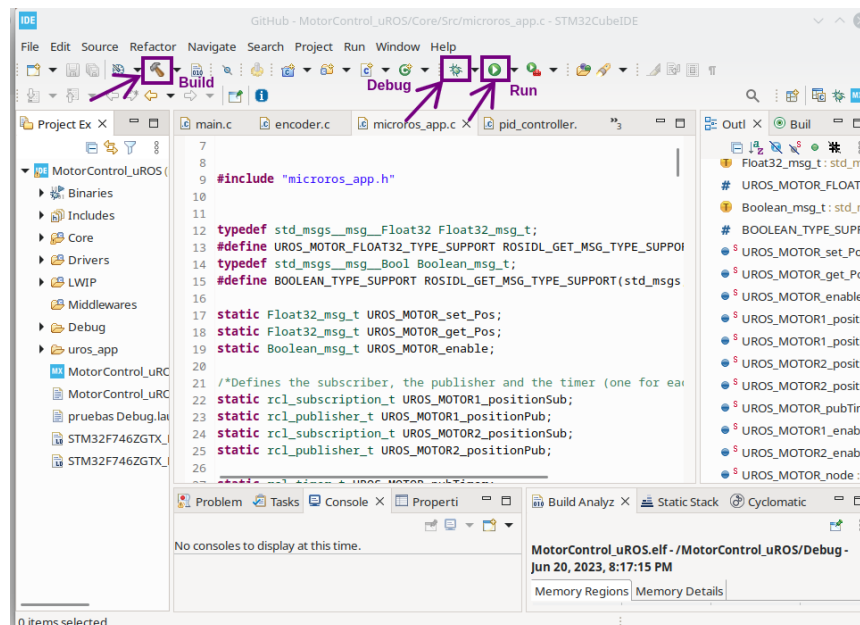


Figure C.5: Step 4: Compiling and debugging options.

From the *rqt* console (which, as mentioned before, is provided by ROS2 [49]) it is

possible to manually change the data sent to both motors, either the enable to allow activation of the PID controller or the position to which it shall move (see Figure C.6). The *rqt* console is launched with the following command:

```
1 rqt&
```

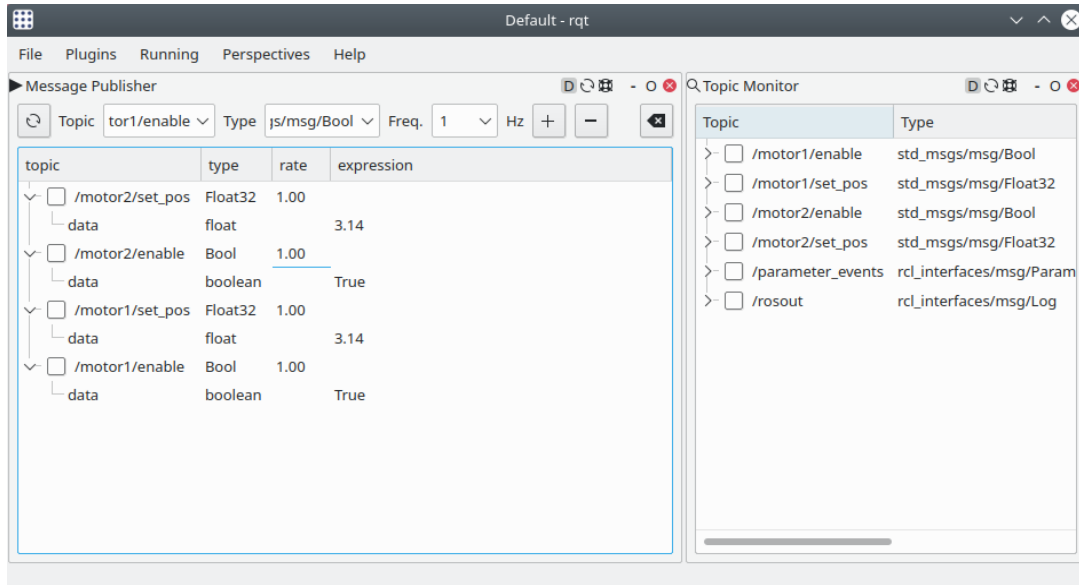


Figure C.6: Capture of the *rqt* console.

Finally, to run the developed python application, where one motor acts as master and the other as slave, the following command must be executed once the *python3* dependencies have been installed:

```
1 cd ControlMotorROS/software/firmware/ControlMotorTFG_uros
2 python3 uros_app/uos_app/uos_app.py
```

The screen display once it is executed can be seen in Figure 4.8.