

POLITECNICO DI TORINO

**Master of Science in Mechatronics  
Engineering**

Master Degree Thesis



**Motion control for an inflatable robotic arm using  
ROS**

**Supervisor**

Prof. Stefano Mauro  
Pierpaolo Palmieri  
Matteo Gaidano

**Candidate**

Matteo Barisione

Academic year 2021-2022





## Abstract

Nowadays, soft robotics is becoming increasingly significant, it is a subfield of robotics that focuses on the development of devices made of light, flexible materials that perform better when interacting with humans and their surroundings. Soft robots have a considerable range of applications such as surgery, prosthetics, pain management, and space exploration. Soft robots are suitable in space applications, where they may perform and provide considerable benefits.

There are different categories of soft robots including inflatables, which thanks to their lightweight nature allows them to be easily contained in a small package and deployed when required. This characteristic indirectly lowers the cost of launching the manipulator into orbit because it takes up less space and weighs less than a rigid robot.

Due to their flexible elements, certain type of soft robots can only exert a limited amount of force. Furthermore, under stress, they might exhibit deformations that are difficult to compensate. As a result of their highly non-linear dynamics, rigid robotics control cannot be successfully applied to soft robots. Furthermore, in the case of the aerospace industry application, it is necessary to carry out an in-depth study of the materials used for the construction of the manipulator in order to make it resistant to hostile environments such as space.

The goal of this thesis is to control a three-degree-of-freedom robot using the robotic software ROS 1.0. The robot to be controlled, the POPUP robot, is a soft manipulator developed for space applications in the laboratories of the Politecnico di Torino, it is made up of two inflatable links and three rigid joints.

The robotic software ROS 1.0 used to control the manipulator is an open-source robot meta-operating system. It offers several services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message passing between processes, and package management. It also includes tools and libraries for obtaining, constructing, writing, and running code across multiple computers. Using this library tool, it was feasible to create complete robot control, allowing for the integration of other devices and the creation of a modular framework.

The application and the entire system have been configured and designed to be versatile and modular, allowing for easy integration both in terms of hardware, with the addition of sensors and motors on the manipulator, and software with the use of tools for simulating and monitoring the POPUP robot.



# Acknowledgements

I would like to thank my supervisors prof. Stefano Mauro, Pierpaolo Palmieri and Matteo Gaidano for accepting me into their team and for all of their assistance and advise during my dissertation.

My deepest gratitude go to all of my family members for their encouragement and support throughout my studies, without which this would not have been possible. Lastly, I would like to thank all my friends who supported me during this journey, especially I want to thank Margherita, Marco, Giovanni and Luca for being not only classmates but also great friends over these years.

*Sincerely, thank you  
Matteo*



# Contents

<b>List of Tables</b>	VIII
<b>List of Figures</b>	IX
<b>1 Introduction</b>	1
1.1 Description POPUP robot . . . . .	1
<b>2 Robotic framework</b>	5
2.1 Introduction . . . . .	5
2.2 Robotic framework comparison . . . . .	7
2.3 ROS 1.0 vs ROS 2.0 vs micro-ROS . . . . .	9
2.4 ROS1 vs ROS2 performance evaluation . . . . .	13
2.4.1 Scheduling latency . . . . .	14
2.4.2 Task periodicity . . . . .	14
2.4.3 Message loss . . . . .	16
2.5 Final evaluation . . . . .	17
<b>3 ROS 1.0</b>	19
3.1 History . . . . .	19
3.2 Distribution . . . . .	19
3.3 What is ROS (Robot Operating System) . . . . .	19
3.4 Ros basic functionalities . . . . .	20
3.4.1 Communication between nodes . . . . .	21
3.4.2 ROS package . . . . .	23
3.4.3 IDE - Visual Studio Code . . . . .	24
3.4.4 Visualization . . . . .	24
3.4.5 Simulation . . . . .	25
3.5 Rosserial . . . . .	26
3.5.1 Rosserial package . . . . .	26
3.5.2 Rosserial protocol . . . . .	27

<b>4</b>	<b>Popup robot devices</b>	<b>29</b>
4.1	Actuator AK80-80 Robotic Actuator . . . . .	31
4.1.1	Controller Area Network (CAN) . . . . .	33
4.1.2	Actuator - PID schematic . . . . .	35
4.2	Microcontroller STM32H7 . . . . .	36
4.3	PC/Raspebrry Pi4 . . . . .	38
<b>5</b>	<b>STM32H7 configuration</b>	<b>39</b>
5.1	UART3 configuration . . . . .	39
5.1.1	UART: Hardware Communication Protocol . . . . .	40
5.2	FDCAN configuration . . . . .	43
5.3	Timer configuration . . . . .	43
5.4	Rosserial configuration . . . . .	45
5.4.1	STM32H7 . . . . .	45
5.4.2	PC . . . . .	46
5.5	Basic publisher/subscriber on STM32H7 . . . . .	48
<b>6</b>	<b>STM32H7 firmware</b>	<b>51</b>
6.1	main.c . . . . .	52
6.1.1	Function description . . . . .	53
6.2	mainpp.cpp . . . . .	55
6.2.1	Publisher/subscriber functions . . . . .	57
<b>7</b>	<b>ROS workspace</b>	<b>61</b>
7.0.1	Basic publisher/subscriber functions on ROS . . . . .	63
<b>8</b>	<b>ROS node development</b>	<b>65</b>
8.1	Position controller . . . . .	65
8.2	One motor control - ROS node . . . . .	66
8.2.1	Terminal Interface . . . . .	67
8.2.2	Graphical Interface . . . . .	67
8.3	Continous communication . . . . .	68
8.3.1	GUI motor control . . . . .	72
8.4	Three motor control - ROS node . . . . .	74
<b>9</b>	<b>Message latency comparison</b>	<b>77</b>
9.1	One motor at time - (single mode) . . . . .	77
9.1.1	Time analysis - script python . . . . .	79
9.2	Three motor at time - (matrix mode) . . . . .	81
9.2.1	Time analysis - script python . . . . .	83

<b>10 Popup speed control</b>	<b>87</b>
10.1 Gamepad control . . . . .	89
10.2 GUI - speed control . . . . .	90
<b>11 PID firmware</b>	<b>93</b>
11.1 PID funciton . . . . .	93
11.2 PID in C . . . . .	94
11.3 PID tuner . . . . .	98
<b>12 Conclusion</b>	<b>103</b>

# List of Tables

1.1	Robot links dimensions and operating pressure . . . . .	3
2.1	ROS 1.0 and ROS 2.0 comparison . . . . .	10
11.1	PID parameters . . . . .	98



# List of Figures

1.1	POPUP robot . . . . .	1
1.2	POPUP robot deflated . . . . .	2
1.3	Link exploded . . . . .	3
1.4	POPUP robot - Actuator AK80-80 . . . . .	3
1.5	Microcontroller STM32H743 . . . . .	4
2.1	Middleware . . . . .	6
2.3	Frameworks/middleware comparison . . . . .	7
2.2	Robotic framework and middleware comparison . . . . .	8
2.4	Middleware structure ROS 1.0 and ROS 2.0 . . . . .	12
2.5	Middleware structure micro-ROS . . . . .	13
2.6	Software stack architerture of ROS 1.0 and ROS 2.0 . . . . .	14
2.7	ROS 1.0 and ROS 2.0 latency comparison . . . . .	14
2.8	Timeline of a ROS node . . . . .	15
2.9	Periodicity of ROS node . . . . .	15
2.10	Table node periodicity . . . . .	16
2.11	Scheme of the environment used to test the ROS node communication . . . . .	16
2.12	Maximum communication latency on varying data size . . . . .	17
3.1	Message communication . . . . .	20
3.2	Topic message communication . . . . .	22
3.3	Message communication between nodes . . . . .	23
3.4	ROS's package folder . . . . .	24
3.5	VS Code: ROS node development environment . . . . .	24
3.6	ROS Visualization (RViz) . . . . .	25
3.7	Gazebo ROS simulator . . . . .	25
3.8	roserial server (for PC) and client (for embedded system) . . . . .	26
3.9	Rosserial frame format . . . . .	27
4.1	Robot ecosystem diagram . . . . .	29
4.2	POPUP devices connection . . . . .	30

4.3	Structural design actuator AK80-80 . . . . .	31
4.4	Motor's abbreviation . . . . .	31
4.5	Motor AK80-80 USB configuration . . . . .	32
4.6	CAN data frame . . . . .	34
4.7	CAN data frame . . . . .	34
4.8	Data frame - Actuator AK80-80 . . . . .	35
4.9	Schematic of the PID controller inside the actuator's driver . . . . .	36
4.10	Microcontroller STM32H7 . . . . .	36
4.11	Raspberry Pi4 or PC - POPUP controller . . . . .	38
5.1	UART packet . . . . .	40
5.2	USART3 & DMA - IDE configuration . . . . .	41
5.3	Example IDE configuration . . . . .	42
5.4	FDCAN - IDE configuration . . . . .	43
5.5	TIM2 & TIM8 - IDE configuration . . . . .	44
5.6	STM32H7 firmware file . . . . .	45
5.7	ROS distribution . . . . .	47
5.8	Terminal output <code>ros</code> command . . . . .	49
6.1	Flow chart main.c . . . . .	52
6.2	The following figure summarises all the topics and message types that have been implemented to control the robot . . . . .	57
7.1	<b>controller popup</b> package's folder . . . . .	62
7.2	<b>roserial python</b> package's folder . . . . .	62
7.3	Terminal output <code>\$ roserial_python</code> command . . . . .	62
8.1	Graph - $k_p$ value . . . . .	66
8.2	Motor control - terminal interface . . . . .	67
8.3	Motor control - Graphical interface . . . . .	69
8.4	Flow chart ROS node . . . . .	70
8.5	GUI - Three motor position controller . . . . .	73
8.6	Scheme communication - one motor . . . . .	73
8.7	Scheme communication - three motor . . . . .	75
9.1	Scheme measured transmission time - single mode . . . . .	78
9.2	Terminal output - ROS node time analysis . . . . .	79
9.3	Time response - one motor at time . . . . .	80
9.4	Histogram time response - one motor at time . . . . .	81
9.5	Scheme measured transmission time - matrix mode . . . . .	82
9.6	Time response - three motor at time . . . . .	84
9.7	Histogram time response - three motor at time . . . . .	85

10.1	POPUP robot - speed controller . . . . .	87
10.2	Gamepad speed control command . . . . .	90
11.1	PID scheme . . . . .	94
11.2	Complete firmware's flow chart . . . . .	97
11.3	GUI - PID tuner . . . . .	99
11.4	Movement POPUP robot . . . . .	100
11.5	Motors feedback - position . . . . .	100
11.6	Motors feedback - speed . . . . .	101
11.7	Motors feedback - current . . . . .	101



# Chapter 1

## Introduction

### 1.1 Description POPUP robot

The object examined in this project is a robotic manipulator called POPUP robot, developed by engineer Pierpaolo Palmieri as part of his PhD. The purpose of this project is to control the robot entirely through software. The manipulator is made up of three 3D printed rigid joints which contain the three AK80-80 actuators giving the manipulator 3 DOF, and two inflatable links made-up of PVC material. Due to the inflatable links, the manipulator is now classified as a soft robot.

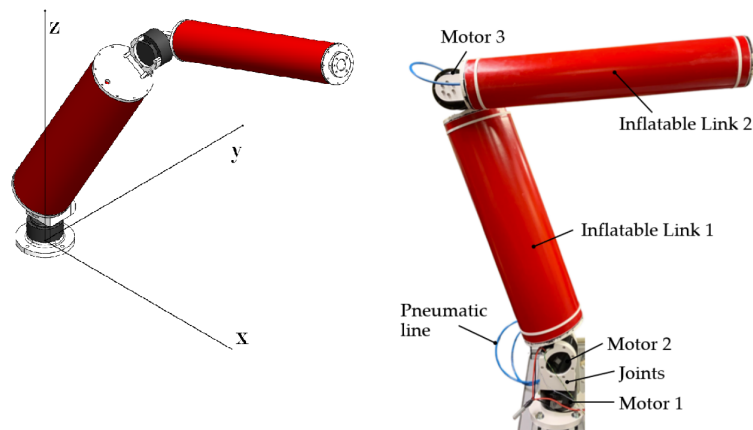


Figure 1.1: POPUP robot

**Soft robotics** is a subfield of robotics that concerns the design, control, and fabrication of robots composed of compliant materials, instead of rigid links. In contrast to rigid-bodied robots built from metals, ceramics and hard plastics, the

compliance of soft robots can improve their safety when working in close contact with humans.<sup>1</sup>

An advantage of inflatable links is the lightness of the robot itself since the links are filled with air. Another positive aspect of inflatable links is the possibility of occupying a very small space when they are deflated, as shown in the figure 1.2. The robot with these features can be used in various applications such as aerospace, collaborative robotics and biomedicine. For example the robot can be used in an aerospace environment due to its light weight allowing easier transportation, and the softness of the links favours application in collaborative robotics, reducing damage caused in the event of a collision.

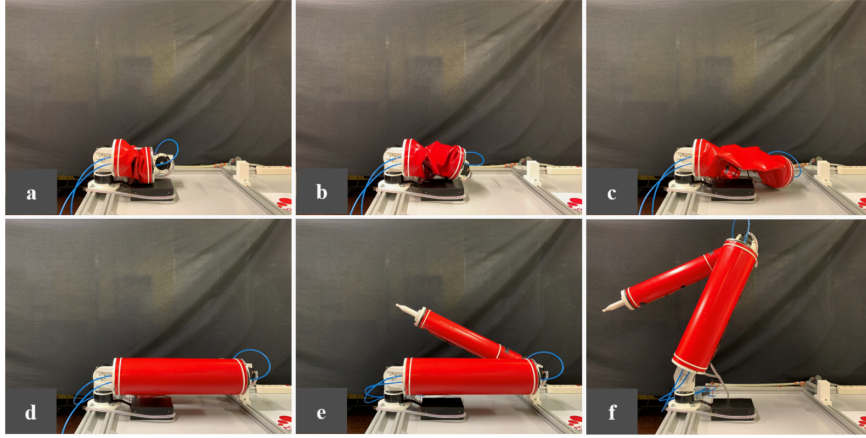


Figure 1.2: POPUP robot deflated

The structure of the POPUP robot shown in the figure below can be divided into several parts:

- **links:** generally, links are defined as follows *links are the rigid members that connect the joints*, in this case links connect the joints, but cannot be considered rigid, since when the robot is operational there are only air inside them. The inflatable links have cylindrical shape and are made out PVC fixed to a 3D printed support which connect the link to the actuator, that allow the robot to move.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Soft\\_robotics](https://en.wikipedia.org/wiki/Soft_robotics)



Figure 1.3: Link exploded

In order for the robot to be able to make movements, the air pressure inside the links must be about 1 bar. The links are inflated via a pneumatic line, and the pressure inside is controlled via a pressure gauge at the base of the two links.

Dimension	Link 1	Link 2
Length (mm)	600	600
Radius (mm)	85	55
Pressure (kPa)	30	30

Table 1.1: Robot links dimensions and operating pressure

- **Actuators:** Three motors (ID: 1,2,3) were chosen to allow the robot to move. The actuator consists of a BLDC motor directly connected to a high-precision planetary gearbox with a reduction ratio of 80:1; the model of the actuator is: AK80-80 Robotic Actuator.



Figure 1.4: POPUP robot - Actuator AK80-80

The AK series module is a high-end actuator with a brushless DC motor, integrated planetary gear, encoder and driver. It can be widely used in exoskeletons, walking robots, automation equipment, scientific research and education and other fields. The driver adopts a field-oriented control algorithm (FOC), and cooperates with a high-precision angle sensor to achieve precise position and torque control. The design of 36N42P, rare earth magnets and

high-precision planetary gears can make the motor more stable and more torque. AK80 series motors support a variety of communication protocols, and provide a human-machine interface through communication with PC, allowing users to control the motor faster and more accurately.

It also integrates the inverter and the controller that receives and transmits data and commands via the CAN communication protocol.

- **Microcontroller STM32H7:** Robot movements are controlled by a microcontroller that communicates with the motors via CAN. The microcontroller is used both to communicate with the motors and to communicate with the PC. The high performance microcontroller STM32H743 was chosen to complete this tasks.

This MCU represents the highest-performance ARM Cortex-M core implementation on the market today. The STM32H7 microcontrollers are ideal for industrial gateways, home automation, telecommunications equipment and smart consumer products, as well as high-performance motor controls, home appliances and small devices with complex user interfaces.

The STM32H743 chip integrates 35 communication peripherals that support protocols and standards such as CAN FD, SD CARD (4.1), SDIO (4.0) and MMC (5.0). There are also 11 advanced analogue functions including low-power 2Msamples/s 14-bit ADC, 12-bit DAC, op-amp, as well as 22 timers, including a high-resolution timer running at 400MHz.

Therefore, the following board was chosen as the robot's main controller because it guarantees high performance and reliability, and finally, it was configured to communicate via the CAN protocol thus enabling the sending and receiving of data from the motors.

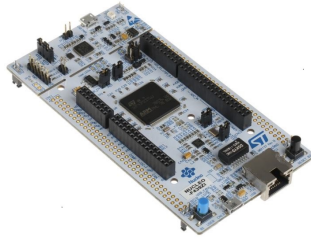


Figure 1.5: Microcontroller STM32H743

- **Pneumatic line:** allows the links to be inflated by switching on a compressor, thus enabling the robot to be used. Furthermore, the advantage is that once the pressure inside the links has been reached, it is no longer necessary to supply air to the links.



## Chapter 2

# Robotic framework

### 2.1 Introduction

As described earlier, the POPUP robot is composed of three motors which are operated by a microcontroller. Initially the board was in charge of managing all the movements of the robot, so within the firmware there were all the functions for the complete control of the robot. Functions like position control, speed control, inverse kinematic, and the functions for sending and receiving commands via CAN protocol, were present within the firmware and were called up periodically to move the robot.

This solution was not so flexible, as each time a new movement or trajectory had to be executed, the robot had to be stopped and the firmware on the board had to be reloaded with the new trajectory.

In order to have a more dynamic solution and thus to have more flexibility in the control of the robot, obtaining the possibility of executing different trajectories, it was necessary to modify the firmware and add new devices and software to control the robot.

The paper listed in the bibliography suggested the use of a middleware to control a robot:

**Middleware** *is a type of computer software that provides services to software applications beyond those available from the operating system. It can be described as "software glue"*<sup>1</sup>

---

<sup>1</sup><https://it.wikipedia.org/wiki/Middleware>

Starting from this paper [1], an in-depth research was carried out to find out which robotic middleware, was the best to use in order to meet the requirements listed above, i.e. the possibility of executing different trajectories, creating a modular environment allowing easier integration of sensors on the robot, and ensuring great flexibility of the whole system.

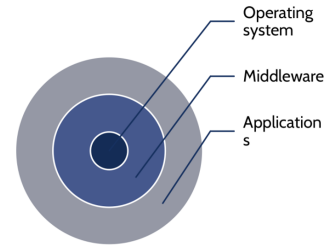


Figure 2.1: Middleware

Robotic systems require a high degree of specialisation and large number of different scientific fields are involved. Software is therefore needed to enable easy integration between the different modules, allowing the robotic system to work fluently. For this reason, there are a large variety of frameworks and middleware that offer different methods for integrating the different modules.

As described in [1], since robots are becoming complex systems with more sensors and elements to control, a single software is no longer sufficient to perform this task, so frameworks are used. A robotic framework is a collection of software tools, libraries and conventions, aiming at simplifying the task of developing software for a complex robotic device.

Instead the Robotic middleware is a type of computer software that provides services to software applications beyond those available from the operating system.

After determining what was required for the project, an examination and comparison of the various robotic frameworks and middleware on the market was carried out. In [2] are listed as the most used robot software platforms:

- MSRDS<sup>2</sup> Microsoft Robotics Developer Studio, Microsoft - U.S.
- ROS Robot Operating System, Open Robotics<sup>12</sup> - U.S.
- OROCOS Europe
- other middleware are MRPT, CARMEN, LCM, Player, Microsoft RDS

Most of the time, the negative aspect of the robotics software are: language support limitations, unoptimized communication between processes, lack of support for various devices which is arguably the hardest problem to fix.

---

<sup>2</sup><https://www.microsoft.com/robotics/>

## 2.2 Robotic framework comparison

Several papers listed in the bibliography were read comparing the various middleware on the market. The choice of middleware was guided by the following parameters:

- **Open source:** the middleware must be open source so all material can be found online without buying software.
- **Practical to use:** middleware must make it possible to develop software easily, even without the use of advanced knowledge; allowing whoever to learn.
- **Flexibility:** the middleware sought must guarantee that the system can be easily modified, while also providing flexibility in modifications.
- **Packages and lessons available:** Another important parameter that was taken into account when choosing the middleware is the presence/availability of guides and tutorials.
- **no advanced systems required:** The middleware chosen must not required the use as super computers to run. It must in fact be possible to run the robotics software on commercial devices. In this way, the flexibility and modularity of the entire system is maintained.

The tables below compares the various middleware according to their main characteristics; a more in-depth description of the various middleware can be found in the following papers [1]:

Robotic Frameworks	Programming Language	OpenSource	Distributed Architecture	Fleet Management	Robotic Cloud	Resilience	Dynamic Scheduling	Simulation	Robot/Robot Communication
ROS	Python, C++	✓	✓	✓	×	×	×	✓	×
Player	C++, Tcl, Java, Python	✓	×	✓	×	×	✓	×	×
ARIA	C++, Python, Java	✓	×	✓	×	×	×	×	×
ASEBA	Aseba	✓	✓	✓	×	×	×	×	×
Carmen	C++	✓	✓	✓	×	×	×	✓	×
OpenRDK	C++	✓	✓	×	×	×	×	×	×
Orca	C++	✓	✓	×	×	×	×	×	×
Orocos	C++	✓	✓	✓	×	×	×	×	×
Urbi	C++ like	✓	×	✓	×	×	×	×	×
TalkRoBots	Python	✓	✓	✓	✓	✓	✓	✓	✓

Figure 2.3: Frameworks/middleware comparison

RFWs	OS	Programming language	Open source	Distributed architecture	HW interfaces and drivers	Robotic algorithms	Simulation	Control / Realtime oriented
ROS	Unix	C++, Python, Lisp	✓	✓	✓	✓	~	✗
HOP	Unix, Windows	Scheme, Javascript	✓	✓	~	✗	✗	✗
Player/Stage/Gazebo	Linux, Solaris, BSD	C++, Tcl, Java, Python	✓	~	✓	✓	✓	✗
MSRS (MRDS)	Windows	C#	✗	✓	~	✗	✓	✗
ARIA	Linux, Win	C++, Python, Java	✓	✗	✓	✓	✗	✗
Aseba	Linux	Aseba	✓	✓	✓	✗	~	✓
Carmen	Linux	C++	✓	✓	✓	✓	✓	✗
CLARAty	Unix	C++	✓	✓	✓	✓	✗	✗
CoolBOT	Linux, Win	C++	✓	✓	~	✗	✗	✗
ERSP	Linux, Win	?	✗	✓	✓	✓	✗	✗
iRobot Aware	?	?	✗	?	✓	?	✗	?
Marie	Linux	C++	✓	✓	✓	✗	✗	✗
MCA2	Linux, Win32, OS/X	C, C++	✓	✓	✓	✗	✗	✓
Miro	Linux	C++	✓	✓	✓	✗	✗	✗
MissionLab	Linux, Fedora	C++	✓	✓	✓	✓	✓	✗
MOOS	Windows, Linux, OS/X	C++	✓	~	✓	✓	✗	✗
OpenRAVE	Linux, Win	C++, Python	✓	✗	✗	✓	✓	✗
OpenRDK	Linux, OS/X	C++	✓	✓	✓	✗	✗	✗
OPRoS	Linux, Win	C++	✓	✓	✓	✓	✓	✗
Orca	Linux, Win, QNX Neutrino	C++	✓	✓	✓	~	✗	✗
Orocos	Linux, OS/X	C++	✓	✓	✓	✓	✗	✓
RoboFrame	Linux, BSD, Win	C++	?	✓	✓	✗	✗	✗
RT middleware	Linux, Win, CORBA platform	C++, Java, Python, Erlang	✓	✓	✓	✗	✗	✗
Pyro	Linux, Win, OS/X	Python	✓	✗	✓	✓	✓	✗
ROCI	Win	C#	✓	✓	✗	✗	✗	✗
RSCA	?	?	✗	✗	✓	✗	✗	✓
ROCK	Linux	C++	✓	?	✓	✓	✗	✓
SmartSoft	Linux	C++	✓	✓	✗	✗	✗	✗
TeamBots	Linux, Win	Java	✓	✗	✓	✓	✓	✗
Urbi (language)	Linux, OS/X, Win	C++ like	✓	✗	✓	✗	✗	✗
Webots	Win, Linux, OS/X	C, C++, Java, Python, Matlab, Urbi	✗	✗	✓	✗	✓	✗
YARP	Win, Linux, OS/X	C++	✓	✓	✓	✗	✓	✗

Figure 2.2: Robotic framework and middleware comparison

After a careful analysis of the different middleware, made by reading the following papers[3] [4] [5], , it was concluded that ROS is the most advantageous middleware to use. As also reported in [6] ROS has several advantages including:

- **High-end capabilities:** ROS presents many ready-to-use features within its packages, enabling faster software development. For example, packages for

SLAM are present in Ros, as well as packages for AMCL and control. Furthermore, these packages have the possibility of changing the main parameters so that they can be used for different applications.

- **A great variety of tools:** There are simulation and debugging tools such as Rviz, Gazebo, MoveIT
- **Inter-platform operability:** The ROS message-passing middleware allows communication between different programs. In ROS, this middleware is known as nodes. These nodes can be programmed in any language that has ROS client libraries. We can write high-performance nodes in C++ or C and other nodes in Python or Java.
- **Modularity:** Ros allows modular software to be created through the use of nodes, thus guaranteeing two advantages over middleware on the market: in the event of a node crash, the internal software can continue to function, and it also allows greater flexibility by allowing nodes developed by outsiders to be integrated.
- **Active community:** Finally, over the years a large online community has emerged that has enabled and improved the middleware, creating new packages and resources for the beginners

Similar advantages are also reported in the books [2] [6].

## 2.3 ROS 1.0 vs ROS 2.0 vs micro-ROS

The **Robot Operating System (ROS)** is a set of software libraries and tools that help build robot applications. ROS provides functionality for hardware abstraction, device drivers, communication between processes over multiple machines, tools for testing and visualization, and much more.

There are different types of ROS, in fact there is **ROS 1.0**, **ROS 2.0** and **micro-ROS**. Originally developed by Willow Garage in 2007, ROS1 has become a household name in the open source robotics community.

The team behind ROS1 has years of experience in understanding which key features are missing and which can be improved. Unfortunately, adding all these modules to ROS1 required a lot of breaking changes and made ROS1 very unstable. So ROS2 was redesigned from the ground up and is a brand new ROS.

At present, ROS is not very popular in the industry, and it lacks some important requirements such as real-time, security, authentication, and security. One of

the goals of ROS2 is to make it compatible with industrial applications. In the following part of the paragraph, the two middlewares were compared, evaluating their performance and characteristics.

The main differences between ROS 1.0 and ROS 2.0 are listed in the table 2.1.

ROS 1.0 (since 2007)	ROS 2.0 (released 2017)
Provide the software tools for users who need to R&D projects with the PR2 also designed ROS to be useful on other robots.	Provide new features: robot Security, real-time control, increase distributed processing
C++ 03; C++ 11; Python 2	C++11; C++14; C++17; Python 3.5
Ubuntu OS X	Ubuntu Xenial    Windows 10    OS X
Linux    Mac	Linux    Windows    MAC    RTOs
Not possible to create more than one node in a process	Possible to create multiple nodes in a process
The powerful community has accumulated rich and stable packages, debugging tools and a complete tutorial help beginners to quickly understand the use cases and methods of ROS	<ul style="list-style-type: none"> <li>- Minimal dependencies</li> <li>- Better portability</li> <li>- Greater reliability and persistence</li> <li>- Strong real-time</li> </ul>
<ul style="list-style-type: none"> <li>- Beginner/Students</li> <li>- General developer</li> <li>- Robotics Algorithm Developer</li> </ul>	<ul style="list-style-type: none"> <li>- Computer professional learners</li> <li>- Developer who value real-time</li> </ul>

Table 2.1: ROS 1.0 and ROS 2.0 comparison

Starting from the following papers [7] [8], an analysis will be performed on the different characteristics of ROS 1.0 and ROS 2.0:

## ROS 1.0

The most used and widely spread mobile robotics middleware. Has a large active community and a wide support for various sensors, actuators and mobile robots. Many companies support ROS by providing and maintaining interfaces to their

products. Modular design with separate execution and data management, encapsulated into nodes. Communication is performed using message passing over TCP (nodes) or shared memory (nodelets). The system provides both a **publisher/-subscriber** paradigm for asynchronous communication, as well as a **client/server** paradigm to enable synchronized execution on desired input data.

A central server maintains discovery and connection handling between nodes, while the actual node connections are implemented in a peer-to-peer fashion. ROS supports a large variety of programming languages, and includes both the communication interface infrastructure, as well as a software packaging, building, and release toolchains.

## ROS 2.0

The successor of ROS, where the key difference lies in the communication. ROS utilize a custom designed communication via TCP/IP. ROS2 instead utilize DDS (Data Distribution Service), which is a standardized way that has been used in a plethora of different systems and applications. DDS is the mechanism that handles the discovery between nodes in ROS2, which allows for a fully decentralized system. In ROS the mechanism of handling node communication is centralized and takes place through the roscore software which acts as a master where nodes can ask the master to discover other nodes for them. In ROS2 this discovery mechanism is instead embedded in each node (via DDS), and there is no master used in ROS2. This is, for example, helpful in multi-robot setups. Apart from these underlying changes, the overall functionality of ROS and ROS2 is similar. To transfer from ROS to ROS2 requires that the underlying code that builds up the nodes has to be modified. Overall, ROS2 is gaining momentum but many packages are still only available for ROS.

The Robot Operating System (ROS), open-source middleware, has been widely used for robotics applications. However, the ROS is not suitable for real-time embedded systems because it does not satisfy real-time requirements and only runs on a few OSs.

This represents one of the main drawbacks of ROS1, so a version has been developed to solve these problems, ROS 2. ROS 2 uses DDS, DDS is middleware for data-centric systems (from the Anglo-Saxon data-centric systems), i.e., distributed systems whose operation is based on the exchange of real-time data from multiple sources to multiple destinations.

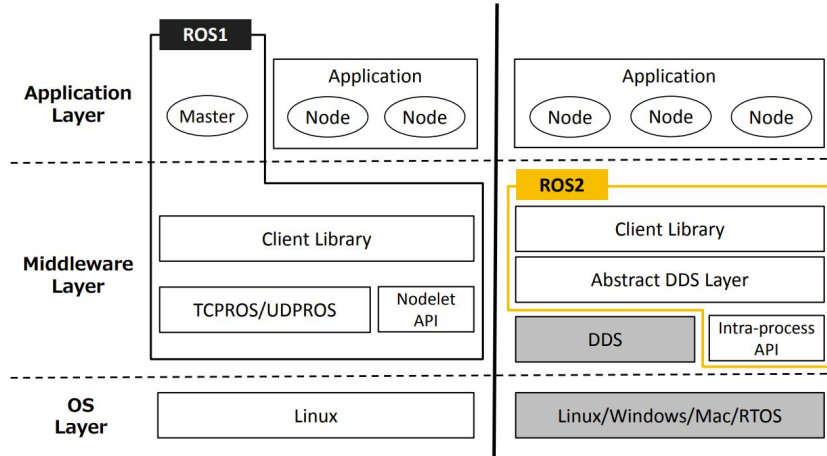


Figure 2.4: Middleware structure ROS 1.0 and ROS 2.0

Another type of ROS has emerged in recent years and is micro-ROS, which was created to bridge the gap between resource-limited microcontrollers and larger processors in robotics applications based on the Robot Operating System. The architecture of the micro-ROS stack follows the ROS 2 architecture.

## micro-ROS

micro-ROS is a framework developed on the basis of ROS2. It aims to integrate microcontrollers with limited resources into the system that would otherwise not be able to use ROS2. The micro-ROS was created due to the large number of microcontrollers in a robot that manage almost the entire hardware layer. This makes software development very time-consuming and difficult to manage and expand. The micro-ROS succeeds in integrating a structure of layers down to the application level while requiring a minimum amount of resources from the system on which it is installed.

micro-ROS is compatible with Robot Operating System (ROS 2.0), the de facto standard for robot application development. In turn, it puts ROS 2 onto microcontrollers. micro-ROS enables the interoperability of traditional robots with IoT sensors and devices, creating truly distributed robotic systems using a common framework. micro-ROS empowers these computational devices with constrained resources to become first-class participants of the ROS ecosystem, allowing the creation of smaller robots using the same tools as well as taking advantage of the increasing overlapping between robotics, smart embedded devices, and IoT. micro-ROS follows the ROS 2 architecture and makes use of its middleware pluggability to use DDS-XRCE, which is suitable for microcontrollers.



The dark blue components shown in the architecture are developed specifically for micro-ROS. The light blue components, on the other hand, are taken from the standard ROS 2 stack.

The whole stack that is necessary to implement ROS features in microcontrollers that are lacking an operating system like Linux or Windows, which are prerequisites to operate ROS2, is being developed in a particular stack that is interoperable with ROS 2, totally and fully integrated.

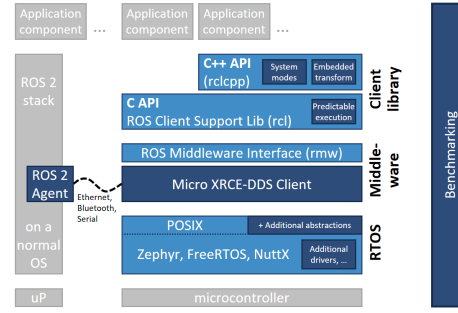


Figure 2.5: Middleware structure micro-ROS

If you are dealing with real-time, with microcontrollers, or extremely resource-constrained environments, micro ROS will offer what you need to implement standard solutions that avoid continuous implementation of ad-hoc features.

micro-ROS presents a viable solution for this project, it has not been used as it is still under great development and and very complex to work with.

## 2.4 ROS1 vs ROS2 performance evaluation

Since ROS 1 does not meet the requirements for real time, it was decided to compare the performance of ROS1 and ROS2 starting from this paper [8] [9]. To benchmark two middleware empirically is complicated since a middleware can perform better in some applications and worse in others. So what has been tried to figure out is taken the same task to perform what is the difference in latency between ROS 1.0 and ROS 2.0.

ROS enables easy and rapid software development with various tools, libraries, and rules to implement complex control algorithms in various robot platforms. However, since the robot shares the working environment with humans in real-time, it can cause physical damage to the user if malfunctions occur due to system latency. Therefore, real-time constraints must be satisfied for stable operation.

The paper [9] reports a hemipirical study comparing the two versions of ROS; ROS 1 Melodic compared with ROS 2 Dashing.

In order to evaluate the performance of the two middlewares, the same task is launched on both PCs on which the two versions of ROS are installed. The implemented software stack architecture is shown in Fig. 2.6.

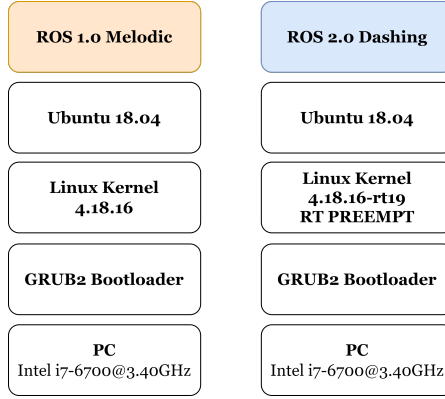


Figure 2.6: Software stack architecture of ROS 1.0 and ROS 2.0

### 2.4.1 Scheduling latency

The first test is evaluating the scheduling latency in an idle environment. The scheduling latency is defined as the time difference between the actual task activation time and the configured period. The figure 2.7 represents the latency difference between ROS 1 and ROS2 given a single task to execute in 1 ms. It can be seen that ROS 2 has a maximum latency of 11  $\mu$ s, in contrast ROS 1 has a much higher latency, 290  $\mu$ s. Therefore, the ROS 2.0 system provided higher real-time performance than ROS 1.0 and operated stably.

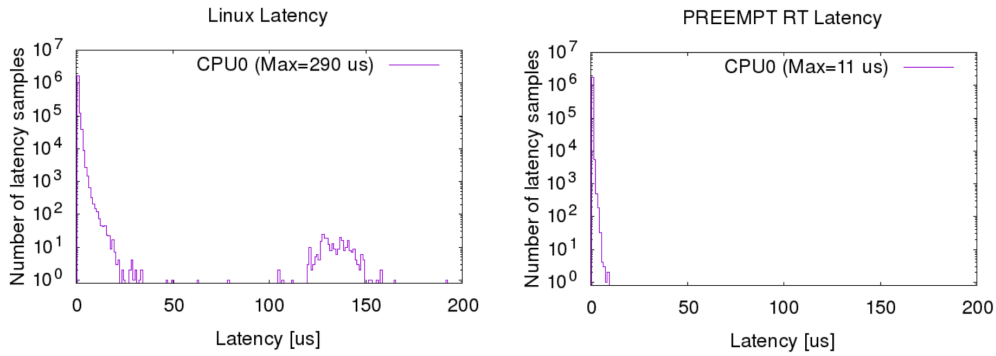


Figure 2.7: ROS 1.0 and ROS 2.0 latency comparison

### 2.4.2 Task periodicity

Then, the periodicity of the nodes were compared. Four nodes with different priority were defined, the nodes were executed to start approximately at the same time, and then, they were experimented on for 30 minutes. Two nodes were then

initialised with different periodicities. Node 1 is executed every 5 ms for 30 minutes, while node 2 is executed every 10ms for the same period of time. This experiment compares the ability and accuracy of the two middleware to execute tasks periodically. The figure 2.8 shows the execution behaviour of a node.

As shown in the figure, the initialisation of a node does not coincide with the start of its execution; the time between the release of the node and execution is called jitter, this time period may vary depending on the version of ros and the type of node. the node has finished executing, there is a period of time during which no action is taken, before the next node is initialised; again, this time period varies depending on the node type and middleware version.

After running the test for 30 minutes, the results are shown in the table 2.10. Again, the node running on ROS 2 turns out to be more efficient.

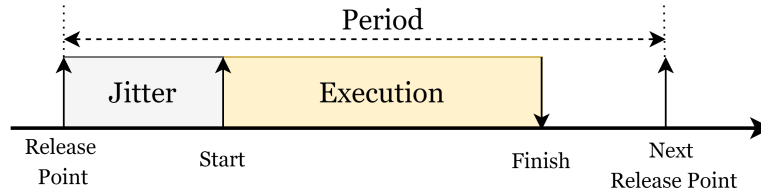


Figure 2.8: Timeline of a ROS node

As can be seen from the figure 2.9, the best results are obtained for nodes run on ROS2. Node execution in the set period is much more precise in nodes run on ROS2, on the contrary, nodes run on ROS 1 have a less precise periodicity. This result worsens by increasing the CPU load, as described in the paper [9].

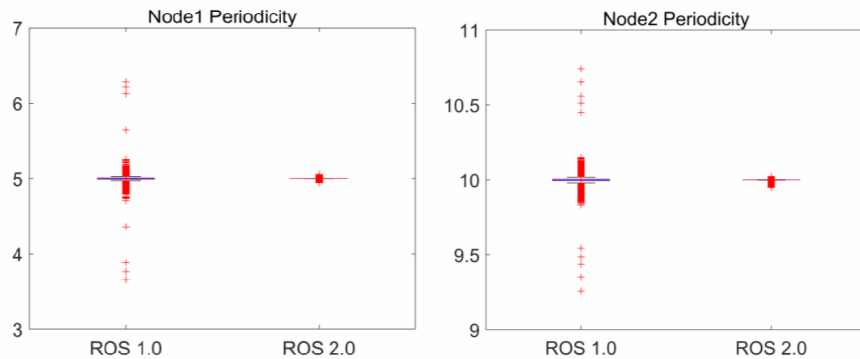


Figure 2.9: Periodicity of ROS node

ROS 1.0	Node1		Node2	
Metric (ms)	$T_{period}$	$T_{jitter}$	$T_{period}$	$T_{jitter}$
avg.	4.999062	0.016156	9.999163	0.009272
max.	8.012010	3.012010	14.581200	4.584600
min.	1.992630	0.000000	5.415400	0.000000
st. d.	0.026875	0.021498	0.022926	0.020984

ROS 2.0	Node1		Node2	
Metric (ms)	$T_{period}$	$T_{jitter}$	$T_{period}$	$T_{jitter}$
avg.	4.999819	0.000569	9.999830	0.000719
max.	5.053560	0.055300	10.023400	0.051460
min.	4.944700	0.000000	9.948540	0.000000
st. d.	0.001726	0.001640	0.001821	0.001682

Figure 2.10: Table node periodicity

### 2.4.3 Message loss

Finally, the cited paper evaluates message loss as a function of dimension of the message and sending frequency. The test is performed for 30 minutes, sending packets of different size from the PC to the RPi3 with different frequency. This test is very similar to what was done in this project. As will be described later in this project, the node on the PC will communicate and send messages of different size and frequency, to the microcontroller via USB.

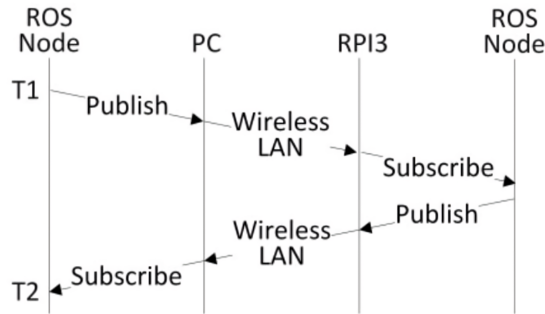


Figure 2.11: Scheme of the environment used to test the ROS node communication

The results are shown on the graph 2.12. As can be seen in both ROS 1 and ROS 2 as the packet size increases the latency increases. By increasing the size of the message sent, the latency in both cases increases slightly. As can be seen

from the graph, a constant latency is always present between ROS1 and ROS2, so in any case ROS2 has a lower latency and better performance.

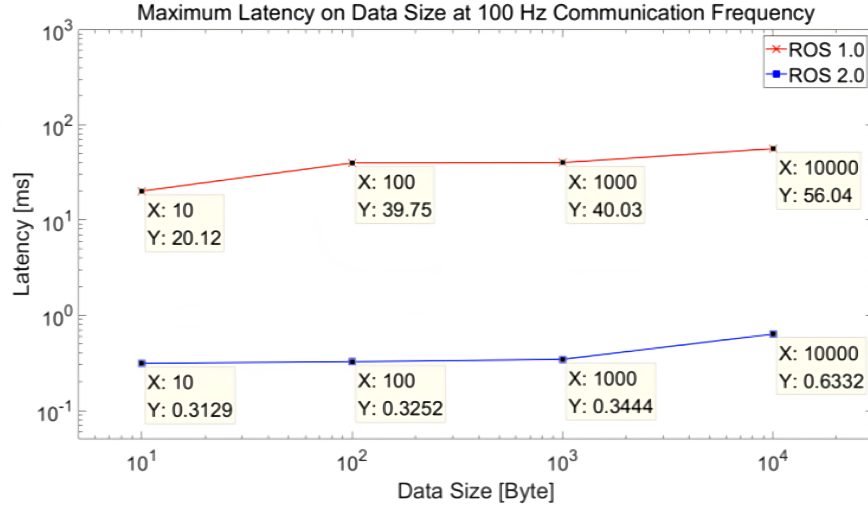


Figure 2.12: Maximum communication latency on varying data size

Given the analysis done above, it can be concluded that ROS 2 performs better than ROS 1 in terms of real time and thus latency. A more in-depth evaluation in detail is described in the following paper [8]. Also in this study, the latencies of ROS 1 and ROS 2 are compared by evaluating message loss and transmission throughput. Again, though, ROS 2 appears faster and with less loss.

From the research done and as reported in the various papers listed in the bibliography, it can be concluded that in most cases ROS2 performs better than ROS1. In that, as the graphs show, ROS2 is better in terms of latency and precision in task execution, as well as being optimised to occupy less memory. In the following, the two middlewares will be evaluated in the specific case of this project.

## 2.5 Final evaluation

As demonstrated in the previous section ROS 2 presents a better and more optimised solution for robotic software development. However, it is necessary to emphasise the negative aspects of the latter. Being developed after ROS1, ROS2 has some improvements, but it is more complex to use and to learn; moreover, being still under great development, there is a lack of study material, and there are no clear books or documentation that allow a complete understanding of the

middleware. For the purposes of the project, it was therefore decided to use the ROS 1 middleware.

Finally robotics companies such as ABB, Fanuc, Universal Robots, Robotiq, Omron and Staubli have ROS controllers, software packages and drivers for their robots in the Github repository. They provide ROS-1 support and solutions that enable integration of their tools with other industry platforms, leveraging the ROS ecosystem.

Summary of the advantages and disadvantages of ROS 1.0.

## Pros

- ROS is general, it can be used for almost any robot. Gives to the user a large number of packages to start the project
- Open source: that's the end of all-proprietary systems. It can be used some code with a permissive license (BSD) which is great for a company.
- Big community: ROS is now 10 years old, and its community is growing exponentially. It can be can find the community on github (mostly ROS packages are on github), on ROS Answers.
- Documentation: There are many tutorials and packages descriptions on Documentation - ROS Wiki. There are still some big holes in the documentation, but it is really improving over time.

## Cons

- Lack of support for some packages. Example: Some guys in a university or company have a project to do, and for this project they realize they can share their code with the ROS community by creating a library.
- Real-time computation. ROS is mainly running on Ubuntu, which is not a hard real-time OS. When it is necessary to deal with industrial robotics applications, there are quite a few chances that it is needed a real-time system
- Support for embedded systems. There is actually no real support for micro-controllers and other embedded chips. ROS must run on a computer.

The list above shows the pros and cons of the two middleware, the biggest disadvantage that can be considered in ROS 2 is the lack of material and books that explain in detail how it works and also it is more complex than ROS 1.

Therefore it was decided to use ROS 1 as it is useful for the purpose of the project although not optimally.

# Chapter 3

## ROS 1.0

### 3.1 History

The first version of ROS was released in 2007 by Willow Garage. ROS started as an open source framework, other people participated in the development. ROS caught on because the main problem with robotic platforms was having to reimplement algorithms every time you wanted to build a more complex infrastructure. For this reason, to overcome this problem ROS offers different features, like an easy process communication, code reuse and software modularity. Today, ROS represents the standard for robot programming and it is already integrated in many robots and used by many universities and companies.

### 3.2 Distribution

ROS updates are released with new ROS distributions. A new distribution of ROS is composed by an updated version of its core software and a set of new/updated ROS packages. ROS follows the same release cycle of Ubuntu Operating System: a new version of ROS is released every six months. Typically, for each Ubuntu LTS (Long Time Support) version, an LTS version of ROS is released. LTS stands for Long Term Support and means that the released software will be maintained for long period time (5 year in case of ROS and Ubuntu). The version used in this project is ROS Noetic, and it's supported by Ubuntu 20.04.

### 3.3 What is ROS (Robot Operating System)

Ros is defined as follows:

*"ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction,*

low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computer"<sup>1</sup>.

### 3.4 Ros basic functionalities

In ROS1 the essential element are nodes, which can be described as different programme parts working in parallel and communicating with each other using the publisher-subscriber paradigm. All communication of the various parts of the programme is handled by the main node: **master**. All the components in ROS and the communication between them will be listed and described below.

ROS is composed by different element:

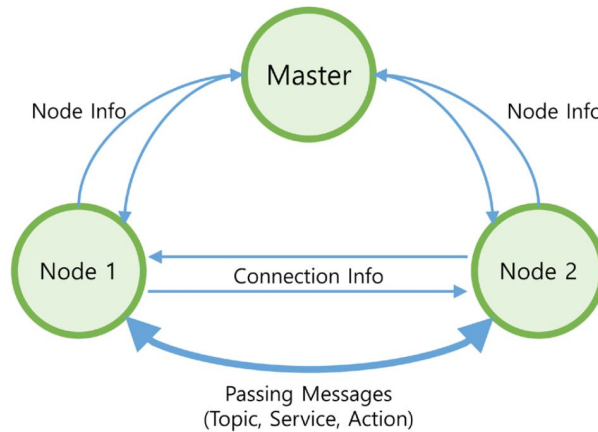


Figure 3.1: Message communication

- **Node:** Computation in ROS occurs in small units called nodes. Instead of writing a heavy program, ROS divides it into smaller units-namely, nodes. A single node can, for example, take care of the conversion of received coordinates, or the publication of the engine encoder. in fact, these nodes communicate with each other by sending messages. A node can function as a publisher, subscriber, server or client.
- **Package:** A package is the basic unit of ROS. The ROS application is developed on a package basis, and the package contains either a configuration

---

<sup>1</sup><https://www.ros.org/>



file to launch other packages or nodes. The package also contains all the files necessary for running the package, including ROS dependency libraries for running various processes, datasets, and configuration file.

- **Master:** The master acts as a name server for node-to-node connections and message communication. The connection between nodes and message communication such as topics and services are impossible without the master.

### 3.4.1 Communication between nodes

Two processes (ROS Nodes) can communicate in different ways. The first communication protocol discussed here is an asynchronous communication protocol based on the publish/subscribe paradigm in which a process streams a series of data that can be read by one or more processes. This communication relies on an entity called **topic**. In particular, each **message** in ROS is transported using named buses called topics. When a node sends a message through a topic, then it can be say the node is **publishing** a topic, while when a node receives a message through a topic, then it can be say that the node is **subscribing** to a topic.

The following list specifies the concept described above:

- **Message:** A node sends or receives data between nodes via a message. Messages are variables such as integer, floating point, and boolean.
- **Topic:** The topic is literally like a topic in a conversation. The publisher node first registers its topic with the master and then starts publishing messages on a topic. Subscriber nodes that want to receive the topic request information of the publisher node corresponding to the name of the topic registered in the master. Based on this information, the subscriber node directly connects to the publisher node to exchange messages as a topic.
- **Publish and Publisher:** The term "*publish*" stands for the action of transmitting relative messages corresponding to the topic. The publisher node registers its own information and topic with the master, and sends a message to connected subscriber nodes that are interested in the same topic. The publisher is declared in the node and can be declared multiple times in one node.
- **Subscribe and Subscriber:** The term "*subscribe*" stands for the action of receiving relative messages corresponding to the topic. The subscriber node registers its own information and topic with the master, and receives publisher information that publishes relative topic from the master. Based on received publisher information, the subscriber node directly requests connection to the

publisher node and receives messages from the connected publisher node. A subscriber is declared in the node and can be declared multiple times in one node

The **topic communication** is an **asynchronous** communication which is based on publisher and subscriber, and it is useful to transfer certain data. Since the topic continuously transmits and receives stream of messages once connected, it is often used for sensors that must periodically transmit data. The figure 3.2 shows how the communication between publisher and subscriber works. In this project, several topics were created and the main message stream is linked to the engines of the robot.

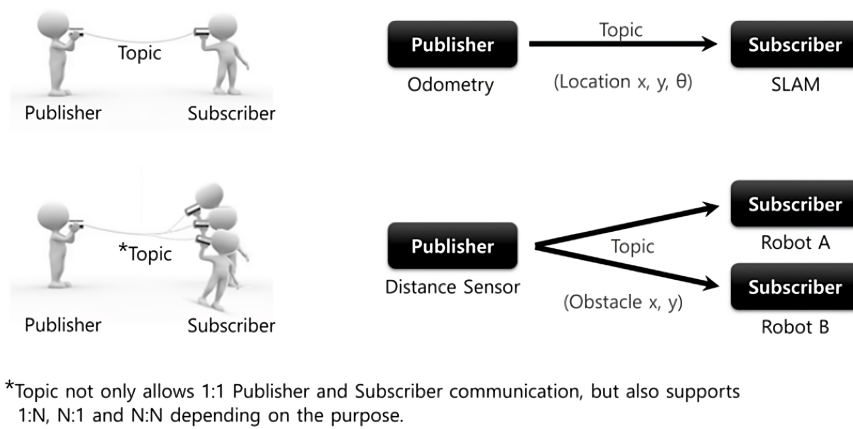


Figure 3.2: Topic message communication

**ROS Services** ROS also supports a synchronous communication protocol: ROS services. It establish a request/response communication between nodes. One node will send a request and wait until it gets a response from the other. This type of communication protocol should be used when a node is specialized in doing a specific tasks, like some complex calculation. For example, this node could be responsible to calculate the inverse of a matrix. In this way, all the other modules of the system that need to invert a given matrix could directly use this service, without replicate the inversion operation in their source code.

- **Service:** The service is synchronous bidirectional communication between the service client that requests a service regarding a particular task and the service server that is responsible for responding to requests.
- **Service server:** The "service server" is a server in the service message communication that receives a request as an input and transmits a response as an output.

- **Service client:** The *"service client"* is a client in the service message communication that requests service to the server and receives a response as an input.

The figure 3.3 represents all methods of communication between nodes; present in ROS. In this project, it was decided to make the nodes communicate using only the publisher-subscriber paradigm.

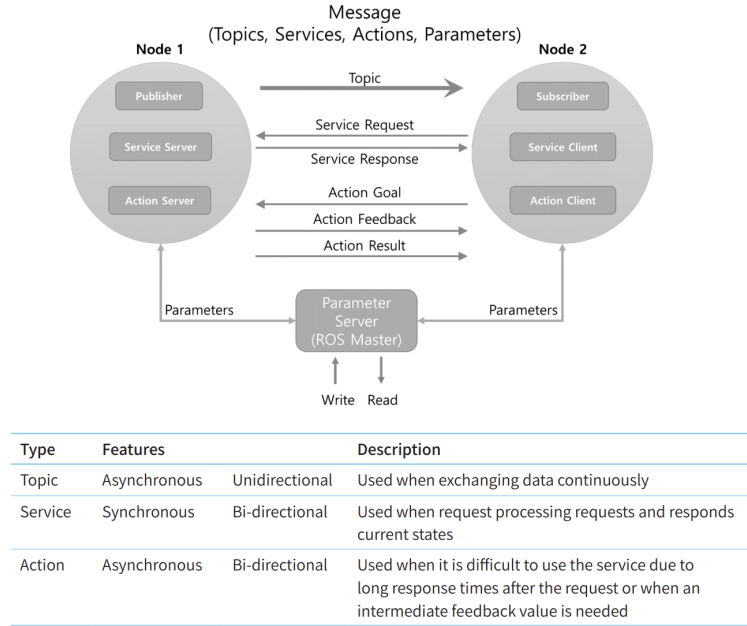


Figure 3.3: Message communication between nodes

### 3.4.2 ROS package

ROS packages are the basic units of ROS programs.

The following folder created within the workspace will contain all files relating to the project. The package can be created by running the following command from the terminal:

```
$ catkin_create_pkg package_name [dependency1] [dependency2]
```

The benefit of organising a project within a package is the possibility to share and transfer it.

The following are the main folders that the user creates within the package:

- **config:** All configuration files that are used in this ROS package are kept in this folder. This folder is created by the user and it is a common practice to name the folder config as this is where the configuration files are saved.

- **include/[package\_name]**: This folder contains the headers and libraries that are needed to use inside the package.
- **script**: This folder contains executable python scripts.
- **src**: This folder stores the C++ source codes.
- **launch**: This folder contains the launch files that are used to launch one or more ROS nodes simultaneously.

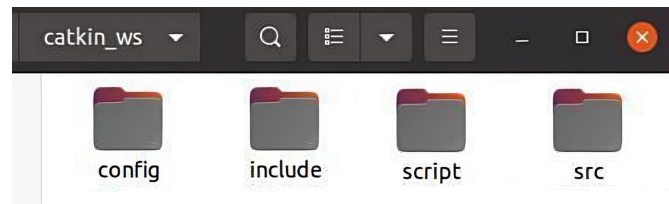


Figure 3.4: ROS's package folder

### 3.4.3 IDE - Visual Studio Code

Once the package has been created and integrated into the ROS environment, the development of the nodes that will create the application is started. In this project, all nodes on ROS were developed using the python programming language, and Visual Studio Code was used as the IDE for writing the nodes. When the IDE is opened, a file with the extension `.py` is created and saved in the package's script folder, and the ROS node is created. In this way, the node was created. The image 3.5 shows the IDE (VS Code) during the development of a node.

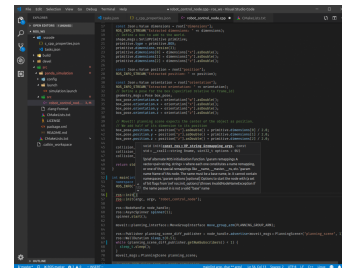


Figure 3.5: VS Code: ROS node development environment

### 3.4.4 Visualization

ROS provides a variety of tools to assist developers.

Among these tools, Ros Visualization (RViz) is very useful. It is a 3D visualizer to graphically display the contents of a topic using `visualization_msgs` messages. RViz can visualize robot models, the environments they work in, and sensor data directly from ROS topics. Built-in and custom plugins can be loaded on RViz to add additional functionalities like motion planning or motion control.

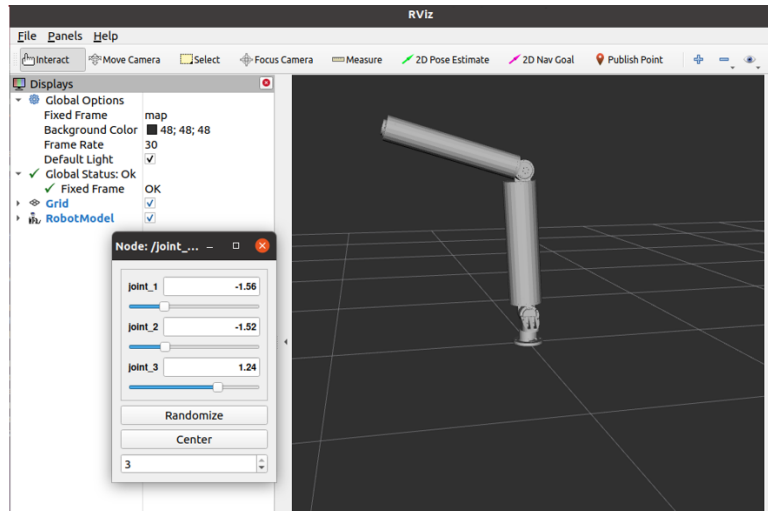


Figure 3.6: ROS Visualization (RViz)

### 3.4.5 Simulation

Another important feature of ROS is the simulation. In particular, ROS is strictly integrated with Gazebo simulator <http://gazebo.org/>, a multi-robot simulator for complex indoor and outdoor robotic simulation. In the Gazebo environment can simulate complex robots, robot sensors, and a variety of 3D objects. Gazebo already has simulation models of popular robots, sensors, and a variety of 3D objects in their repository. In addition, several plug-in already exists to interact with the simulated using ROS. Using Gazebo it can be simulate the real sensor mounted on our robots receiving exactly the same stream of data (thanks to the same ROS message definition between the real and simulated sensors). An example of the Gazebo interface is shown in Fig. 3.7, where a wheeled mobile robot is simulated.

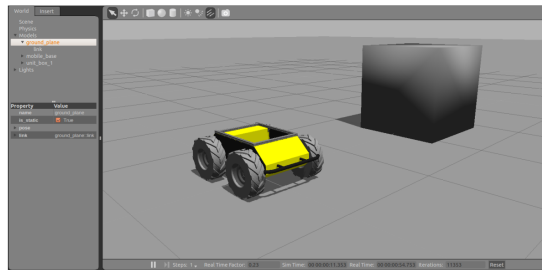


Figure 3.7: Gazebo ROS simulator

## 3.5 Rosserial

Finally, the most important package used in this project is the **roserial package**. This package enabled communication between the ROS node and the microcontroller, USB-connected.

### 3.5.1 Rosserial package

Rosserial is a package that converts ROS messages, topics, and services to be used in a serial communication. Generally, microcontrollers use serial communication like UART rather than TCP/IP which is used as default communication in ROS. Therefore, to convert message communication between a microcontroller and a computer using ROS, roserial should interpret messages for each device. In this project, the roserial package was used for UART communication with the microcontroller.

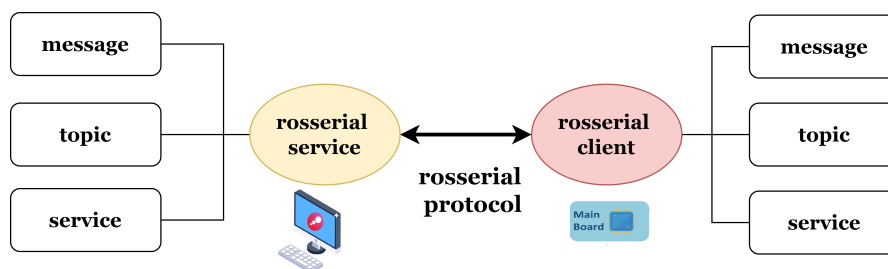


Figure 3.8: roserial server (for PC) and client (for embedded system)

For example, if the analog sensor connected to the microcontroller is read and converted its analog value into a digital value and then transmitted to the serial port, the "roserial server" node of the computer receives this sensor data and converts it to a topic used in ROS. Conversely, if motor control value from other node is received as a topic in the "roserial server" node, the "roserial server" node converts the value and sends it to the microcontroller in serial format to control the connected motor. In order to have communication via UART, as shown in Fig. 3.8 there are: **roserial server** and **roserial client**.

The **roserial server** for PC is a node which relays communication with roserial protocol between embedded devices and a PC running ROS. Depending on the programming language implemented, up to three nodes are supported as of now.

- **roserial\_python** This package is implemented with Python language and is commonly used to use roserial. This node was used inside this project since all other nodes were developed in python

- **roserial\_server** Although the performance has been improved with the use of C++ language, there are some functional limitations compared to roserial\_python.
- **roserial\_java** The roserial\_java library is used when a Java-based module is required, or when it is used with the Android SDK.

The **roserial\_client** library is ported to the microcontroller embedded platform in order to use it as a client for roserial. The library supports Arduino platform. Therefore, any board that supports Arduino can use the library, and the open source code makes it easy to port to other platforms.

- **roserial\_arduino** This library is for the Arduino board that supports Arduino UNO and Leonardo board, but it can also be used on other boards through source modification. The OpenCR board used in TurtleBot3 has a modified roserial\_arduino library.
- **roserial\_mbed** This library supports mbed platform, which is an embedded development environment, and enables the use of mbed boards.
- **roserial\_stm32** This library supports STM32 devices; this library was used in this project

### 3.5.2 Rosserial protocol

The roserial server and roserial client send and receive data in packets based on serial communication. The roserial protocol is defined in byte level and contains information for packet synchronization and data validation.

The roserial packet includes the header field to send and receive the ROS standard message and the checksum field to verify the validity of the data. The packet representation is shown next:

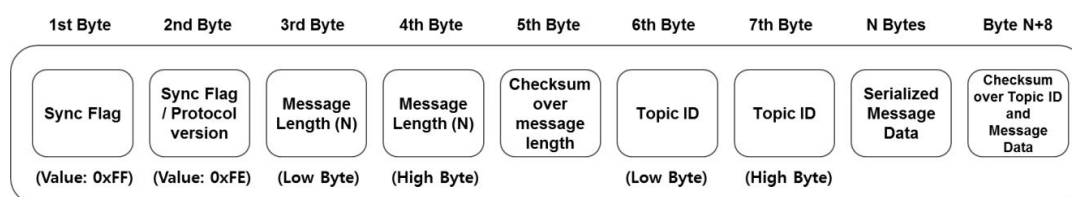


Figure 3.9: Rosserial frame format

- **Sync Flag** : This flag byte is always 0xFF and indicates the start of the packet.

- **Sync Flag/ Protocol version** : This field indicates the protocol version of ROS where Groovy is 0xFF and Hydro, Indigo, Jade, Kinetic are 0xFE
- **Message Length** : This 2 bytes field indicates the data length of the message transmitted through the packet. The Low byte comes first, followed by the High byte.
- **Checksum over message length** : The checksum verifies the validity of the message length and is calculated as follows.  
$$255 - ((\text{Message Length Low Byte} + \text{Message Length High Byte}) \% 256).$$
- **Topic ID** : The ID field consists of 2 bytes and is used as an identifier to distinguish the message type. Topic IDs from 0 to 100 are reserved for system functions. The main topic IDs used by the system are shown below and they can be displayed from 'rosterial\_msgs/TopicInfo'.
- **Serialized Message Data** : This data field contains the serialized messages.
- **Checksum over topic ID and Message Data** : This checksum is for validating Topic ID and message data, and is calculated as follows.  
$$255 - ((\text{Topic ID Low Byte} + \text{Topic ID High Byte} + \text{Data byte values}) \% 256)$$
- **Query Packet**: When the rosterial server starts, it requests information such as topic name and type to the client. When requesting information, the query packet is used. The Topic ID of query packet is 0 and the data size is 0. The data in the query packet is shown below.

*0xff 0xfe 0x00 0x00 0xff 0x00 0x00 0xff*

A negative aspect of the rosterial protocol is the memory constraints. The microcontrollers used in the embedded system have a considerably smaller memory compare to standard PCs. Therefore, the memory capacity has to be considered in advance before defining the number of publishers, subscribers, and transmit and receive buffers. By launching the following command, the rosterial server is initialised and it is possible to communicate with the device via UART.

```
$ rosrn rosterial_python serial_node.py _port:=/dev/ttyACM0 _baud:=115200
```

```
[INFO] [1495609829.326019]: ROS Serial Python Node
[INFO] [1495609829.336151]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1495609831.454144]: Note: subscribe buffer size is 1024 bytes
[INFO] [1495609831.454994]: Setup subscriber on led_out [std_msgs/Byte]
```



## Chapter 4

# Popup robot devices

A general introduction to the project and the various devices and software used was given in the previous chapters. In the following chapter, the various components used will be described in detail. The following diagram in the Fig. 4.1 illustrates the robot ecosystem.

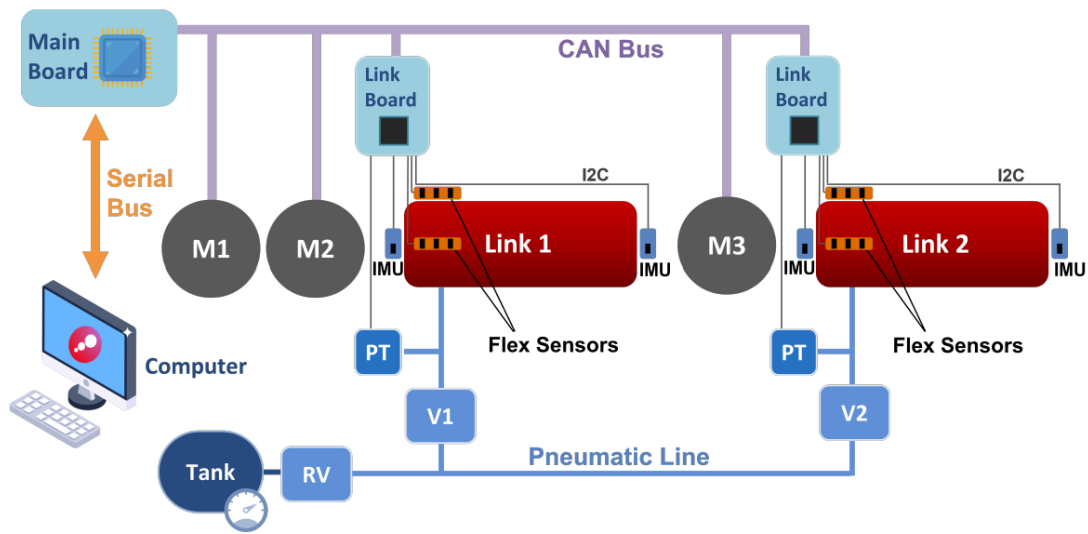


Figure 4.1: Robot ecosystem diagram

As illustrated in the figure 4.1, the ecosystem is represented by three different macro components:

- **PC:** Used to communicate with the user and provide commands to the micro-controller. Ubuntu 20.04 operating system is installed on it, which is necessary

for the operation of the ROS1 Noetic. Using ROS1, python nodes were developed to control the robot. On the PC are present the nodes that implement the graphic interface for controlling the robot and the communication with the microcontroller

- **STM32H7:** This board has the task of "*mediating*" communication between the PC and the robot's motors. The board communicates with the three motors using the CAN protocol, while it is connected with the PC via USB and communicates using the roserial protocol (communication protocol present in ROS 1)
- **Popup robot:** The main components of the popup robot are the three AK80-80 BLDC motors, which are necessary for the movement of the robot. They are arranged as follows: the motor at the base marked ID 1, above it is the ID 2 motor, and finally at the end of the first link is the motor marked ID 3. These three motors are connected in series to the board, the exchange of information between these two devices being carried out via CAN protocol.

The various components used and how they were configured will be described in detail below. The Fig. 4.2 show the connection between the various devices.

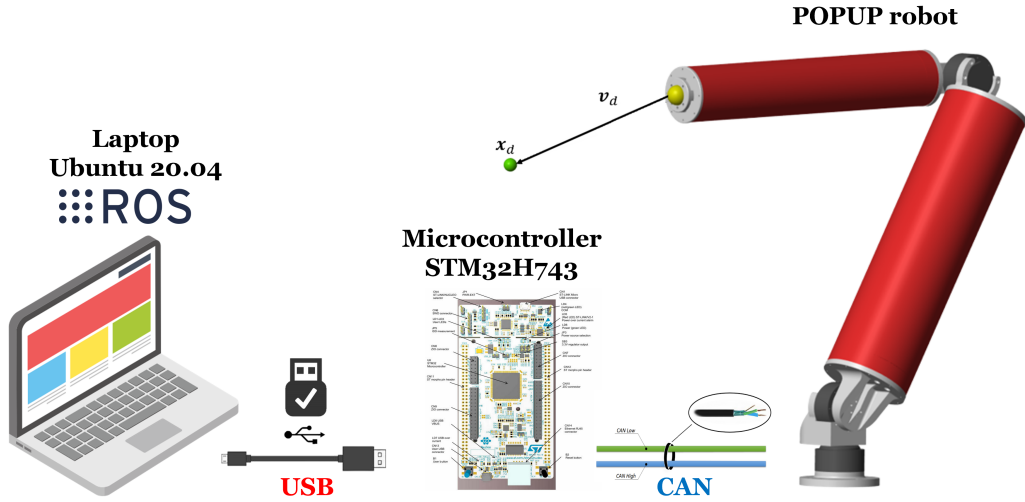


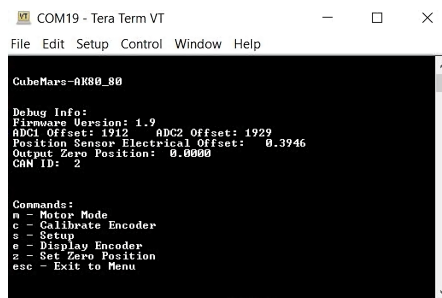
Figure 4.2: POPUP devices connection



has three different inputs:

- **Power line:** power supply is attached to the center connector. The motor can be powered by 24 V or 48 V, depending on the value of the power supply the motor will have different speed and maximum torque. In this project the motor are supplied with 24 V, since no high torque or speed is required.
- **CAN Port:** After configuration, the motor can be controlled by the microcontroller via this port. The motor sends and receives commands using the CAN protocol. The CAN port has two pins with which it is connected to the microcontroller. Connecting the motor to the board consists of two cables, on which the H signal and the L signal will pass. Messages travelling on the CAN protocol are called data frames.
- **UART:** The following port is used to configure the motor using the R-link device. Configuration of the motor is done by connecting it to the PC via USB and using the software provided. Using the motor configuration software, it is possible to set the maximum speed and torque of the motor, calibrate the encoder on the motor, set the type of communication with which commands are sent and received, and finally, it is possible to assign a unique identifier to the motor, ID. If there are several motors connected together, they must be assigned an ID to identify them. In this project there are three motors, which have been assigned IDs 1 to 3 respectively.  
Finally, being composed of a driver (containing a uC of the ST family) it is possible to carry out the firmware update through a procedure provided by the manufacturer that involves opening the back of the motor.

This output shown in Fig. 4.5 appears automatically on the terminal once the motor is linked to the PC via USB and turned on. Through this terminal it is possible to configure engines. Once the configuration is done it can be controlled by the SMT32H743 with the CAN protocol.



```

COM19 - Tera Term VT
File Edit Setup Control Window Help

CubeHaro-AK80_80

Debug Info:
Firmware Version: 1.9
ADCL Offset: 1912  ADC2 Offset: 1929
Position Sensor Electrical Offset: 0.3946
Output Zero Position: 0.0000
CAN ID: 2

Commands:
n - Motor Mode
c - Calibrate Encoder
s - Setup
e - Display Encoder
z - Set Zero Position
esc - Exit to Menu
  
```

Figure 4.5: Motor AK80-80 USB configuration

### 4.1.1 Controller Area Network (CAN)

In this project the engine receives messages processed by the STM32H7, messages sent via CAN must have a well-defined formatting; starting from the following notes [10] follows a brief general description of the CAN protocol and how the engine driver handles this communication.

The CAN protocol is a set of rules for transmitting and receiving messages in a network of electronic devices. It defines how data is transferred from one device to another in a network. The Controller Area Network was developed by Robert Bosch GmbH for automotive applications in the early 1980s and publicly released in 1986. CAN is a serial, multimaster, multicast protocol, which means that when the bus is free, any node can send a message (multimaster), and all nodes may receive and act on the message (multicast). The node that initiates the message is called the transmitter; any node not sending a message is called a receiver. Messages are assigned static priorities, and a transmitting node will remain a transmitter until the bus becomes idle or until it is superseded by a node with a higher priority message through a process called arbitration. A CAN message may contain up to 8 bytes of data. A message identifier describes the data content and is used by receiving nodes to determine the destination on the network. Bit rates up to 1Mbit/s are possible in short networks ( $\leq 40$  m). Longer network distances reduce the available bit rate (125kbit/s at 500 m, for example). "High speed" CAN is considered to be 500kbit/s.

There are four types of CAN messages, or "frames:" Data Frame, Remote Frame, Error Frame and Overload Frame.

- The data frame is the standard CAN message, broadcasting data from the transmitter to the other nodes on the bus.
- A remote frame is broadcast by a transmitter to request data from a specific node.
- An error frame may be transmitted by any node that detects a bus error.
- Overload frames are used to introduce additional delay between data or remote frames.

The **CAN data frame** is composed of seven fields: Start of frame (SOF), arbitration, control, data, cyclical redundancy check (CRC), acknowledge (ACK) and end of frame (EOF). CAN message bits are referred to as "dominant" (0) or "recessive" (1). The SOF field consists of one dominant bit. All network nodes waiting to transmit synchronize with the SOF and begin transmitting at the same time. An arbitration scheme determines which of the nodes attempting to transmit will actually control the bus.

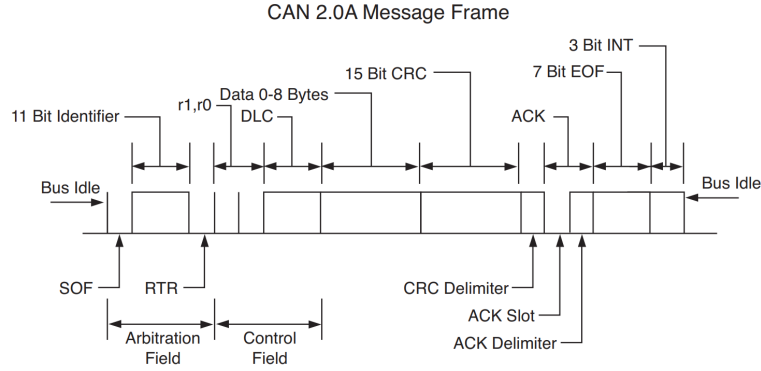


Figure 4.6: CAN data frame

Table 1: CAN 2.0A Message Frame		
Field	Length (bits)	Description
Start of Frame (SOF)	1	Must be dominant
Identifier	11	Unique identifier indicates priority
Remote Transmission Request (RTR)	1	Dominant in data frames; recessive in remote frames
Reserved	2	Must be dominant
Data Length Code (DLC)	4	Number of data bytes (0-8)
Data Field	0-8 bytes	Length determined by DLC field
Cyclic Redundancy Check (CRC)	15	
CRC Delimiter	1	Must be recessive
Acknowledge (ACK)	1	Transmitter sends recessive; receiver asserts dominant
ACK Delimiter	1	Must be recessive
End of Frame (EOF)	7	Must be recessive

Figure 4.7: CAN data frame

The **control field of the data frame** consists of 6 bits (of which only the lower 4 are used) that indicate the amount of data in the message. Since up to 8 bytes of data may be sent in one message, the control field may take values ranging from 000000 to 000111. The data to be transmitted are contained in the data field. The most significant bit (MSB) of a data byte is sent first.

CAN provides a robust, simple and flexible network solution for manufacturing, automotive and many other applications. The major drawback to CAN is that message latency is non-determinant (due to the existence of Error Frames, Overload Frames and retransmissions), and latency increases with the amount of traffic on the bus. In general, bus utilization should not exceed 30% messages do not experience unacceptable delay. Bus utilization is defined as total bit consumption over the total bits available, and is calculated as follows:

$$\text{utilization} = \frac{47 \text{ bits} \times \text{number of transmissions/unit\_time} \times 1.1}{\text{total bits available}}$$

Finally, after this brief introduction on CAN in order to communicate correctly with the motor's driver, the data frame transmitted via CAN must be formatted as shown in the figure 4.8. The sending and receiving of data via CAN is carried out by the STM32H7 board, it was therefore necessary to create a function that would format the commands to be sent to the motors in the correct way, the following functions were written in C and are present in the board's firmware. An important feature of the motor driver is the automatic sending of motor feedback immediately after receiving a command via CAN. They were therefore created respectively: the `CAN_TX_AK8080()` function, which takes the float variables as input and creates the data frame to be sent to the motor, the `unpack_motor_FB()` function performs a similar function but for the feedback received from the motors, again via CAN

IDENTIFIER: SET MOTOR ID NUMBER (DEFAULT IS 1)				FRAME TYPE: STANDARD FRAME	
FRAME FORMAT: DATA				DLC: 8 BYTES	
Data Field	DATA[0]	DATA[1]	DATA[2]	DATA[3]	
Data bits	7-0	7-0	7-0	7-4	3-0
Data content	Motor position 8-H	Motor position 8-L	Motor speed 8-H	Motor speed 4-L	KP 4-H
Data Field	DATA[4]	DATA[5]	DATA[6]		DATA[7]
Data bits	7-0	7-0	7-4	3-0	0-7
Data content	KP 8-L	KD 8-H	KD 4-L	Current 4-H	Current 8-L

Figure 4.8: Data frame - Actuator AK80-80

### 4.1.2 Actuator - PID schematic

After having described how communication via CAN with the motors takes place, it is necessary to define what kind of commands to send to the latter in order to perform the rotation. The presence of the driver with a uC allows these motors to have internal control. Inside the driver is present a PID control which allows greater control of the motor.

The figure 4.9 shows the logical scheme of the PID present inside the motor, so in order to make the motor move five different parameters must be sent to the driver:

- **id**: identifies the engine to which the command will be sent
- **desired position**: indicates the final position (in rad) at which the motor will stop,
- **desired speed**: indicates the motor's rotational speed (if set to zero the motor will stop),

- **kp value:** parameter referring to position can take a value between 0 and 500 (the relationship between kp and desired position will be described in detail below),
- **kd value:** parameter referring to speed between 0 and 5,
- **desired torque:** converted into current, indicates the motor's torque to reach.

For only position or speed or torque control loop only the corresponding variable must be set, the other parameter required by the driver must be set to zero.

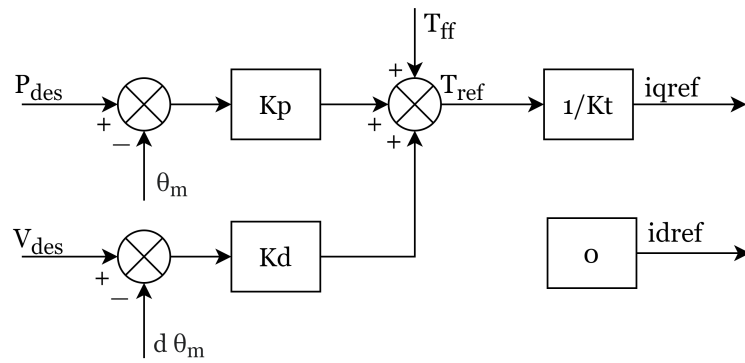


Figure 4.9: Schematic of the PID controller inside the actuator's driver

## 4.2 Microcontroller STM32H7

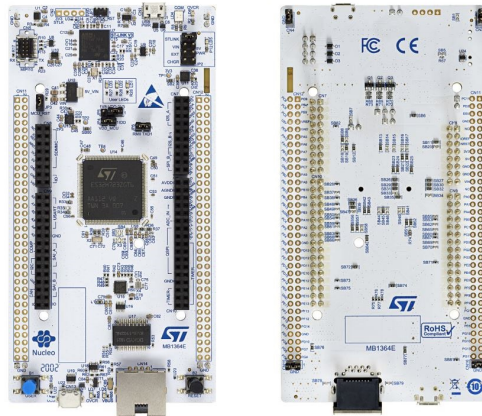


Figure 4.10: Microcontroller STM32H7



Another important component of the robot is the microcontroller, necessary to control the motors and the sensor present on the robot. A microcontroller is defined as follows:

*A microcontroller is a compact integrated circuit designed to govern a specific operation in an embedded system. A typical microcontroller includes a processor, memory and input/output (I/O) peripherals on a single chip.*

The microcontroller used for this project is the STM32H7.

STM32 is a family of 32-bit Flash microcontrollers developed by ST Microelectronics. It is based on the ARM Cortex-M processor and offers a range of 32-bit products that high performance, real-time functionality, digital signal processing, and low-voltage operation. The STM32 H7-series is a group of high performance STM32 microcontrollers based on the ARM Cortex-M7F core with double-precision floating point unit and optional second Cortex-M4F core with single-precision floating point. Cortex-M7F core can reach working frequency up to 480 MHz, while Cortex-M4F - up to 240 MHz. Each of these cores can work independently or as master/slave core.

The STM32H7 Series is the first series of STM32 microcontrollers in 40 nm process technology and the first series of ARM Cortex-M7-based microcontrollers which is able to run up to 480 MHz, allowing a performance boost versus previous series of Cortex-M microcontrollers, reaching new performance records of 1027 DMIPS and 2400 CoreMark <sup>1</sup>.

A high-performance microcontroller was chosen for motor management in order to guarantee the management of three motors simultaneously. As briefly mentioned before the microcontroller performs the function of "*mediator*" between the PC and the motors. The node in the PC manage all the part related to sending commands and receiving the feedback; on the PC is also present the graphical interface that allows interaction with the robot.

The board is connected to the PC via USB, through which receives commands to be sent to the motors. The motors exchange data with the microcontroller using the CAN protocol. They are connected to the board via two wires on which the data frames travel. The microcontroller is then responsible of "translating" the commands received from the PC into data frames to be sent to the motors.

---

<sup>1</sup><https://en.wikipedia.org/>

## 4.3 PC/Raspebrry Pi4



Figure 4.11: Raspberry Pi4 or PC - POPUP controller

After describing the motors that make up the robot and the board that manages them, it is necessary to describe who is the device that defines the value of the input depending on the current position of the robot. This task is performed by a PC on which Ubuntu 20.04 has been installed. The same task can be done using a Raspberry Pi4.

The robot control was fully developed using ROS 1 Noetic, which as described in previous chapters is a middleware that presents several tools for robotic software development. The version of ROS used must be installed on the operating system Ubuntu 20.04. All the node used for controlling the robot and display data was developed using python 3.10. The PC is then connected to the STM32H7 board via USB, all data processing concerning the robot is done by python scripts running on the PC. Using python, it was possible to create several nodes on ROS that control the robot and manage the data received. The following nodes were created with python: for position and speed control of the motors, for plotting real-time graphs on the status of the motors, for creating a GUI used to sending commands to the motor. A GUI have also been developed using python in order to make the control of the motors easier and more immediate.

The use of ROS 1 and python made it possible to control the robot in a simple and effective way, creating a modular structure that can be easily integrated with other devices.

## Chapter 5

# STM32H7 configuration

In the previous chapters, the various components and devices that constitute the POPUP robot ecosystem have been described. The main features of these components have been illustrated.

In this chapter, a more in-depth description will be given, illustrating the configuration made on the STM32 board. As described above, the high-performance STM32H7 board was chosen for controlling the robot. The board's firmware was written using the STM32CubeIDE.

STM32CubeIDE is an all-in-one multi-OS development tool, which is part of the STM32Cube software ecosystem. It is an advanced C/C++ development platform with peripheral configuration, code generation, code compilation, and debug features for STM32 microcontrollers and microprocessors. The peripheral configuration made on the board will be described below.

- LEDs: Used to understand the status of the programme during execution. Used for debugging purposes
- Timers (TIM2, TIM8): Used for periodic monitoring of motors and reception of commands from the PC
- FDCAN: peripheral for communication between microcontroller and motors
- USART3: peripheral for communication between PC and microcontroller

### 5.1 UART3 configuration

In order to enable the exchange of data between the PC and the board, it is necessary to configure the UART.

A **universal asynchronous receiver-transmitter** is a computer hardware device for asynchronous serial communication in which the data format and transmission speeds are configurable. It sends data bits one by one, from the least significant to the most significant, framed by start and stop bits so that precise timing is handled by the communication channel <sup>1</sup>.

### 5.1.1 UART: Hardware Communication Protocol

As described in the book [11] UART, or universal asynchronous receiver-transmitter, is one of the most used device-to-device communication protocols. The transmitting UART is connected to a controlling data bus that sends data in a parallel form. From this, the data will now be transmitted on the transmission line (wire) serially, bit by bit, to the receiving UART. This, in turn, will convert the serial data into parallel for the receiving device.



Figure 5.1: UART packet

#### Data Transmission:

In UART, the mode of transmission is in the form of a packet. The piece that connects the transmitter and receiver includes the creation of serial packets and controls those physical hardware lines. A packet consists of a start bit, data frame, a parity bit, and stop bits.

#### Start Bit:

The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

#### Data Frame:

The data frame contains the actual data being transferred. It can be five bits up to eight bits long if a parity bit is used. If no parity bit is used, the data frame can be nine bits long. In most cases, the data is sent with the least significant bit first.

---

<sup>1</sup>wikipedia

**Parity:**

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

**Stop Bits:**

To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) duration.

**7.7. USART3****Mode: Asynchronous****7.7.1. Parameter Settings:****Basic Parameters:**

Baud Rate 115200  
 Word Length 8 Bits (including Parity)  
 Parity None  
 Stop Bits 1

**Advanced Parameters:**

Data Direction Receive and Transmit  
 Over Sampling 16 Samples  
 Single Sample Disable  
 ClockPrescaler 1  
 Fifo Mode Disable

**8.2. DMA configuration**

DMA request	Stream	Direction	Priority
USART3_RX	DMA1_Stream0	Peripheral To Memory	High *
USART3_TX	DMA1_Stream1	Memory To Peripheral	High *

**USART3\_RX: DMA1\_Stream0 DMA request Settings:**

Mode: Normal  
 Use fto: Disable  
 Peripheral Increment: Disable  
 Memory Increment: **Enable \***  
 Peripheral Data Width: Byte  
 Memory Data Width: Byte

**USART3\_TX: DMA1\_Stream1 DMA request Settings:**

Mode: Normal  
 Use fto: Disable  
 Peripheral Increment: Disable  
 Memory Increment: **Enable \***  
 Peripheral Data Width: Byte  
 Memory Data Width: Byte

Figure 5.2: USART3 &amp; DMA - IDE configuration

In the STM32H7 several pins allow communication via UART, in this case for convenience UART3 has been configured, which refers to the micro-USB port on the board. The figure 5.2 shows the configuration of USART3 and DMA, the complete detailed configuration of the board can be found in the project report. In this case, in order to have a faster data exchange between PC and board, USART3 was set to interrupt, and also DMA was enabled for both input and output data.

**The Interrupt Method:** Using interrupt signals is a convenient way to

achieve serial UART data reception. The CPU initializes the UART receive hardware so that it fires an interrupt signal whenever a new byte of data is received. And in the ISR code, we save the received data in a buffer for further processing.

This way of handling the UART data reception is a non-blocking way. As the CPU initiates the receiving process and it resumes executing the main code. Until an interrupt is received, then it freezes the main context and switches to the ISR handler to save the received byte to a buffer and switches back to the main context and resume the main code. Finally, by setting DMA (direct memory access) on UART3 it is able to increase the speed of communication via USB.

**The DMA Method:** Using the DMA unit in order to direct the received serial UART data from the UART peripheral directly to the memory is considered to be the most efficient way to do such a task. It requires no CPU intervention at all, you'll have only to set it up and go execute the main application code. The DMA will notify back the CPU upon reception completion and that received data buffer will be in the pre-programmed location.

The DMA can prioritize the channels, decide on the data width, and also the amount of data to be transferred up to 65536 bytes. Which is amazing in fact, and all you've got to do is to initialize it like this.

Finally, a crucial parameter that can be changed is the baud rate, which in this case is set to 115200. The baud rate is the speed at which information is transferred to a communication channel. In the context of the serial port, the set baud rate will serve as the maximum number of bits per second to be transferred.

Using STM32CubeIDE, it was possible to configure this peripheral. It was only necessary to set the baud rate and enable DMA, after which the IDE automatically generated the C code containing the configuration made. The configuration of the all board's peripheral was made via the graphic interface is shown in the Fig. 5.3.

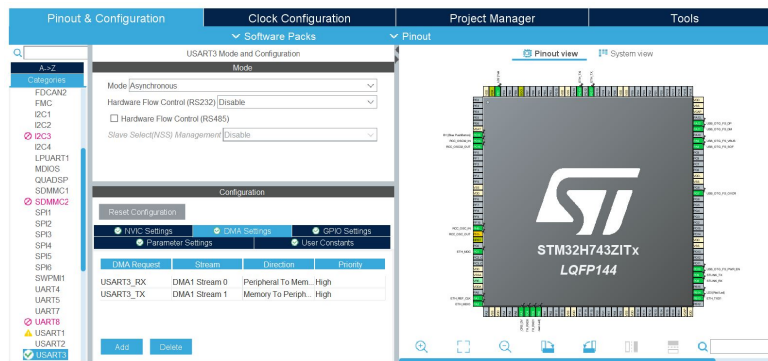


Figure 5.3: Example IDE configuration

## 5.2 FDCAN configuration

The configuration of the FDCAN peripherals is also highly significant., which are essential for the board’s communication with the motors. The configuration is shown in the Fig. 5.4. An important parameter to take into account when configuring this peripheral is the size of the FIFO, (size of the message queue), because with three different engines to which messages containing different values are sent, it is important that they are not lost. By setting the FIFO to 5, it was possible to communicate at a high speed with the engines, without losing data. When configuring the board’s CAN, it must also be borne in mind that the motors are connected in series and that whenever an input is sent, the driver of the motor that received the message automatically sends feedback to the board.

<b>7.2. FDCAN1</b>			
<b>mode: Activated</b>			
<u>7.2.1. Parameter Settings:</u>			
<b>Basic Parameters:</b>			
Frame Format	Classic mode	Data Prescaler	1
Mode	Normal mode	Data Sync Jump Width	1
Auto Retransmission	Disable	Data Time Seg1	1
Transmit Pause	Disable	Data Time Seg2	1
Protocol Exception	Disable	Message Ram Offset	0
Nominal Prescaler	<b>4 *</b>	Std Filters Nbr	0
Nominal Sync Jump Width	1	Ext Filters Nbr	0
Nominal Time Seg1	<b>12 *</b>	Rx Fifo0 Elmts Nbr	<b>5 *</b>
Nominal Time Seg2	2	Rx Fifo0 Elmt Size	8 bytes data field

Figure 5.4: FDCAN - IDE configuration

## 5.3 Timer configuration

Finally, two timers of different periods were used in order to have a constant control and exchange of data between the various devices. Specifically, the TIM2 and TIM8 were used. The TIM8 whose configuration is shown in the figure 5.5 periodically calls up the function for sending data to the motors, the `CAN_TX_AK8080()` function is the main function for sending data to the motors; in particular this function takes as input the variables to be sent to the motor and return the data frame which will be sent to the drivers via CAN. As a result, the motor driver sends back the status of the motor via CAN every time it receives input data. Then the TIM8 periodically calling up the function for motor control, it is possible to send each motor a command.

The TIM2 has a much faster period than the TIM8. The TIM2 was implemented at a later stage after communication with the engines was stable. The

TIM2, the configuration of which is shown in the figure 5.5, periodically calls up the `PIDcal()` function, the following function written in C implements a PID control for the three motor. This function has the task of calculating the torque to be sent to the motors based on their feedback speed, in order to compensate the load on the motors. The PID function, being called up periodically at a very high frequency, guarantees immediate compensation on the motors.

The configuration of the timers was done via the graphical interface as in the previous case. To set a timer, it is necessary to specify the period, the prescaler and to enable the use of the timer in interrupts. The timer period was calculated as follows:

$$T_{\text{out}} = \frac{\text{PSC} \times \text{CounterPeriod}}{F_{CLK}}$$

$$\text{TIM8} = \frac{75 \times 65535}{240 \text{ MHz}} = 20.5 \text{ ms} \quad (5.1)$$

$$\text{TIM2} = \frac{60 \times 1000}{240 \text{ MHz}} = 250 \text{ us}$$

- **Prescaler:** it divides the timer clock by a factor ranging from 1 up to 65535 (this means that the prescaler register has a 16-bit resolution). For example, if the bus where the timer is connected runs at 48MHz, then a prescaler value equal to 48 lowers the counting frequency to 1MHz.
- **CounterMode:** it defines the counting direction of the timer. Some counting modes are available only in general purpose and advanced timers. For basic timers, only the `TIM_COUNTERMODE_UP` is defined.
- **Period:** sets the maximum value for the timer counter before it restarts counting again. This can assume a value from 0x1 to 0xFFFF (65535) for 16-bit timers, and from 0x1 to 0xFFFF FFFF for TIM2 and TIM5 timers in those MCUs that implement them as 32-bit timers. If Period is set to 0x0 the timer does not start.

<b>7.6. TIM8</b>		<b>7.5. TIM2</b>	
<b>Clock Source : Internal Clock</b>		<b>Clock Source : Internal Clock</b>	
<b>Channel1: Output Compare CH1</b>		<b>7.5.1. Parameter Settings:</b>	
<u>7.6.1. Parameter Settings:</u>			
<b>Counter Settings:</b>		<b>Counter Settings:</b>	
Prescaler (PSC - 16 bits value)	75*	Prescaler (PSC - 16 bits value)	60*
Counter Mode	Up	Counter Mode	Up
Counter Period (AutoReload Register - 16 bits value )	65535	Counter Period (AutoReload Register - 32 bits value )	1000 *
Internal Clock Division (CKD)	No Division	Internal Clock Division (CKD)	No Division
Repetition Counter (RCR - 16 bits value)	0	auto-reload preload	Disable
auto-reload preload	Disable	<b>Trigger Output (TRGO) Parameters:</b>	
		Master/Slave Mode (MSM bit)	Disable (Trigger input effect not delayed)
		Trigger Event Selection TRGO	Reset (UG bit from TIMx_EGR)

Figure 5.5: TIM2 & TIM8 - IDE configuration



Finally, three GPIO as led were configured. These three LEDs are used to provide feedback on the status of the programme by only looking at the board. For example based on the led that is switched on, can be tell if the board is performing a certain task.

Once these peripherals have been configured with the graphical interface provided by IDE, the C code that will be loaded onto the board is generated. This just described represents the configuration required to operate the board. However, it is necessary to modify a few more parameters in order to make the board able to recive the command from the PC through USB. By adding a few files to the firmware's board, it will be able to receive data from ROS, via the roserial protocol, these steps will be described later.

## 5.4 Rosserial configuration

### 5.4.1 STM32H7

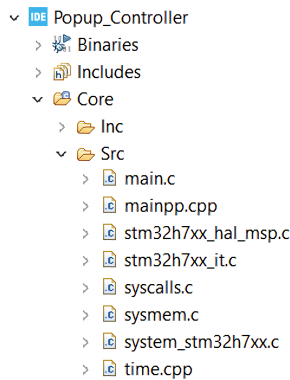


Figure 5.6: STM32H7 firmware file

In order to receive data from the PC via USB using the **roserial protocol**, the board requires additional configuration. Once the parameters for the peripherals have been defined, the IDE automatically generates the .c code containing all the peripheral information. The code for motor control was also written to this file.

The IDE automatically generates the **main.c** file, which contains the code that defines the peripheral configuration and code developed to control the robot. As can be seen from the image 5.6, the /src folder also contains files with the extension .cpp, meaning that the files are written in C++. Files with the extension .cpp are externally imported files on which parameters for communication with ROS are defined, and on which publishers and subscriber are defined.

- **main.c:** The following file contains the peripherals configuration and the functions for sending commands to the motors via CAN. The functions used to communicate with the motors have the task of correctly creating the data frame to be transmitted via CAN. This file contains the code for communication with the engines.
- **mainpp.cpp:** The following file was imported and is required for communication via the roserial protocol. Within this file, the publishers and subscribers that will communicate with the python node on the PC are defined. The files

needed for communication with ROS have the extension `.cpp` because ROS nodes can only be written in C++ or in python. This file contains the code for communication with the ROS nodes.

- **main.h, mainpp.h:** The following files represent the headers of the `main.c` and `mainpp.cpp`, in which the function definitions are listed.
- **ros.h, STM32Hardware.h, node\_handle.h:** Finally, the following header files are imported and contain the re-configuration of the UART port in order to be able to use it with the `roserial` protocol. Modifying these files it is possible to specify the maximum number of publishers and subscribers, the baud rate of the USB port and the buffer size of each publisher and subscriber and other technical parameters related to communication. In addition, the `node_handle.h` end contains all the functions necessary for formatting the message according to the parameters defined by the `roserial` protocol.

After importing these files into the project folder (`/src`), a further modification of project parameters is necessary; these modifications are reported in the appendix.

## 5.4.2 PC

With this configuration, made using the IDE, the microcontroller is able to communicate with the two devices, i.e. receive and send commands via CAN to the motors, and receive input and communicate the status of the robot to the PC. In this section will describe how the PC or RPi must be configured to allow POPUP control via ROS.

As described above, ROS is an open-source middleware that brings together various tools for robotics software development, but requires an operating system to run. By construction ROS 1 is based on the linux operating system Ubuntu, each ROS distribution requires a certain version of Ubuntu. A ROS distribution is a versioned set of ROS packages.

For this project it was decided to use the latest ROS 1 distro: **ROS Noetic Ninjemys** released in May 23rd, 2020, which it is based on Ubuntu 20.04. For the purposes of the project, the previous distribution could also be used: ROS Melodic Morenia, based on Ubuntu 18.04. It was decided to use the latest distribution in order to have support for longer period, (shown in the figure as EOL).

It was therefore necessary as a first step to install Ubuntu 20.04 on the PC. Initially all the development part of the ROS nodes was done with the PC on which a virtual machine was installed there. In this way through one PC it was possible to write the firmware for the board, using the STM32CubeIDE software, installed on Windows 10, as well as develop the ROS nodes in python using ROS

Distro	Release date	Poster	Turtle, turtle in tutorial	EOL date
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)

Figure 5.7: ROS distribution

installed on the virtual machine present on Win 10.

The installation of the virtual machine and configuration of Ubuntu was done by following the guide at this link <sup>2</sup>.

Once the installation of the operating system is finished, the installation of the middleware is carried out. The procedure for installing ROS Noetic was done by following the guide in this book [6].

Finally, once these installations have been successfully completed, the roserial package must be installed in order to enable communication between the PC and the microcontroller. Rosserial <sup>3</sup> is a protocol for wrapping standard ROS serialized messages and multiplexing multiple topics and services over a character device such as a serial port or network socket. The installation of this package was done by following the guide in this book [2].

Once the installation of this package is complete, the entire environment is ready for the node development.

Below are some basic commands used within ROS, other commands are collected within the ROS cheat sheet:

\$ **roscore** : A collection of nodes and programs that are pre-requisites of a ROS-based system. roscore must run in order for ROS nodes to communicate.

\$ **rostopic**: A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages

\$ **roslaunch**: roslaunch allows you to run an executable in an arbitrary package without having to cd (or roscd) there first.

<sup>2</sup>[https://linuxhint.com/install\\_ubuntu\\_vmware\\_workstation/](https://linuxhint.com/install_ubuntu_vmware_workstation/)

<sup>3</sup><http://wiki.ros.org/roserial>

\$ **roslaunch**: Starts ROS nodes locally and remotely via SSH, as well as setting parameters on the parameter server.

## 5.5 Basic publisher/subscriber on STM32H7

Once the installations of all components were completed, a basic script was created to check their functioning. In order to verify the communication between the board and PC, a publisher and a subscriber were written into the mainpp.cpp file and then loaded into the STM32H7 microcontroller. Once the board is connected via USB to the PC, it is possible to control the board from Ubuntu's terminal. The code to verify the correct communication between the PC and the microcontroller has:

the publisher which publish the string: "Hello world from STM32!" on the topic "chatter", while the subscriber waits for the message sent by the terminal to the topic "toggle led", once the message is received, the LED on the microcontroller changes state.

The following code represents the initialisation of the publisher and subscriber in the mainpp.cpp file. This code will be loaded onto the STM32H7 using the STM32CubeIDE.

```
ros::NodeHandle nh;
void messageCb( const std_msgs::Empty& toggle_msg);

std_msgs::String str_msg;
//initialization of the ros publisher to the topic "chatter"
ros::Publisher chatter("chatter", &str_msg);
char hello[] = "Hello world from STM32!";

ros::Subscriber<std_msgs::Empty> sub("toggle_led", &messageCb );
```

After writing the publisher and subscriber functions, the code is uploaded to the board, which is then connected to the PC via USB. To check the operation of the publisher and subscriber four different command are launched from Ubuntu's terminal:

- (a) terminal 1: \$ **roscore**: the following command is essential for communication between nodes in ROS 1. Enable the master node.
- (b) terminal 2: \$ **roslaunch roserial\_python serial\_node.py /dev/ttyACM0 115200**: the command that calls the roserial\_python node is launched, allowing communication between the board and the PC. If the board is detected correctly by the terminal, the output will be as shown in the figure below.

- (c) terminal 3: `$ rostopic echo /chatter`: this command reads the messages on the topic `/chatter`", as shown in the figure 5.8. This verifies that the board periodically sends messages, so the publisher function works correctly.
- (d) terminal 4: `$ rostopic pub toggle_led std_msgs/Empty --once`: by issuing this command, it can be check whether the board also works as a subscriber, i.e. is able to receive commands from the terminal and subsequently execute them. In this case the board receives a empty message on the topic `"toggle_led"`, once the message is received the the led on the board changes state.

```
adminub@adminub:~$ roscore
... logging to /home/adminub/.ros/log/0859487a-3535-11ed-8aa4-6756f412be93/roslaun
ch-adminub-virtual-machine-2576.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
none checking log file disk usage. Usage is <1GB.

started roslaunch server http://adminub-virtual-machine:42865/
ros_comm version 1.15.14

SUMMARY
*****
PARAMETERS
 * /roslistro: noetic
 * /rosversion: 1.15.14
NODES
auto-starting new master
process[master]: started with pid [2614]
ROS_MASTER_URI=http://adminub-virtual-machine:11311/
```

(a) `$ roscore`

```
adminub@adminub:~$ sudo chmod 666 /dev/ttyACM0
adminub@adminub:~$ roslaunch roserial_python serial_node.py _port:
=/dev/ttyACM0 _baud:=115200
[INFO] [1663274797.078452]: ROS Serial Python Node
[INFO] [1663274797.083760]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1663274799.193724]: Requesting topics...
[INFO] [1663274800.101180]: Note: publish buffer size is 1024 bytes
[INFO] [1663274800.103395]: Setup publisher on chatter [std_msgs/string]
[INFO] [1663274800.111188]: Note: subscribe buffer size is 1024 bytes
[INFO] [1663274800.113287]: Setup subscriber on toggle_led [std_msgs/Empty]
```

(b) `$ roslaunch roserial_python`

```
adminub@adminub:~$ rostopic echo /chatter
data: "Hello world from STM32!"
---
data: "Hello world from STM32!"
---
data: "Hello world from STM32!"
---
data: "Hello world from STM32!"
---
data: "Hello world from STM32!"
---
data: "Hello world from STM32!"
---
```

(c) `$ rostopic echo /chatter`

```
adminub@adminub:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
adminub@adminub:~$ rostopic pub toggle_led std_msgs/Empty --once
publishing and latching message for 3.0 seconds
```

(d) `$ rostopic pub toggle_led`Figure 5.8: Terminal output `ros` command

With this code it was verified that the communication between the board and PC works. It was verified that through the configuration made the latter is able to publish, then send data to the PC and subscribe, then receive input from the PC. Instead, the verification that all motors were correctly receiving data from the board was done by entering a position command directly into the firmware using the function `CAN_TX_AK8080()`. This verified that the function in the firmware

correctly formatted the values to be sent to the motors.

The firmware for controlling the robot was developed starting from this configuration. Inside the robot's firmware, two publishers have been defined, one publishes strings reporting the robot's status to the topic *"statusRobot"*, the second publisher publish the float vector containing the feedback from the motor to the topic *"motorFeedback"*, finally, the subscriber receives the commands to be sent to the engines to the topic *"motorInput"*. In the following chapters will be described the firmware to control the robot.

## Chapter 6

# STM32H7 firmware

In the previous chapters, the general configuration of the entire ecosystem was described, specifically how the engines, the microcontroller and the PC must be configured. The code on the microcontroller will be described in detail in the following chapter, outlining the functions required to interface with the motors and the PC.

The code on the board aims to be as essential as possible, thus having as its only task to mediate communication between the motors and the PC. In fact, the computation performed by the board is as minimal as possible. In this project, in order to have more computational capacity and also to have a better control over the data it was chosen to leave the computational part to the PC/Rpi. The firmware on the board is written in C programming language using the HAL library provided by the ST manufacturer. In order to function optimally, the code constituting the firmware must have well-optimised functions, since the microprocessor's computational capacity is limited. Furthermore, the C language does not easily allow the creation of graphical and real-time plot interfaces. For this reason, it was decided to have the microcontroller perform only the intermediary task, while the control/interface part was written in python on ROS. Using python allows more freedom in software development, and integrating nodes within ROS adds flexibility to the software.

Once the configuration of the STM32 has been completed using the STM32CubeMX software, the `.c` file containing all the configuration previously made can be generated. By generating the file from the IDE used for configuration, a `/src` folder will be automatically created in which the files automatically generated by the IDE will be located, and then `.cpp` files will be imported for use of the board with ROS.

## 6.1 main.c

The complete flow chart of the firmware loaded on the STM32H7 microcontroller for the initial motor control is shown in Fig. 6.1.

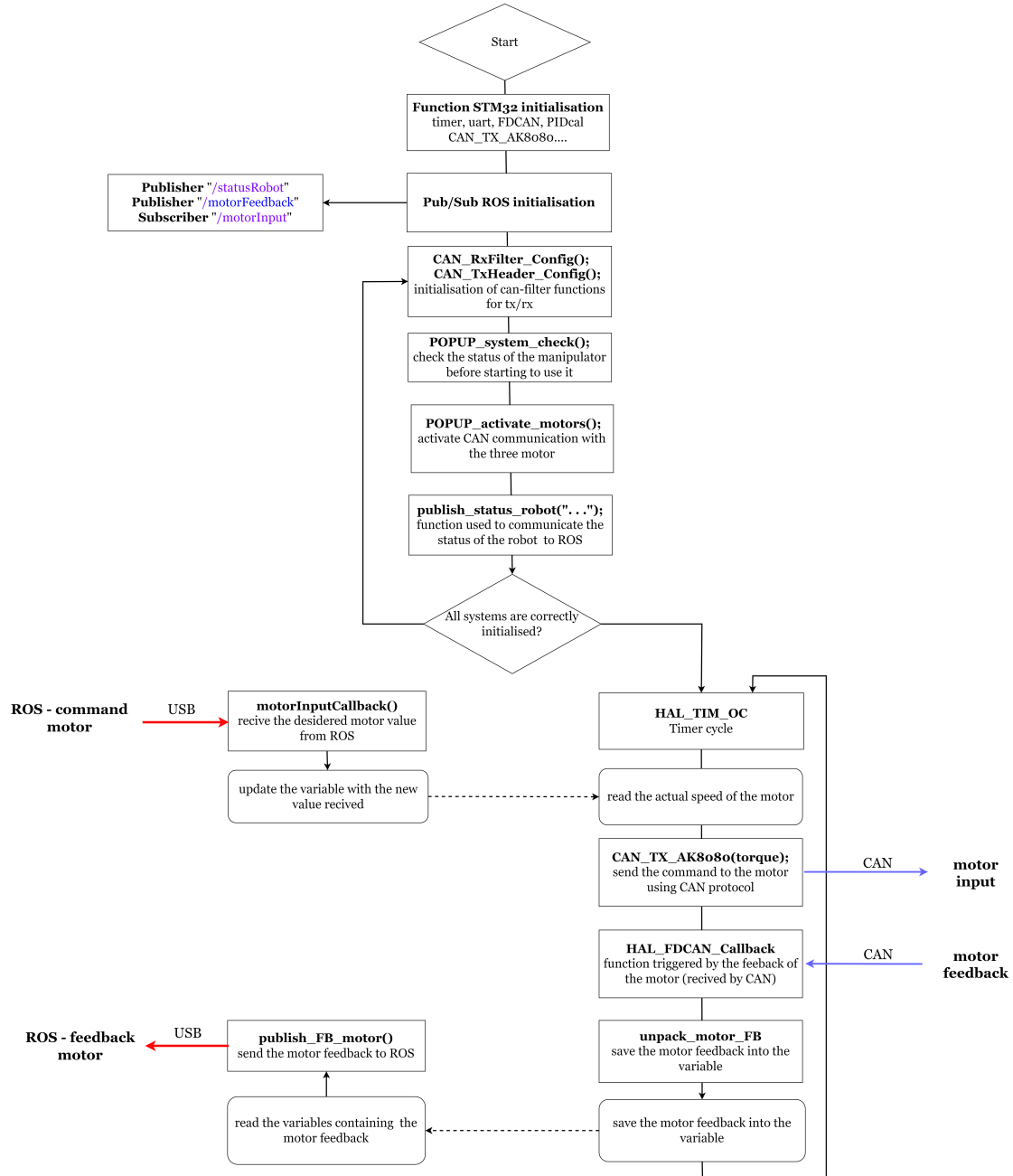


Figure 6.1: Flow chart main.c



As mentioned above, the firmware inside the microcontroller contains the essential code for communicating with the motors and the board, there is also a function that implements a PID; it is implemented on the board to control the motor's torque in order to guarantee better stability.

Timers are essential in the operation of the robot because they call up all functions for communication with the various devices on a regular basis. As mentioned above, two timers were created with different periods. All operations are performed within the timer callbacks.

### 6.1.1 Function description

The firmware flow chart is shown in the Fig. 6.1. The various functions that are called up are described in detail below.

When switched on, the microcontroller calls up the functions for initialising the peripherals, which were previously activated via the STM32CubeMX. Next, the publishers and subscribers for communication with the ROS node are initialised; these are defined in the mainpp.cpp file.

In order to be able to transmit messages over CAN, it is necessary to initialise functions that act as filters, thus filtering both transmitted and received messages. This task is performed by the functions `CAN_TxHeader_Config()` and `CAN_RxHeader_Config()` initialised after the pub/sub of the ROS node.

Once the initialisation of the peripheral devices is complete, the following two functions are called up. `POPUP_system_check()` this function is used to check the status of the manipulator before starting to use it, then the following function:

`POPUP_activate_motors()` it is used to initialise CAN communication with the three motors.

Finally via the `publish_status_robot(. . .)` function; communicated to the ROS node the status of the robot, also reporting any errors in the initialisation of peripherals.

After ensuring that all manipulator components are operational, the main routine that allows the robot to move can begin.

The `CAN_TX_AK8080()` function is used to communicate with the motors through CAN. This function is called up on a regular basis by the TIM8 callback in order to maintain continuous communication with all three motors.

```
void CAN_TX_AK8080(int id, float p_des, float v_des, float kp, float kd,
                  float t_ff){
    . . .
```

---

```

//re-formatting motor input
CAN_tx_buffer[0] = p_int >> 8;           //Motor position 8H
CAN_tx_buffer[1] = p_int & 0xFF;         //Motor position 8L
CAN_tx_buffer[2] = v_int >> 4;           //Motor speed 8H
CAN_tx_buffer[3] = ((v_int & 0xF) << 4) | (kp_int >> 8); //Speed 4L
CAN_tx_buffer[4] = kp_int & 0xFF;         //KP 8L
CAN_tx_buffer[5] = kd_int >> 4;          //KD 8H
CAN_tx_buffer[6] = ((kd_int & 0xF) << 4) | (t_int >> 8); //KD 4L
CAN_tx_buffer[7] = t_int & 0xFF;         // Torque 8L

//Send message over CAN
HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader, CAN_tx_buffer);
}

```

This function is used to format and transmit CAN messages to the engines. Motors can receive commands via CAN if they are prepared according to the criteria stated in the documentation. This function accepts as input the five values that must be **SEND** to the **motor driver** in order to make it move.

- **id motor**: The integer number between 1 and 3 denotes the motor to which the command is to be sent.
- **desired position** [rad]: is expressed in radians and shows the position that the motor should reach (if the same position is sent more than once the motor remains stationary in that position)
- **desired speed** [rad/s]: specifies how fast the motor should move; if the speed is zero, the motor remains motionless
- **kp**: is a parameter required for the motor to move into position (below is a graph showing the relationship between the KP value and the displacement angle)
- **kd**: is a number between 1 and 5 that must be greater than zero to allow for motor speed variation.
- **torque** [Nm]: is a parameter that is converted to current and allows motor movement

To accomplish the movement appropriately, the motor must receive the data within a vector with the properties indicated on the datasheet. This function generates and transmits the vector to the motor with the commands.

The publisher on the ROS node sends the data given as input to the `CAN_TX_AK8080()` function.

Since when the motor driver receives an input, it automatically sends back the motor feedback via CAN. The `unpack_motor_FB()` function performs the task of reformatting the data frame holding the engine feedback.

```
void unpack_motor_FB() {  
    //save motor feedback into C variable  
    int motor_id = CAN_rx_buffer[0];  
    int pos_int = CAN_rx_buffer[1] << 8 | CAN_rx_buffer[2];  
    int vel_int = CAN_rx_buffer[3] << 4 | CAN_rx_buffer[4] >> 4;  
    int current_int = (CAN_rx_buffer[4] & 0xF) << 8 | CAN_rx_buffer[5];  
}
```

This function reallocates the individual bytes of the data frame into variables that are usable by C code.

The data frame **RECEIVED** from the **driver** contains the following motor information:

- **id motor:** (value between 1 and 3) shows which motor provided the feedback.
- **position:** represents the engine's current location in radians
- **speed:** represents the engine's current speed in radians/second
- **current:** indicates the amount of current used by the motor to keep the position stable.

This function, unlike the preceding one, gets recalled into the FDCAN's callback. The FDCAN is set to interrupt because motor feedback is not constantly received by the board but is only delivered when a command is given as input to the motor, this configuration improves the board's performance; recalling the `unpack_motor_FB()` function only when necessary and not periodically.

Finally, motor feedback is transmitted to the ROS node via a microcontroller publisher.

Using these two functions (`CAN_TX_AK8080()` and `unpack_motor_FB()`), the microcontroller can fully control and monitor the three motors, allowing CAN communication to be built. The functions required for communication with the ROS node on the PC are defined in the firmware's `mainpp.cpp` file.

## 6.2 mainpp.cpp

The preceding section described the functions that enable CAN communication with the motors. Instead, the following section will go through the roserial protocol, which is used to communicate between the microcontroller and the PC. As mentioned in the previous chapter, the board's code includes a number of C++ libraries necessary for implementing functions utilising ROS.

To have full control and monitoring of the POPUP robot data on the microcontroller, two publishers (one transmits motor feedback to the ROS node, while

the other publishes the robot status) and one subscriber (which receives motor commands supplied by the ROS node) are defined.

The following publishers/subscribers were defined in the mainpp.cpp file, inside the firmware's folder.

- **Publisher #1:** The following publisher send a string-type message on the topic *"/statusRobot"*. The following messages are sent from the microcontroller to the ROS node, and contain information on the status of the manipulator and the operation of the transmission, reporting any errors if necessary. Initially, messages sent from the microcontroller to the board were read from the terminal using the command:

```
$ rostopic echo /statusRobot
```

Subsequently, a window was added to the GUI showing the status of the manipulator.

- **Subscriber #1:** This subscriber receives float vectors on the topic *"/motorInput"*. The received vector contains the five values that will be delivered through CAN to the motor driver. These parameters are set and transmitted by a publisher (linked to the topic *"/motorInput"*) on the ROS node.
- **Publisher #2:** Finally, this publisher provides engine feedback to the ROS node's subscriber. The message is a float vector that is sent to the topic *"motorFeedback"*. The vector transmitted will have the following data: id motor, actual position, actual velocity, and actual current consumed by the motor.

The code below shows the initialisation of publishers/subscribers described earlier. It can be observed that in the case of the subscriber, a function indicating the operations to be performed once the data has been received is required, whereas for the publishers, simply the vector to be delivered is required.

```
ros::NodeHandle nh;

void motorInputCallback(const std_msgs::Float64MultiArray&
                        motor_input_value);
//Publisher used to sent info to ROS node
std_msgs::String str_msg;
ros::Publisher chatter("statusRobot", &str_msg);
//Publisher used to sent motors feedback to ROS node
std_msgs::Float64MultiArray motor_fb_value;
ros::Publisher pub("motorFeedback", &motor_fb_value);
//Subscriber used to receive motors input from ROS node
ros::Subscriber<std_msgs::Float64MultiArray> sub("motorInput",
                                                &motorInputCallback);
```

### 6.2.1 Publisher/subscriber functions

The following section will go through in detail the functions specified in **mainpp.cpp** that will be called inside **main.c**, allowing communication between the microcontroller and the PC. The figure below shows an overview of the functions that are called up by the various devices. In this section, all functions referring to the microcontroller have been described as reproduced in the figure 6.2. Also shown are the topics on which communication takes place as well as the messages contained.

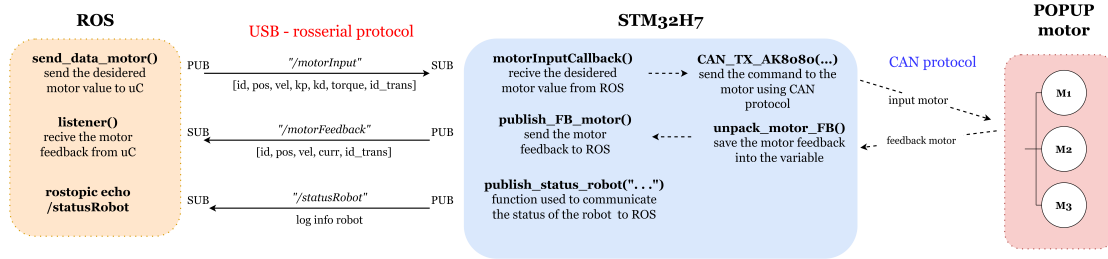


Figure 6.2: The following figure summarises all the topics and message types that have been implemented to control the robot

The `motorInputCallback()` function is linked to the subscriber, sub. It is executed whenever the microcontroller receives a new message on the topic `"/motor-Input"`.

The publisher on the ROS node transmits a float vector with six values (five values that must be supplied to the motor driver in order to make it move plus one values necessary for sync message). This function copies the incoming data into variables defined inside `main.c`. These received values will then be passed as input to the function `CAN_TX_AK8080()`, which will deliver the command to the engines via CAN.

```
void motorInputCallback(const std_msgs::Float64MultiArray&
                        motor_input_value) {
    id_can_tx_8080 = motor_input_value.data[0]; //id
    value_can_tx_8080[0] = motor_input_value.data[1]; //pos
    value_can_tx_8080[1] = motor_input_value.data[2]; //speed
    value_can_tx_8080[2] = motor_input_value.data[3]; //kp
    value_can_tx_8080[3] = motor_input_value.data[4]; //kd
    value_can_tx_8080[4] = motor_input_value.data[5]; //curr
    id_transaction = motor_input_value.data[6];

    nh.spinOnce();
}
```

After the function `unpack_motor_FB()` has been executed, this function is invoked

within `main.c`. It has the task of publish to the ROS node the feedback of the engines. Publish the values on the topic *"motorFeedback"*.

The values sent to the ROS node are as follows: motor id, actual position speed and current of the motor plus the variable `"id_transaction"` which is used to sync messages.

```
void publish_FB_motor(float id, float pos, float speed, float curr)
{
    motor_fb_value.data[0] = id;
    motor_fb_value.data[1] = pos;
    motor_fb_value.data[2] = speed;
    motor_fb_value.data[3] = curr;
    motor_fb_value.data[4] = id_transaction;

    pub2.publish(&motor_fb_value);
    nh.spinOnce();
}
```

In addition to the motor status data, both vectors contain an additional variable called **id transaction**. This variable, which is a randomly generated number, is added to each vector containing input to be sent to the engine. The following variable is used to differentiate a message from the next one, in this way it is possible to check if the message sent by the PC has been correctly received by the board. In fact, as described above, each time the engine receives a command it automatically sends back a feedback, through the publisher this data is also sent to the PC, and the variable `id transaction` is added to the vector. In this way, if the transaction id received by the board is equal to the transaction id sent by the PC, it means that the data transmission has taken place correctly without losses.

This variable was added to distinguish one transmission from the next between the PC and the microcontroller (send motor input, receive motor feedback). Each time the ROS node sent a message to the microcontroller the value of this variable is updated with a random number. This variable is added to the vector containing the parameters to be sent to the engines, so that each command vector is uniquely defined. Then, when the microcontroller sends the message containing the motor feedback to the ROS node, it will add the random number received with the motor input to the vector. In this case, if the microcontroller's feedback contains the same **id transaction** value as the command published by the ROS node, the transmission was successful.

Furthermore, by using this variable, it is possible to transmit the next message only after the previous one is correctly received, eliminating data overlapping or loss. Finally, if the engine feedback is not received within a specified time frame, the transaction related to that message is deemed KO, and the next message is sent.

Finally, the following function publish on the topic *"/statusRobot"* a message of type text reporting information. This function is called up within the main.c and sends information on the programme's progress, reporting any errors, the task of this function can be compared to the function performed by a logger.

```
void publish_status_robot(char sample[])
{
    str_msg.data = sample;
    chatter.publish(&str_msg);
    nh.spinOnce();
}
```

This chapter has laid out and discussed all of the functions on the microcontroller, which is responsible for mediating communication between ROS and the motors. The functions used in the ROS node developed in python will be detailed in the following chapter.





## Chapter 7

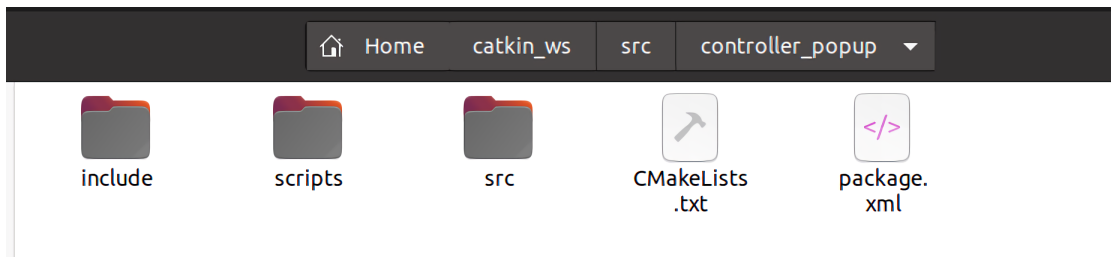
# ROS workspace

In the previous chapters, the entire microcontroller firmware, the configuration required for communication both with the PC via USB and with the motors via CAN was described. Now it is necessary to explain the ROS nodes, used to control the robot.

As mentioned above, the ROS Noetic middleware was used in this project, which requires Ubuntu 20.04 as its operating system. For convenience, a virtual machine with Ubuntu 20.04 was installed on the PC with Windows 10 as its native operating system. Once the installation of ROS is complete, the **controller popup** package was created, which contain the nodes for controlling and communicating with the robot.

**Packages:** *The ROS packages are a central element of the ROS software. They contain one or more ROS programs (nodes), libraries, configuration files, and so on, which are organized together as a single unit. Packages are the atomic build and release items in the ROS software [2].*

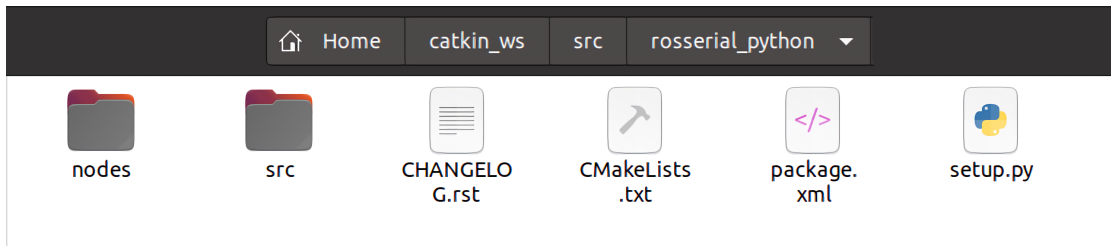
The image [7.1](#) shows the folders inside the **controller popup** package, the folder **/scripts** is the most important one, it contains the python files that define the nodes. Several node have been created to allow different control of the robot, for example there are scripts created only to measure the latency of communication, while others allow only the control of the motors in position through the graphical interface.

Figure 7.1: **controller\_popup** package's folder

The second package inside the ROS workspace is the package referred to the communication with Rosserial. This package is automatically placed inside the workspace once the installation of roserial is complete. Inside the nodes folder is the python file `serial_node.py` that is called up by the command:

```
$ rosrund roserial_python serial_node.py /dev/ttyACM0 _baud 115200
```

The following file (`serial_node.py`) defines the USB communication parameters, running the file from Ubuntu's terminal enable the communication with the microcontroller. As shown in the figure 7.3, this command reads the publishers and subscribers written on the microcontroller connected via USB to the PC.

Figure 7.2: **roserial\_python** package's folder

```
adminub@adminub-virtual-machine: ~
adminub@adminub-virtual-machine:~$ sudo chmod 666 /dev/ttyACM0
[sudo] password for adminub:
adminub@adminub-virtual-machine:~$ rosrund roserial_python serial_node.py _port:=/dev/ttyACM0 _baud:=115200
[INFO] [1662544118.248080]: ROS Serial Python Node
[INFO] [1662544118.252624]: Connecting to /dev/ttyACM0 at 115200 baud
[INFO] [1662544120.367965]: Requesting topics...
[INFO] [1662544121.015601]: Note: publish buffer size is 1024 bytes
[INFO] [1662544121.017232]: Setup publisher on statusRobot [std_msgs/String]
[INFO] [1662544121.023829]: Setup publisher on motorFeedback [std_msgs/Float64MultiArray]
[INFO] [1662544121.032511]: Note: subscribe buffer size is 1024 bytes
[INFO] [1662544121.033592]: Setup subscriber on motorInput [std_msgs/Float64MultiArray]
[INFO] [1662544121.039979]: Setup subscriber on PIDvalue [std_msgs/Float64MultiArray]
```

Figure 7.3: Terminal output \$ `roserial_python` command

After the creation/installation of this two packages (referring to the book [2]), simple nodes were developed in python in order to verify correct operation of the publisher and subscriber. It was previously illustrated, how the roserial communication work (using the terminal command), i.e. by reading the messages in the topic or by publishing messages from the command line. The following will show how the same task can be performed automatically by creating a python node.

### 7.0.1 Basic publisher/subscriber functions on ROS

The following code is used to initialise a publisher which publishes on the topic `"/motorInput"`. The publisher is a `Float64MultiArray` type, which means that message on the topic is a float vector of N elements. This publisher is used to send commands to the motor, in fact, as can be seen from the example, the data contained inside the message corresponds to the commands required to make the motor move. Thus, the commands to be sent to the engine are defined with this function; will then be integrated into the robot control code.

```
import rospy
from std_msgs.msg import Float64MultiArray, MultiArrayDimension

def talker():
    pub = rospy.Publisher('/motorInput', Float64MultiArray)
    rospy.init_node('publish_to_controller', anonymous=True)
    rate = rospy.Rate(1) # 10hz

    msg = Float64MultiArray()
    msg.data = [3.0, 1.57, 0.0, 15.0, 0.0, 0.0]

    while not rospy.is_shutdown():
        rospy.loginfo("sent command")

        pub.publish(msg)
        rate.sleep(1)
```

Similarly, the following function implements a subscriber subscribed to the topic `"/motorFeedback"`. When a message is received (again, a float-type vector containing the four values on the motor feedback is received), the callback function is executed, which in this case prints out the received data.

```
import rospy
from std_msgs.msg import Float64MultiArray, MultiArrayDimension

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + 'I heard %s', data.data)
    rospy.loginfo('I heard %s', data.data)
```

```
def listener():
    rospy.init_node('listener', anonymous=True)
    rospy.Subscriber("/motorFeedback", Float64MultiArray, callback)
    rospy.spin()
```

Combining the two functions written above, there is a complete cycle of communication with the engine. Described more in detail, the user define the command to be sent to the motor with the GUI on ROS. The following values published with the ROS node will be received by the subscriber present on the microcontroller, which will copy the values and format them into data frames and then send them over CAN.

Then, the driver automatically sends back to the microcontroller the feedback from the engine; by performing the reverse process the board reformats the data frame into variables, in this way the publisher present on the board will publish the data that will be received by the subscriber present on the ROS node. Once the feedback of the command sent to the engine is received the transaction related to that command is closed. Similar function was developed for receiving messages referring to the status of the robot, so a subscriber was created that prints the information/error messages sent by the microcontroller on the screen.

## Chapter 8

# ROS node development

### 8.1 Position controller

As a first method for controlling the motors, it was decided to use position control. In order to implement this type of control, the motors AK-8080 must be received the data frame with the following information :[motor id, position, speed, kp, kd, torque]. To have only position control, it is necessary to specify only three of the six parameters listed above, which are: motor id, position and kp.

During the tests, it was realised that a different kp value is required for each shift of the motor angle; if, for example, the motor must reach a position closely to its actual position, the kp value must be very high otherwise the motor does not move. Conversely, if the kp value is too high, the motor will reach the specified position more precisely but with greater overshoot.

It was therefore necessary to understand the relationship between the position of the motor (expressed in radians) and the correct kp value, which allows the motor to move smoothly, with as little overshoot as possible. The following relationship (motor displacement angle vs kp value) was found by creating a python script.

Launching the script a displacement angle (ex. 1.57 rad) must be defined and automatically a for cycle present in the script increase the kp value sending the same position. The kp value is increased until the motor return non-zero feedback, this indicates that it did not move.. Once the motor reached the position specified, the kp value that allowed the movement was saved in a .txt file. By repeating this process for different angles of movement, it was possible to create the graph shown in the figure [8.1](#).

The script has the only purpose of finding the relationship between the displacement of the motor in position and the value of  $k_p$ . It consists of two nested for loops, the first one cycles over all possible  $k_p$  values (from the motor datasheet the  $k_p$  value must be between 0 and 500), the second for loop instead changes the starting position of the motor, but keeping the same displacement angle, this is done in order to have more data to be analyse.

Consulting this graph, it is then possible based on the desired displacement angle, to set the correct  $k_p$  so that the move takes place with as little overshoot as possible.

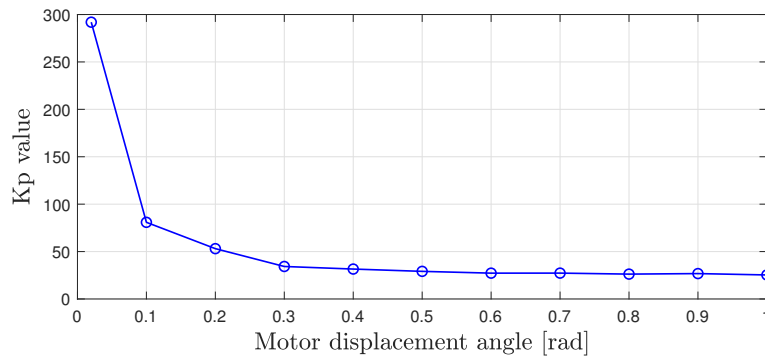


Figure 8.1: Graph -  $k_p$  value

In order to control the motor specifying only the position, it is therefore necessary to send a single message from the PC to the microcontroller containing the desired position, the  $k_p$  relative to the position and the id of the motor. Sending the same position value to the motor once it has already reached it, the motor remain stationary. Conversely, for speed or torque control, the motors must receive the respective command at constant intervals. Next, it will be described how commands are sent from the ROS node to the microcontroller for all three motors at constant intervals.

## 8.2 One motor control - ROS node

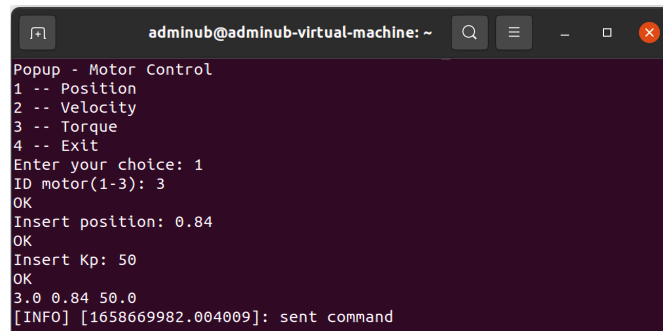
The main functions of ROS nodes and how they communicate with the microcontroller were explained in the previous chapter. This chapter gives a full description of how the robot is controlled.

### 8.2.1 Terminal Interface

As mentioned above, various scripts for controlling and testing the robot were created within the **controller popup** package. The first node that has been developed is the one for controlling a single motor, specifying only the final position. The user specifies which motors to move and what position it must reach, using the menu printed on the Ubuntu's terminal after the node boots. The menu that is proposed to the user to control the motor is shown in Fig. 8.2.

To control a single motor in position, the driver requires the following data: the id of the motor to which the command will be sent, the desired final position to be reached by the motor, and finally the value of the kp tied to the position in order to have overshoot low as possible, all other variables will be automatically set to zero.

Using this script, it is possible to send the selected engine more than one position at time. The following script calls up the `send_data_motor()` function, which publish to the microcontroller the data entered by the user, and once the motor reaches the sent position, the ROS node re-proposes the menu. In this way a new position can be sent to the motor. In addition, the following script checks the validity of the values entered by the user, thus avoiding sending incorrect inputs to the robot's motor



```

adminub@adminub-virtual-machine: ~
Popup - Motor Control
1 -- Position
2 -- Velocity
3 -- Torque
4 -- Exit
Enter your choice: 1
ID motor(1-3): 3
OK
Insert position: 0.84
OK
Insert Kp: 50
OK
3.0 0.84 50.0
[INFO] [1658669982.004009]: sent command

```

Figure 8.2: Motor control - terminal interface

### 8.2.2 Graphical Interface

Later, to improve and facilitate sending commands to the engine, a graphical interface was created shown in Fig. 8.3. The input sent to the engine can be edit by moving a slider.

The graphical interface was created using the python tkinder library. It is the standard python interface to the Tk GUI toolkit, and is python's de facto standard GUI.

In order to move the motor the following operation must be performed: First of all a motor must be selected from the list displayed on the GUI, then the position and the equivalent  $k_p$  can be set-up moving the slider; once the values have been defined, the vector containing all the data will be sent to the microcontroller and then to motor by pressing the Move! button at the top. In the following GUI, position is represented in rad, speed in rad/s and torque in Nm. Once the movement has been executed, the feedback values sent by the motor will appear at the bottom of the window.

With this GUI it is also possible to send more than one command. An advantage of using the graphical interface for sending commands is the control over the data sent, the sliders have a maximum and a minimum limit, prevents the sending of incorrect commands.

Through this graphical interface, however, it is only possible to select one engine at a time and therefore it is not possible to send the command to three engines at the same time. This script was used in the development phase to understand, manage communication between the various devices, and to check the performance of the individual motors. A script is then implemented that allows commands to be sent to all three engines simultaneously.

With this node it is only possible to control and send the position of the motor, since to control motor in speed or in torque it is necessary a continuous communication between the devices. In order to send a speed or torque command, continuous communication with the motor is required, so the speed or torque command must be sent to the motor at a certain frequency, whereas if only the position is specified, only a command indicating the end position to be reached is required. Initially, the following graphical interface was only used to send a single message to the microcontroller, after pressing the Move! button, this only allowed position control, subsequently messages will be sent from the PC to the microcontroller at constant intervals, thus allowing controlling the speed and the torque of the motor.

## 8.3 Continuous communication

Once defined and verified the functioning of the whole ecosystem. In fact, using the GUI, it was possible to verify the operation of the three motors separately, making sure that all of them were able to receive the commands and send-back the correct feedback.

Starting with this node, the next step was to write a script that continuously



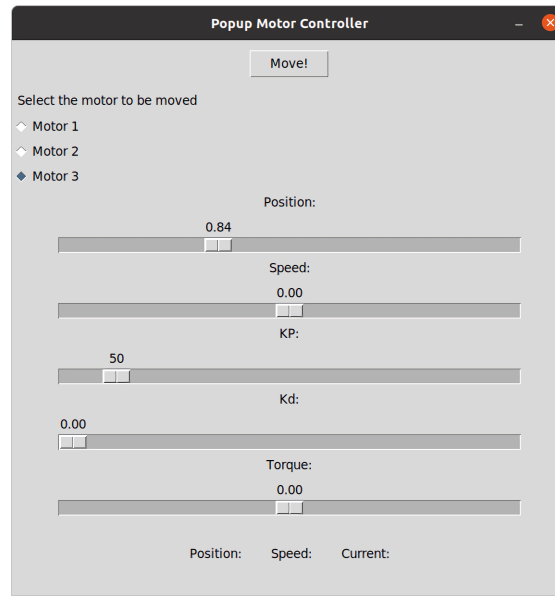


Figure 8.3: Motor control - Graphical interface

communicated with the microcontroller so that it could control the speed or the torque of the motor.

The most complicated part of having continuous communication with the microcontroller and consequently with the motors was the synchronisation of the various messages between the devices.

In fact, the message published, via USB, by the ROS node must be read by the microcontroller and then sent via CAN to the correct motor, after which the feedback from of motor must be sent to the ROS node passing through the board.

This cycle of sending and receiving messages must have a certain frequency and must be synchronised between the various devices, so that when an input is sent from the node, the feedback of the given input is received within a short time.

A problem that can be found is the possibility of the message to be lost or overwritten by the next command. This can happen, since communication via CAN is half duplex, so it is only possible to receive or transmit data, but the two operations cannot be done simultaneously.

Furthermore, in both communications there is a bandwidth limit that can be occupied, so the timer can send up a max value of bits/s, this limit is present in both USB transmission and CAN transmission.

Furthermore, as will be discussed further below, a PID was added to the

firmware, which permits simultaneous control of the motors and so improves the robot's stability. It is only present inside the code on the board, but it must be taken into account because it sends at constant intervals (TIM2 period) commands to the motors based on their feedback speed. This control occupies CAN bandwidth.

This section will then describe the various timer periods that allow communication with both the ROS node and the motors, avoiding data loss and synchronising the various commands with their respective feedbacks

## Development

At this point it is necessary to develop continuous communication with the three motors. However, the CAN protocol only allows one command at a time to be sent to the selected motor via ID, and the feedback is sent back over the same communication line.

The flow chart shown in the figure summarises the ROS node operations.

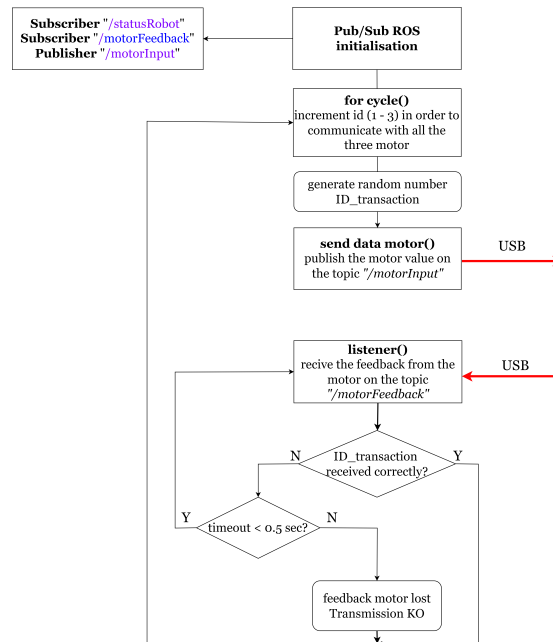


Figure 8.4: Flow chart ROS node

Originally, the commands `rospy.rate`, `rospy.spin` present in ros were used to continuously publish message. `rospy` provides a `rospy.rate` convenience class which makes a best effort at maintaining a particular rate for a loop.

`r = rospy.Rate(10) ## 10hz` , with this command a message is published every 1/10 second.

With this command there is no control over individual messages and no way to know if they were received correctly by the engine or if they were lost due to communication problem. Setting the rate to a high value allows continuous communication with the motor without significant data loss, but this method does not achieve the maximum communication speed between devices.

In order to have constant communication at regular intervals at high speed with all three motors, a for loop inside the node was used. The idea is to cycle the commands to the motors, starting with motor 1, this is done with a for loop. It increases the id at each round, so starting from 1 the node publishes the command that will be received by motor 1, on the second round the id is increased to 2, the command for motor 2 will then be published, the loop increases the id so as to send a command to all motors present, in this case the maximum id is 3. Once the command for the last engine is published, the cycle starts over from 1.

However, with the above idea, all messages are lost because there is no synchronization between different messages and the for loop is faster than data transfer. It was therefore necessary to develop something allowing synchronisation between the various messages.

It was decided to assign a randomly generated number (called **ID transaction** in the vector) to each command sent. The next command will not be sent until the node receives a response from the engine containing the same random numbers in the vector.

In this way each transmission (command sent to the motor and feedback received by the PC) is uniquely defined and if the feedback for the command sent does not arrive within a certain time interval an error is reported and the following message is transmitted.

Therefore, the for loop above must wait for feedback from the motor to be received before incrementing the value of ID and continuing to send commands. In this way, the next command is sent as soon as the response arrives, so there is no waiting time and thus maximum transmission speed can be achieved. This method frees up bandwidth and also ensures that no messages are overlap when sending or receiving messages.

The code shown below represents a for loop that can continuously send messages to all three engines. The function `send_data_motor()` recalled into the for loop, publishes the motor value on the topic `"/motorInput"`. The data sent were specified

using a graphical user interface.

```
def motor_input(id):
    while True:
        for i in range(1, 4, 1):
            send_data_motor(i, pos_motor[i], speed_motor[i], kp_motor[i],
                            kd_motor[i], torque_motor[i])
        print("-----")
```

Instead, the following function continues looping until it receives the **ID transaction** of the command sent (for loops cannot increment). Motor feedback is received in the vector `fb_motor` which also contains a number that uniquely identifies the communication. If this number is not received within 500 ms, an error is reported and the next command is sent.

```
timeout = time.time() + 0.5 # 500ms from now
while(fb_motor[4]!=id_trans):
    if(time.time() > timeout):
        print("Transmission KO")
    break
```

So, with the above two functions, the command is sent to the next motor as soon as the motor's feedback is received, so messages can be sent periodically to the motors with maximum frequency.

The `send_data_motor(...)` function calls a publisher which is used to send a message to the `/motorInput` topic. Messages are received by the microcontroller participants and sent to the motors via CAN.

Motor feedback follows a similar but opposite path, with data sent via the topic `/motorFeedback`. A subscriber function resides in his ROS node to receive feedback data and copy it to the `fb_motor[]` vector.

### 8.3.1 GUI motor control

The image 8.5 shows how the originally displayed graphical user interface has changed. Each motor has two sliders, one for setting the position and one for setting kp.

As can be seen from the image, starting the node initiates continuous communication with the three engines. Having therefore implemented continuous communication with the engines, it is possible to set the final position of the engines via the GUI and move the robot by pressing the Move! button

The engines will move almost simultaneously in order to reach the set position. Once the desired position has been reached, it is possible to set a new position

for all three motors. Sending the same position once it has reached it, the motor remains stationary at that point.

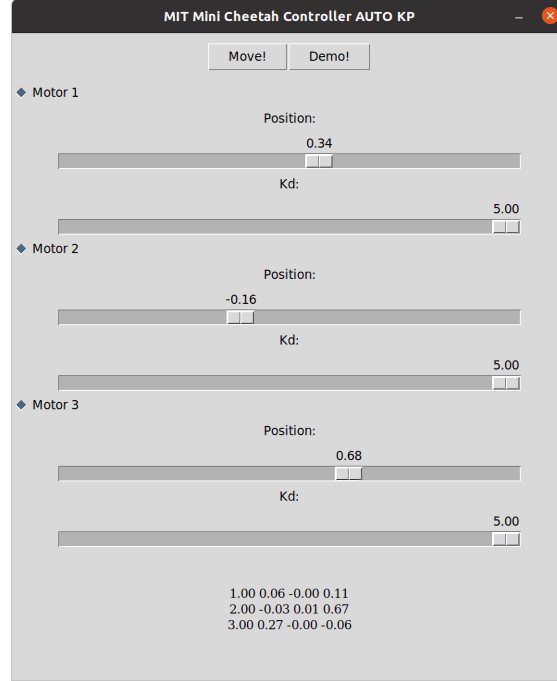


Figure 8.5: GUI - Three motor position controller

A button has been added to the GUI: Demo! By pressing the following button, the motors move to different positions, remaining in each of these for 10 seconds. The following buttons have been added for demonstration purposes so that the user does not have to manually send new positions to the motors. The positions of the motors are saved in a file; the function called by pressing the Demo! button reads the file line by line and sends the read positions to the microcontroller via the publisher.

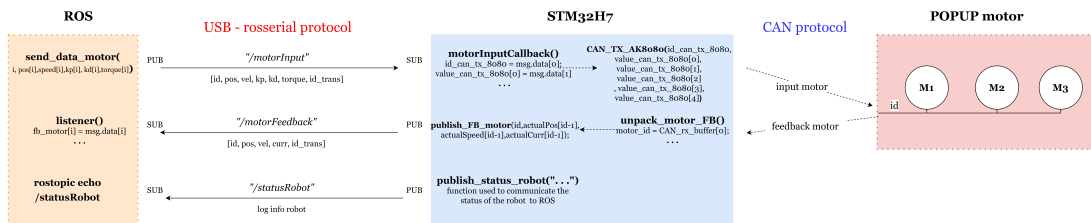


Figure 8.6: Scheme communication - one motor

## 8.4 Three motor control - ROS node

In the previous part, it was discussed the various nodes in ROS, how the user sends commands to all three engines, how commands are read and processed by the board, and how synchronization between devices occurs.

At each cycle the command for the respective motor is published, so starting from motor 1 the command is sent from the PC to the microcontroller, then waited for the corresponding feedback, once the feedback from motor 1 is received by the PC the next command is sent to motor 2 and then waited for the feedback, the same thing is done for motor 3. Sending and receiving each command can take some time, the transmission between microcontroller and ROS node lengthens the period between the sending of commands.

It was therefore decided to reduce the number of communications between the PC and the microcontroller in order to increase the speed of communication between the devices. A duplicate of the previous publisher and subscriber was created, but the key difference between them is the amount of data sent in each communication and the frequency of those communications. In fact, to increase speed, commands are sent to the motors in a single vector, just as the feedback from the motors is stored in vectors and published all at once.

Instead of sending a command and receiving the corresponding feedback, in this case a single vector containing the commands for all three motors is published by the ROS node, then the microcontroller will publish a vector containing the feedback of all three motors. This solution increases the speed of communication between PC and microcontroller, avoiding waiting for feedback from each individual motor.

Again, communication between the various devices must be continuous in order to ensure constant control of the motors. The ROS node constantly publishes a vector containing commands to be sent to all three engines using a for loop. Synchronization between one command packet and the next is done using ID transaction. A vector containing the motor feedback must be received from the PC before issuing the next command vector. The same while loop that exists in ROS nodes and used to communicate with individual engines is reused here. In the previous mode, a different ID transaction is assigned to each individual command; in this case, a different ID transaction is assigned to each command packet (the message containing the command for all the three motors).

The figure [8.7](#) represents the topics on which the various messages are published,

and what information is contained inside them. As you can see in the image, the message contains 3 ids related to 3 different engines and 5 commands required for each engine. The bandwidth occupied for communication between the node and the microcontroller will therefore be greater than in the previous case, since the message must containing the data of the three motor, so the vector published have greater dimension. It is necessary to set the baud rate carefully so as not to have any data loss.

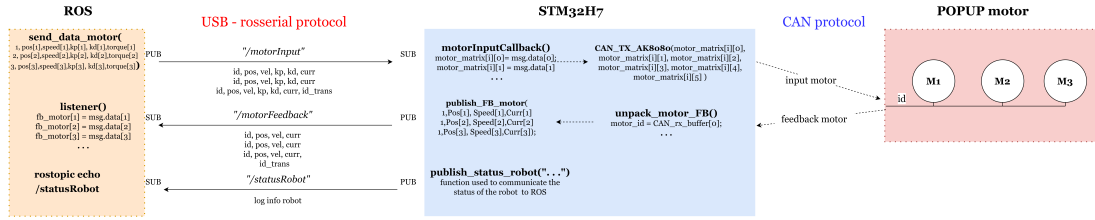


Figure 8.7: Scheme communication - three motor

A message containing the commands to be sent to the three engines is published by the ROS node. The board receives the message published by the ROS node on the topic `"/motorMatrixInput"`. Unlike previous modes, received commands cannot be immediately executed by the engine and must be processed. Since the motors are serially connected via CAN, only one command can be sent at a time.

Therefore the received message containing all motor commands is unpacked and the various motor commands are split into the different variables; in order to be sent via CAN one at a time. The same is done for the feedback of the motors, sending a command to the motors via CAN one at a time so that the feedback is received one at a time.

Unlike previous modes, received feedback is not immediately published, instead feedback from a single motor is stored in a vector. If the vector contains all three motor feedbacks, it will be published on the `"/motorMatrixFeedback"` topic to which the node is subscribed. As soon as the vector containing the motor feedback is received, the PC will send a second command packet. To check if the received feedback is related to the sent command packet, check if the ID transaciton sent in the command packet is the same as the one received in the message containing the motor feedback.

A negative aspect of this mode, compared with the previous one, is the possibility of message loss. In the previous case if the message was lost, only the command or feedback referring to a single motor was lost, on the contrary if in this mode one message is lost, the commands of all three motors or the three motor feedbacks are lost, so there is a greater loss of information and less control over the motor commands, since it is not possible to tell which motor had problems.

The graphical interface shown above in Fig. 8.5 is employed for engine position control, despite the fact that the content of the messages has changed. The user via the sliders sets the final position of the three different motors, pressing the Move button! these values are inserted inside the vector that will be published on the topic *"/motorMatrixInput"*. The GUI developed for the previous mode was reused.

In order to compare the two modes (either sending one command from the PC to the microcontroller or sending all three commands), a script was developed to compare these two modes over time. These scripts and data analysis are described in the next chapter.



## Chapter 9

# Message latency comparison

The previous chapter explained how these two types of communication between the PC and the microcontroller work. This chapter compares the two methods measuring the time required for the communication. Several functions have been added both in firmware and in ROS nodes to calculate message transfer latency. A script was then developed to calculate the transmission times at various points and save the data to a file.

From these files, graphs of transmit and receive times between various devices were created. This makes the comparison of the two modes of control motor more clear.

### 9.1 One motor at time - (single mode)

As mentioned earlier in this communication method, ROS nodes send a message on the `/motorInput` topic containing only **single-motor commands**. This command is received by a subscriber on the board and sent over CAN to the motor specified via the ID variable.

Then the motor driver send via CAN the motor feedback to the board; the publisher present on the board will send the data to the PC on the topic `/motor-Feedback`.

To compare the two different communication modes, some functionality (to measure elapsed time) has been added to the firmware and ROS nodes.

The firmware and the ROS node has been modified to measure the time spent transferring data between different devices. In the figure 9.1 the letters indicate the different transmissions on which the times were measured.

The first measurement data transmission takes place between the microcontroller and the motor. Specifically, the time is measured from when the subscriber

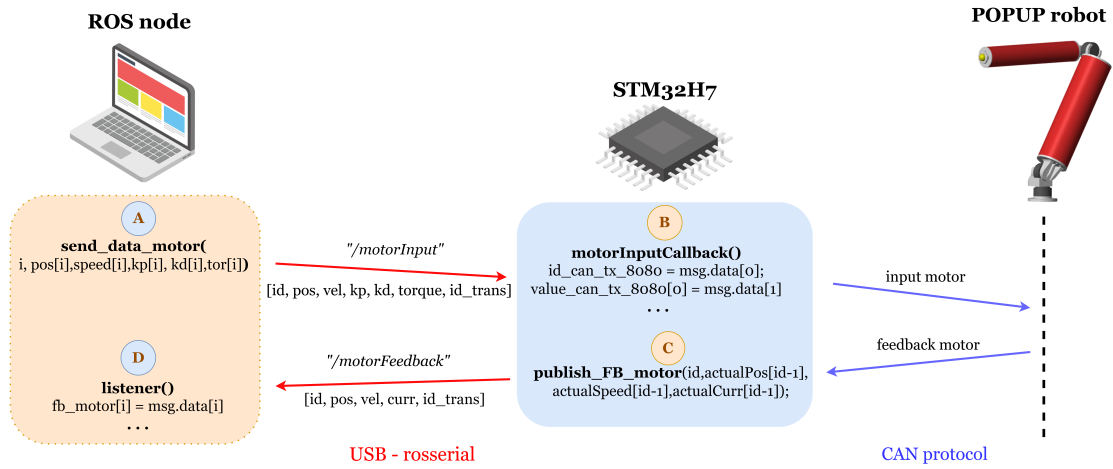


Figure 9.1: Scheme measured transmission time - single mode

(present into the MCU) receives a command from the ROS node to when the MCU publisher publishes the response associated with the received command. In the Fig. 9.1 the following path has been indicated with the letters B-C. In order to be able to measure transmission times, functions have been added inside the publisher/subscriber function. Into the `motorInputCallback()` function, the `getCurrentMicros()` function is reclaimed.

```
void motorInputCallback() {
    tick_motor_input = getCurrentMicros();
    . . .
}
```

As described earlier, the `motorInputCallback()` function is recalled by the microcontroller when the ROS node publishes a message on the `"/motorInput"` topic. The `getCurrentMicros()` function when called provides a tick value in microseconds. By placing the function inside the `motorInputCallback()` function, its value is stored in the `tick_motor_input` variable when a new message on the topic is received.

```
void publish_FB_motor(float id, float pos, float speed, float curr) {
    . . .
    motor_fb_value.data[5] = tick_motor_input;
    motor_fb_value.data[6] = getCurrentMicros();
    . . .
}
```

Activating this function inside the `publish_FB_motor()` function will get the ticks at which the feedback is sent from the MCU to the node.

The two ticks, taken at the two different instants (referring to the figure, the instant taken at the point indicated with the letter B and the one indicated with the letter C), are sent to the ROS node with the feedback from the motor. The script implementing the ROS node then performs a subtraction of the two ticks and the result is saved in a .txt file from which the charts are derived. This operation determine how long it takes the board to communicate with the motor (send commands and get responses).

### 9.1.1 Time analysis - script python

A new python script was developed to measure the communication time between different points. Starting with the script used for continuous communication with the three engines, it has been modified to allow measuring and saving different ticks during transmission.

Upon launching the script, a GUI appears, developed using python's tkinder library, on which there is a Time! button as depicted in the figure 9.2. By clicking on the Time! button, ninety commands are published by the ROS node, which will be received by the microcontroller and then sent to the motors. The data transmission time for each message is measured and saved in a file.

When the script is started, it calls the `send_data_motor()` function which will publish a command for an engine (ex. engine 3). After publishing, the `time.ns` function (present in python library) stores the ticks in the `time_tx` variable, in this way the this variable will contain the time at which the command was published by the ROS node.

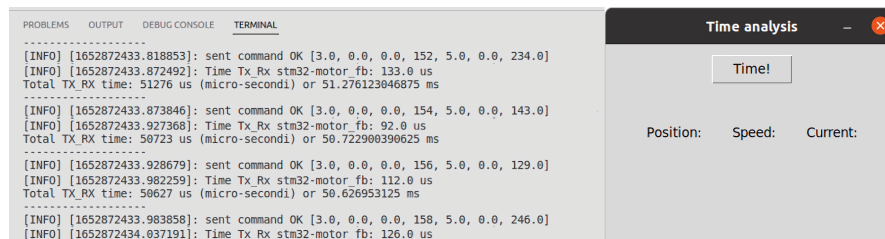


Figure 9.2: Terminal output - ROS node time analysis

The message published by the python node is then received by the microcontroller and triggers the `motorInputCallback()` function present in the firmware. As mentioned above, this tick is stored in a variable.

After that, the microcontroller will send the command to the motor and the time at which the motor feedback is received will be saved and published. Finally,

the `time.ns` function in the python script will save the time at which the python node receives motor data feedback in the variable `time_rx`. After saving the instant at which the node receives the motor feedback, the cycle restarts by sending a new command and re-measuring all transmission times.

The above cycle is repeated 90 times to get more data. All ticks of various communication points are saved in a file from which charts are obtained.

The graph in the Fig. 9.3 shows the communication times between the various points. Points B-C show the sending and receiving time of data between the board and the motor. Communication is very fast. In fact, as it can be see from the graph, it takes less than a millisecond to send the command and get the feedback.

Transmissions, denoted by letters A-D, measure the time it takes for a ROS node to send a command and receive a response. Commands sent from the ROS node have to pass through the microcontroller to the motor driver and back, resulting in a longer send/receive time of around 50 ms. Finally, the letters D-A indicate how long it takes the ROS node to send a new command after receiving feedback from the previous one.

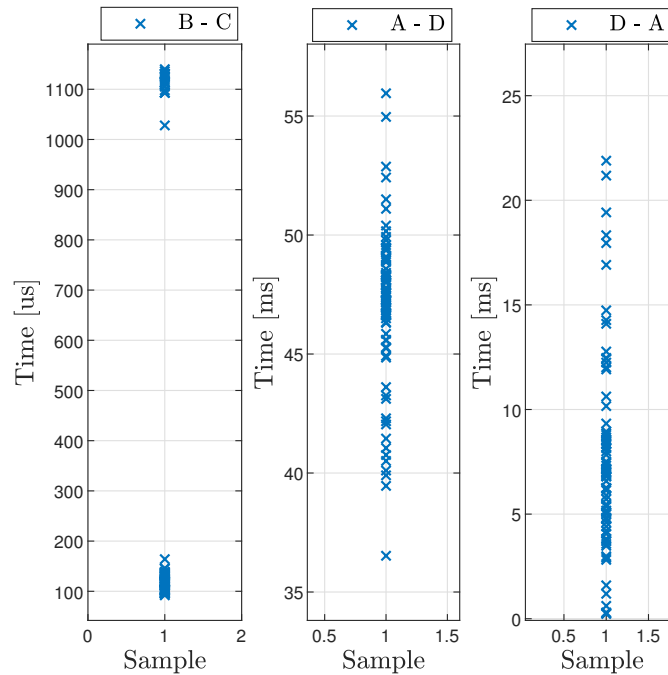


Figure 9.3: Time response - one motor at time

In the figure 9.4 shows the various histograms referring to the various transmissions, from the script the total samples. As can be seen from the graphs, each single command takes a different amount of time to be execute. This is a disadvantage of ROS 1, as it is not able to guarantee constant transmission times; this problem is described in detail in the following paper [9].

As can be see from the picture, the transfer between microcontroller and motor driver is more constant. In fact, the STM32H7 board guarantees high performance and enables accurate and stable communication thanks to its high-precision clock. On the contrary, as can be seen from the figure 9.4 the time distribution is larger when ROS nodes are present. This temporal distribution can be reduced by using ROS 2 middleware.

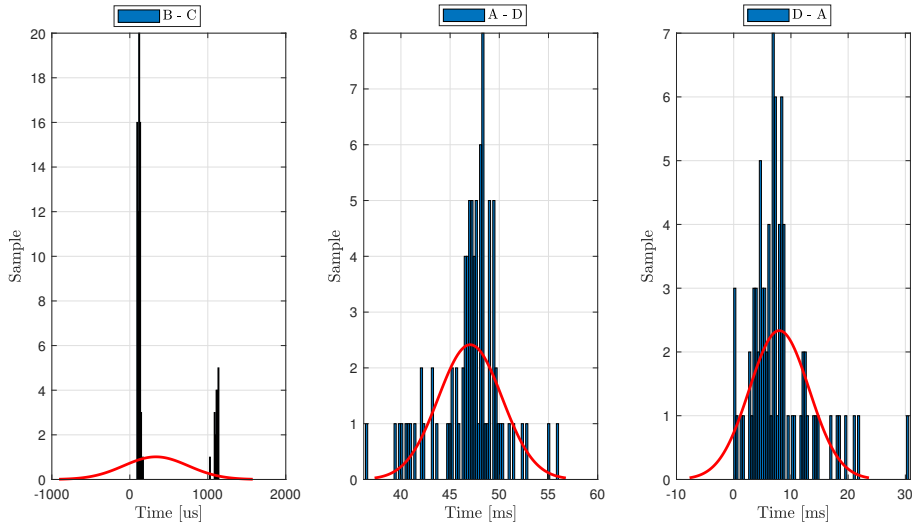


Figure 9.4: Histogram time response - one motor at time

The graphs shown above were printed out using MATLAB, reading the file created by the python node containing all the transmission times.

## 9.2 Three motor at time - (matrix mode)

As mentioned above, in order to have a comparison between the two different communication methods, a second script was developed to measure transmission times in the case where commands to the motors are sent all at once from the PC to the microcontroller, as well as feedback from the motors being received in a single message. This so-called matrix mode has been implemented in order to reduce the number of messages and thus increase communication speed.

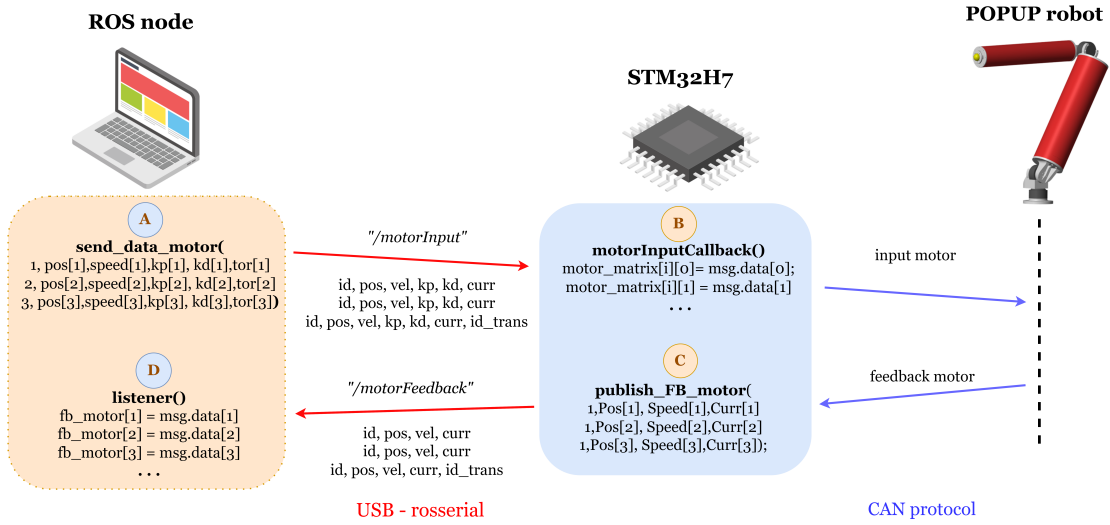


Figure 9.5: Scheme measured transmission time - matrix mode

Similar to the above, the firmware code has also been modified to allow time measurements.

The function called by the microcontroller subscriber (`motorMatrixInputCallback()`) calls the `getCurrentMicros()` function which stores the time at which the motor command was received by the MCU. After receiving the vector containing the commands for all the three motors, the board unpacks the data and deliver a single command to each motor.

As it can see from this figure 9.6, the time between receiving the command and receiving the response is longer due to the time it takes to unpack the received message.

As in the previous mode, the code called up by the subscriber was modified in order to allow the measurement of times. Furthermore, as can be seen from the code, there are for loops, which have the task of unpacking the vector received from the ROS node, and then sending one command at a time to the motors (this operation takes time).

```
void motorMatrixInputCallback() {
    tick_motor_input = getCurrentMicros();
    //for cycle used to unpack the three motor command
    for(int i = 0; i<3; i++){
        for(int j = 0; j<7; j++){
            motor_matrix[i][j] = motor_matrix_input_value.data[i+dstride1*j];
        }
    }
    . . .
}
```

The code below shows the function called up to send the motor feedback from the microcontroller to the PC. In contrast to the previous case, the motor feedback cannot be published as soon as it is received. All three motor feedbacks must be collected within the vector and then published. Depending on the received feedback id, this function saves the data in a different location of the vector. The `getCurrentMicros()` function is also called within this function, in order to save the tick at which the publishing occurs.

```
void publish_FB_motor_matrix(float id, float pos, float speed,
                             float torque){

    //add the variable containing the time to the motor feedback vector
    motor_matrix_FB.data[12] = id_transaction;
    motor_matrix_FB.data[j+5] = tick_motor_input;
    motor_matrix_FB.data[j+6] = getCurrentMicros();
}
```

### 9.2.1 Time analysis - script python

Starting with the script developed for the previous mode, minor changes were made to publish all engine commands in one vector and get all feedback in one message. The timing function remains the same.

The following line of code is present within the python node and is responsible for saving all the transmission times of the various points into a file. The first column of the file will report the time taken to complete path B-C.

Inside the variables `fb_motor[14]` and `fb_motor[13]` are the time instants measured by the microcontroller via the `getCurrentMicros()` function.

Specifically, the variable `fb_motor[13]` contains the instant of time at which the command was received, while the variable `fb_motor[14]` contains the instant at which the motor feedback was published. To get the result of how long it takes to communicate between the board and the engine, a subtraction of the two values is performed and saved to a file.

finally, the variables `time_tx` and `time_rx` contain the time instant at which the command was transmitted and the feedback received; each time a new command is sent, all these variables are reset in order to make a new measurement.

```
while(fb_motor[12]!=id_trans):
    . . .
    file_write((fb_motor[14]-fb_motor[13]), (time_rx/1000)-(time_tx/1000),
              ((time_tx-prec)/1000))
    . . .
```

In the previous mode, the maximum time it took to complete the B-C pass was about 1 ms. This mode takes about 55 ms, but during this period commands and feedback are sent and received from all three motors, whereas in the previous case there was only one command/feedback. sent and received.

A big difference from the previous mode is the time taken in the A-D path, (sending and receiving the command and feedback from the ROS node). In the previous case, in fact, the time taken to send a single command and receive the respective feedback was about 55 ms, in this case, the time taken to send three commands and receive the three feedbacks is not three times the previous value, but is about 100 ms, so the transmission with single message for all the three motor is faster. Finally, the D-A path (the time taken by the ROS node between the end of one transmission and the beginning of the next) is similar in timing to the previous mode, since the python node run on PC.

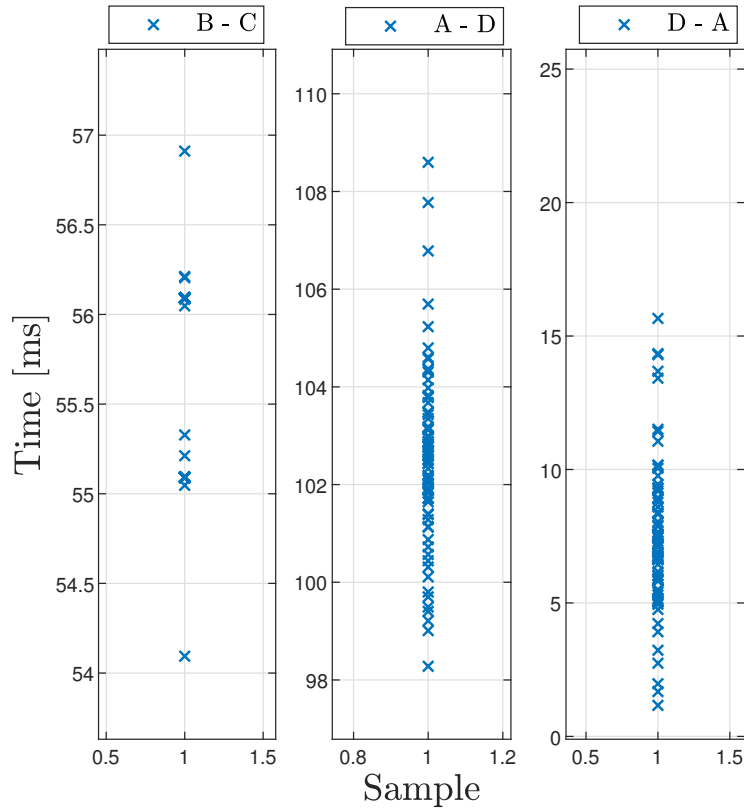


Figure 9.6: Time response - three motor at time

Even in this mode, the times linked to ROS 1 are not precise, on the contrary,



the operations performed by the STM32H7 are precise. As can be seen from the graph, the B-C path, i.e. the CAN communication between the board and the motors is uniform.

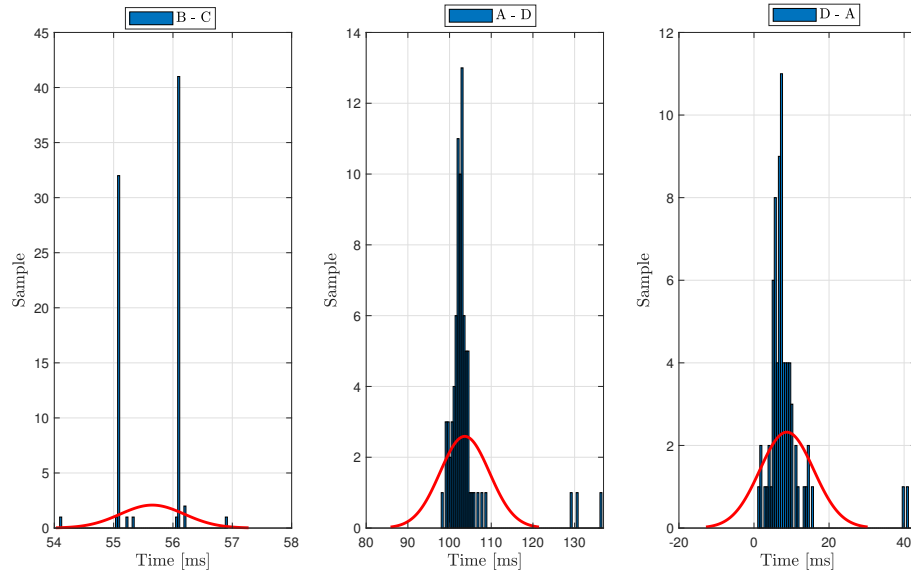


Figure 9.7: Histogram time response - three motor at time



# Chapter 10

## Popup speed control

In the previous chapter, it was discussed how different devices communicate, how they are configured, and how fast they transfer data.

At this point in the project, it was decided to operate the POPUP robot's motors only with speed, rather than sending the position. Velocity control provides smoother motion, is more suitable for low control frequencies, more robust to control signal variation, and it reduces collision forces more effectively.

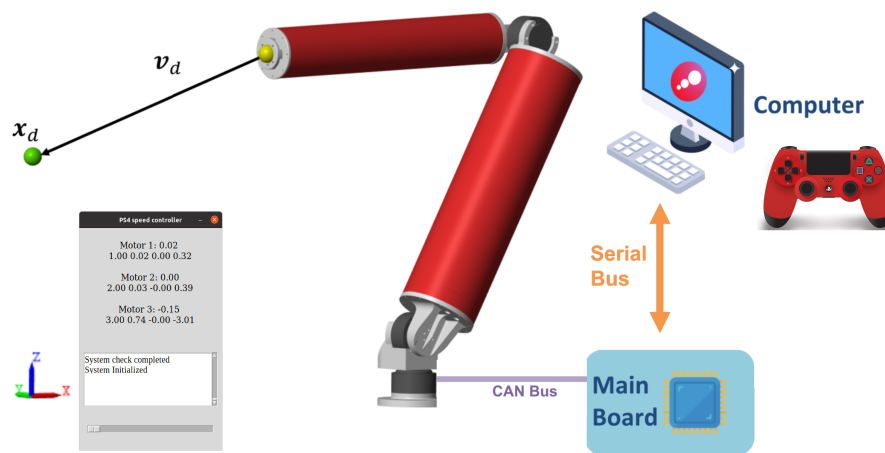


Figure 10.1: POPUP robot - speed controller

Up to this point the motor is controlled only by position. In fact, as mentioned earlier, it needs five parameters to control the motor, as stated in the datasheet. For position control, only the final position to be reached and the associated  $k_p$  should be sent to the motor his driver, all other parameters (speed,  $k_d$ , torque) should be set to zero.

For position-only motor control, there is no need for continuous communication between motor and board, as the motor remains stationary once it reaches the desired position even resending the same position, it still remains stationary.

This concept does not apply to engine speed control. Constant communication between the motor and the microcontroller is required to regulate the speed of the motor. In this case, the microcontroller mediates communication between PC and the motors, so the ROS node constantly publish new commands to enable speed control.

The python script described in the previous chapter was used to control the position of the engine. This script publish commands to three engines through a for loop and waits for feedback before sending the next command to avoid data loss or overload communication line.

Using the same script, the only change that was made was the change of the transmitted data and the removal of the graphic interface. Similar to position control, in pure velocity control it must be specified the motor to move by specifying the ID (1 to 3 in this case), the speed at which the motor is to move, and the corresponding kd (for actuators AK80-80 it is a value between 0 and 5), setting all other parameters to zero (position, kp, torque).

In the same way as the position control via the `send_data_motor()` function, the speed values of the different motors set by the user will be published on the topic `"/motorInput"`, and then sent to the motors using the `CAN_TX_AK8080()` function on the board, which will correctly format the data to be transmitted via CAN to the motors. Having developed the script previously, so solving the synchronisation problems between the various messages, the transition from position motor control to speed motor control was simple. Only the values sent to the motors had to be changed.

```
def motor_input():
    while True:
        for i in range(1, 4, 1):
            send_data_motor(i, pos_motor[i], speed_motor[i], kp_motor[i],
                           kd_motor[i], torque_motor[i])
        print("-----")
```

However, it is removed the graphical interface that allows setting the final position of the motor with a slider and a Move! button that sends the position to the motor. With pure position control, sending the same position multiple times didn't matter because the motor would hold it once it was reached. Conversely, speed control requires constant communication with the motor. This is due to the

fact it is necessary to constantly send the motor to the desired speed. Finally, to stop the motor, it must be send zero speed.

Controlling motors in speed via the GUI is therefore very difficult, since the for loop publishes the same speed at a much faster rate than the user's reaction time, since the user via the GUI would have to set the motor speed, and then reset it to zero to stop the motor at the desired point.

## 10.1 Gamepad control

To have quick control over speed, it was decided to use a controller; by moving the controller's two analogue sticks, the user may adjust the speed of the motors. The script used for position control has therefore been slightly modified to allow the integration of the controller, then the GUI is no longer used to send commands, but is used to show the speed sent to the motors (via the controller) and their feedback. The integration of the controller with the python script was possible thanks to the `ps4controller` library. When the joystick is connected to the PC via USB and these libraries are used, tasks can be programmed to run based on which button is pressed.

The code below shows an example of the use of this library, connecting the controller to the PC via USB and pressing the X button on the controller prints *"hello word"* on the IDE terminal.

```
from pyPS4Controller.controller import Controller

class MyController(Controller):

    def on_x_press(self):
        print("Hello world")
```

The following example code has been modified to allow the user to set the speed of the motors using the controller's analogue stick, thus removing the sliders on the GUI. The library allows real-time reading the button controller. By using this feature, the rotation speed of the motor can be easily controlled, and the above problem can be solved.

The analog sticks is set up so that when moved from the home position, the `speed_motor` variable assumes a non-zero value and the motor moves. Conversely, when the user releases the analog stick, it will automatically return to its initial position and the corresponding motor's speed will be zero. This direct reading of the controller input allows the speed control of the motors in an easier way. By moving the analogue sticks forwards or backwards, the user saves a positive or

negative speed value in the variable `speed_motor`, which will be published in the topic `"/motorInput"`, thus allowing the motor to be moved forwards or backwards.

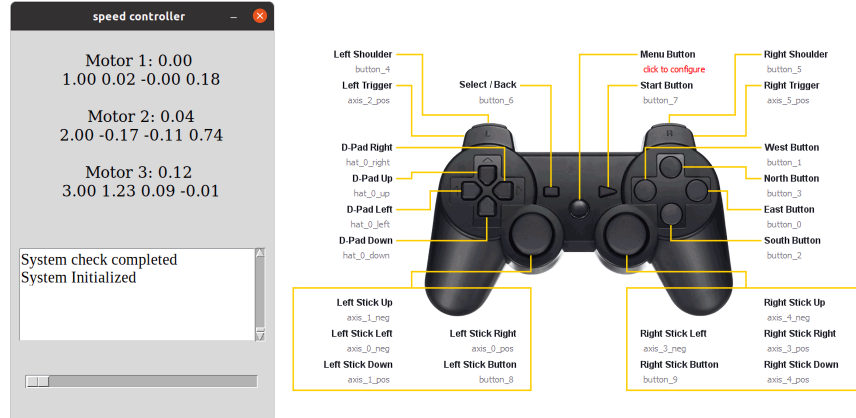


Figure 10.2: Gamepad speed control command

Previously, to regulate the motor, only the position was sent, so separate slider is created for each motor, allowing the user to set different positions for each motor. It is necessary to connect analogue stick motions with each motor in order to control the velocity of all three motors. On the controller there are two analogue stick that can move in all four directions. It was thus established that the forward and backward movement of the analog stick determines the positive or negative velocity of a single motor. In this project, it was used the right analog stick to control motor 2 only. The left analog stick controls motors 1 and 3. Moving the analog stick back and forth controls motor 3, and moving the same analog button left and right controls the speed of motor 1.

## 10.2 GUI - speed control

Having moved the control of the motors to a device and no longer via a graphical interface, the latter has been modified by removing the sliders used to set the position of the motors and replacing them with a representation of the speed sent to the motors and their feedback.

The figure 10.2 shows the graphical interface that appears once the python script for speed motor control is started. The first three lines show the speed values read by the controller. The values shown are the same as those published on the topic `"/motorinput"` and consequently sent to the motor over CAN (the speed sent to the motor is rad/sec). Depending on the movement of an analogue stick

relative to each other, the reported value will change on the GUI. As mentioned above, to control the motor's speed, the  $k_d$  must also be sent to the motor driver. During testing, it was found that by setting a  $k_d$  value of five, which guarantees smooth, controlled movement of the motor, the script automatically sets the  $k_d$  equal to five for all motors.

The next three lines show the feedback from the three motors at regular intervals. The numbers shown represent in order: motor id (which motor the subsequent data refers to), current motor position, current speed and current consumption. Finally, the window at the bottom will display all the messages that will be posted on the topic `"/statusRobot"`, such as info on the progress of the programme or any errors reported during communication. In this way, all commands and data required to control the robot are displayed on a single graphical interface. Collecting feedback from engines in real time, allows to print real-time graphs using the python library `matplotlib`.

Finally, via the following graphical interface, it is possible to adjust the maximum speed that can be sent from the controller to the motors. In fact, the analogue stick is not used as a switch whose state can be ON/OFF, but the controller's analogue stick is read by the library as a potentiometer; therefore by moving the analogue stick forward, the value read by the library increases in value, which can be used to send a higher speed to the motors. The sliders (at the bottom of the GUI window) can thus be used to set the maximum value that can be read by moving the analogue; by setting a high limit, the value sent as speed to the motors will be higher, allowing the robot to make faster movements. Setting a lower analogue value will reduce the maximum motor speed.





# Chapter 11

## PID firmware

The previous chapter described how the switch from position motor control to speed motor control was made, as well as the various changes made to the python script to allow this type of control. As previously stated, the firmware on the MCU for switching from position control to speed control was not altered in any way because the function for sending commands via CAN to the motors was already configured for speed control; the only change was to pass the correct parameters to the function. To gain more control over the POPUP robot's motors, it was decided to implement a PID controller that would run autonomously only on the microcontroller, allowing commands sent by the user via the controller to be executed more smoothly and precisely.

The following chapter will demonstrate how a PID controller works and how it has been integrated into the firmware for motor torque control.

### 11.1 PID funciton

**PID** Control stands for **P**roportional-**I**ntegral-**D**erivative feedback control and corresponds to one of the most commonly used controllers used in industry. It's success is based on its capacity to efficiently and robustly control a variety of processes and dynamic systems, while having an extremely simple structure and intuitive tuning procedures.

Traditionally, control design in robot manipulators can be understood as the simple fact of tuning of a PD or PID compensator at the level of each motor driving the manipulator joints. Fundamentally, a PD controller is a position and velocity feedback that has good closed-loop properties when applied to a double integrator system. Actually, the strong point of PID control lies in its simplicity and clear

physical meaning. Simple control is preferable to complex control, at least in industry, if the performance enhancement obtained by using complex control is not significant enough.

The physical meanings of PID control are as follows:

- **P**-control means the present effort making a present state into desired state.
- **I**-control means the accumulated effort using the experience information of previous states.
- **D**-control means the predictive effort reflecting the information about trends in future states.

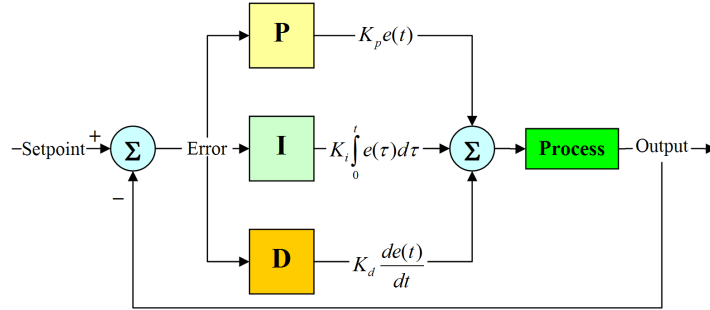


Figure 11.1: PID scheme

After the theoretical operation of the PID had been explained, it was simulated using Simulink, a MATLAB tool. It was then written in C and included into the firmware.

## 11.2 PID in C

Starting from the following equation, C code was created to reproduce the same result

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$

where

- $K_p$  is the proportional gain, a tuning parameter,
- $K_i$  is the integral gain, a tuning parameter,
- $K_d$  is the derivative gain, a tuning parameter,

- $e(t) = SP - PV(t)$  is the error (SP is the setpoint, and  $PV(t)$  is the process variable),
- $t$  is the time or instantaneous time (the present),
- $\tau$  is the variable of integration (takes on values from time 0 to the present  $t$ ).

The following definition was then used to generate the corresponding C code for computing motor torque. By comparison with the prior formulation for motor control, the variables were substituted with the values:

- $u(t)$  represents the output of the PID function, which in this case corresponds to the torque to be delivered to the motor drivers.
- $e(t)$  is the difference in engine speed between desired and actual. The torque required to feed the motors to maintain a steady position is determined by the difference between these two speeds multiplied by the tuning gains.
- $t$  inside the firmware, the PID operates in the digital domain, and this variable is determined by the period of the timer (TIM2) that recall this function.

The following is the firmware function that implements the PID controller.

```
float PIDcal(int id_motor, float setpoint, float fb_motor) {
    //Calculate P, I, D
    error[id_motor] = setpoint - fb_motor;
    derivative = (error[id_motor] - pre_error[id_motor])/dt;
    integral[id_motor] = integral[id_motor] + error[id_motor]*dt;

    //Output of the PID controller
    output = Kp[id_motor]*error[id_motor] + Ki[id_motor]*integral[id_motor]
            + Kd[id_motor]*derivative;

    //Update error
    pre_error[id_motor] = error[id_motor];
    return output;
}
```

1. The error  $e(t)$  was defined as the difference between the setpoint, i.e. the speed set by the user via the controller, and the current motor speed indicated with `fb_motor`.

2. The derivative component  $\frac{de(t)}{dt}$  is calculated as follows:

```
derivative = (error[id_motor] - pre_error[id_motor])/dt;
```

where the variable `pre_error` reports the value of the error at the previous instant, and the constant `dt` was set to the value 0.01.

3. Finally, the integrative part  $\int_0^t e(\tau)d\tau$  was calculated as follows:

```
integral[id_motor] = integral[id_motor] + error[id_motor]*dt;
```

In order to emulate the computation of the integral, each time the function is called, the integral variable is re-calculated taking into account all the previous values.

Finally, all the previously calculated parts are multiplied by the tuning gains and then added together to obtain the output of the PID.

The variables passed as input to this function are, as can be seen from the function, the setpoint, (i.e. the final speed to be achieved by the motor), the current motor speed, and lastly the motor id.

Using the same principle as for continuous control of the three motors, a timer (TIM2) is used to call up the `PIDcal()` function on a regular basis and compute the torque required for each motor. The output of the `PIDcal()` function is utilised as input for the `CAN_TX_AK8080()` function, so the torque computed from the collected data is sent to the motor to be compensated.

The code below shows the TIM2 callback, which starts with id 1 and grows by 1 each time the timer is activated until it reaches value 3, at which point it resets to value 1. As a result, after three timer cycles, all torques for all motors have been computed. Setting the timer frequency as high as possible provides torque control that ensures the stability of the robot. Since the output of the function is a torque, all other parameters to be sent to the driver are set to zero so that only torque control is available.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim == &htim2) {
        m++; //Change the motor to be controlled
        //calculate the torque to be sent to the motor
        curr[m] = PIDcal(m, speed_setpoint[m], fb_motor_speed[m]);
        CAN_TX_AK8080(m, 0, 0, 0, 0, curr[m]);
    }
}
```

The functioning of the firmware has slightly changed as a result of the addition of this control; Fig. 11.2 displays a flow chart of the operations performed. The

speed input entered by the user via the ROS node is not sent directly to the motors, but is provided as input to the PID function. In this way, the `PIDcal()` function calculates the torque based on the received values and sends them to the motors. This is done cyclically for all three motors, using the TIM2 timer. The TIM8 previously used to send data directly to the motors, is now used to periodically send feedback from the motors to the ROS node.

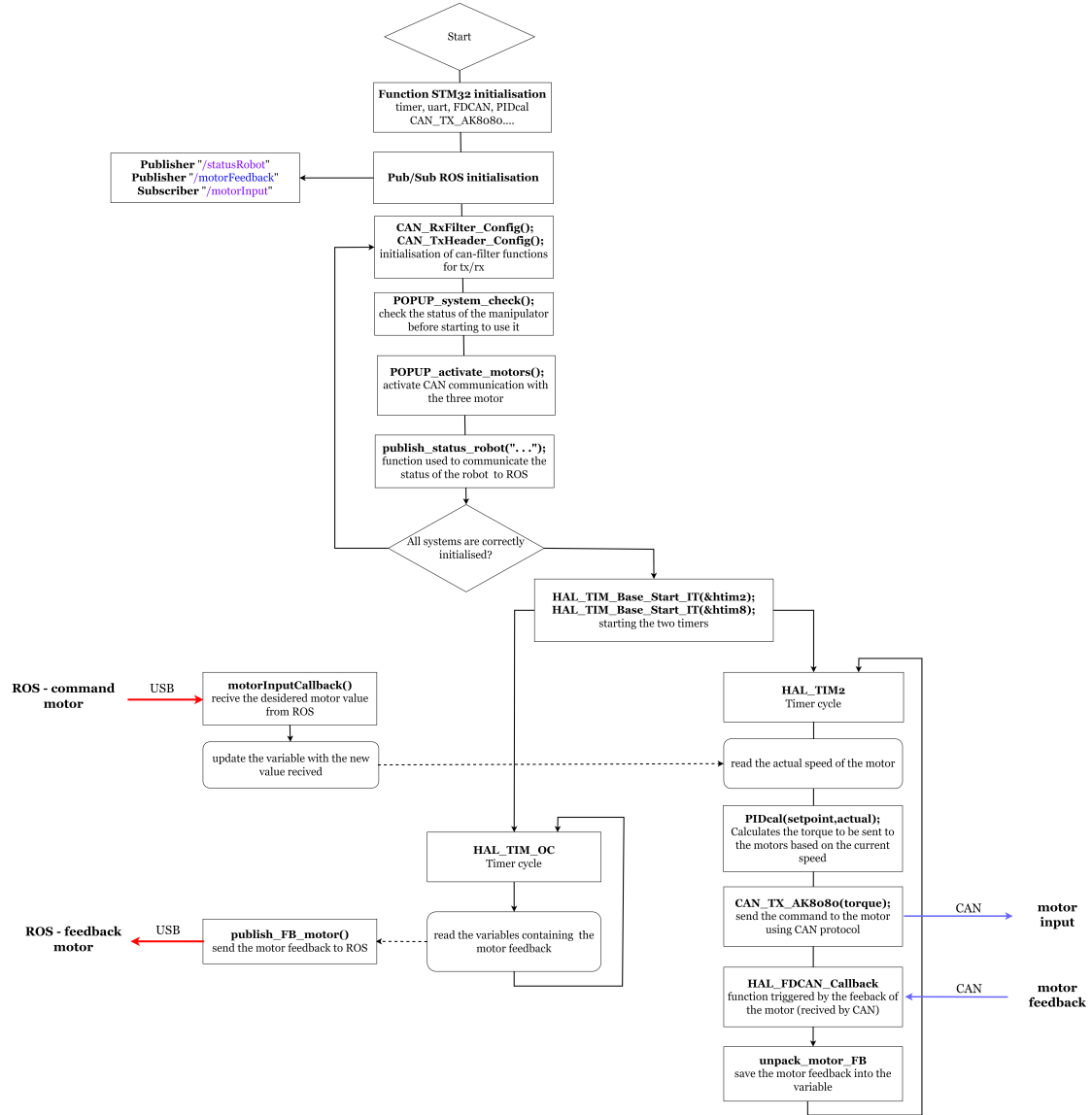


Figure 11.2: Complete firmware's flow chart

## 11.3 PID tuner

So far, it has been shown how the code for the PID implementation was built; in the following section of the chapter, it will be described how the PID's  $k_p$ ,  $k_i$ , and  $k_d$  parameters were chosen for the various engines.

The first motor on which the PID was set was motor 2, because it has to sustain the two links and motor 3 during movement, it requires more torque because it supports the majority of the robot's structure.

In general, the PID parameters ( $k_p$ ,  $k_d$ ,  $k_i$ ) are set one at a time, beginning with zero and progressing to see how the system responds. The table 11.1 shows the relevance of the variation of each parameter.

	Rise time	Overshoot	Settling time	Stability
$K_p$	Decrease	Increase	Small change	Degrade
$K_i$	Decrease	Increase	Increase	Degrade
$K_d$	Minor change	Decrease	Decrease	Improve

Table 11.1: PID parameters

The code that emulates the behaviour of the PID is written within the firmware loaded on the microcontroller, so for the correct setting of the parameters it is necessary to reload the firmware on the board each time and check the motor's behaviour with the new parameter. The time it takes to modify and reload the firmware must be added to the time it takes to validate the robot's behaviour.

To make this PID setting operation faster and easier, a python node was created that allows the tuning gains of each motor to be updated without reloading the firmware on the board.

A new window was built to allow setup of the PID parameters based on the script used for motor speed control. The graphical interface is shown in Fig. 11.3. Use the drop-down menu to pick the motor for which the tuning gains are to be updated, then use the sliders to modify the values of the gains ( $k_p$ ,  $k_d$ ,  $k_i$ ), and lastly press the Tune! button to publish the values set by the sliders on the topic and insert them into the PID controller. The three lines at the bottom of the GUI display the current PID parameters of the three engines.

A subscriber was implemented to accept the values released by the ROS node in order to allow changes to firmware parameters. When you press the Tune! button, the slider parameters are transmitted as a message to the topic `"/PIDValue"` to

which the microcontroller is subscribed, and the previous PID parameter values are overwritten with the new ones. When the TIM2 invokes the `PIDCal()` function, the output will be computed using the newly obtained tuning gains. This script allows the PID settings to be modified more quickly and easily, removing the need to reload the board’s firmware, and the parameter change is done in real time, allowing the motors’ reaction to the new parameters to be seen immediately.

The development of this additional feature demonstrates the system’s adaptability. It was possible to update the PID parameters dynamically by adding a publisher/subscriber. In comparison, adjusting the parameters without the ROS node would have taken much longer because the firmware on the board would have had to be reloaded each time the parameters changed.

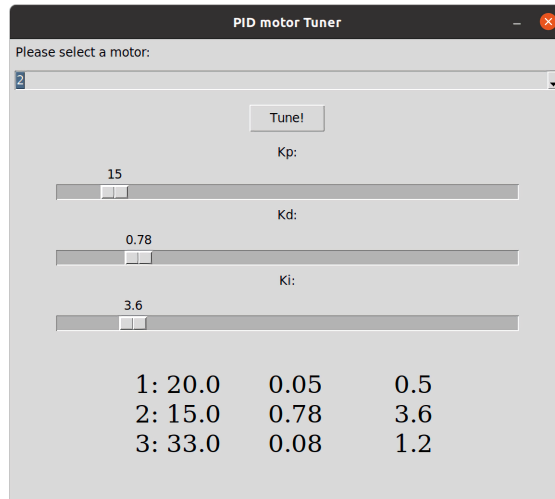


Figure 11.3: GUI - PID tuner

Using this GUI, it was feasible to rapidly and easily tune the PID for all three engines. The table shows the parameters discovered that allow the POPUP robot to operate optimally.

	kp	kd	ki
<b>motor 1</b>	20	0.05	0.1
<b>motor 2</b>	20	0.05	0.5
<b>motor 3</b>	20	0.05	0.1

Finally, after determining the best PID parameters and developing the ROS node to control the manipulator. The POPUP robot was tested by making it

follow various trajectories to obtain various configurations. The graphs in the figure show the various motor feedbacks during a series of movement.

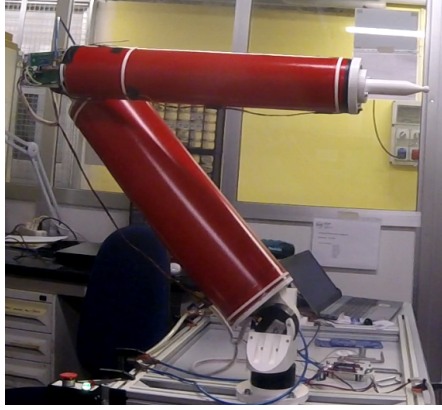


Figure 11.4: Movement POPUP robot

The position feedback of the three motors is shown in radians in the Fig. 11.5. As the manipulator movements are confined inside a specific space, the position of the motors changes inside a range.

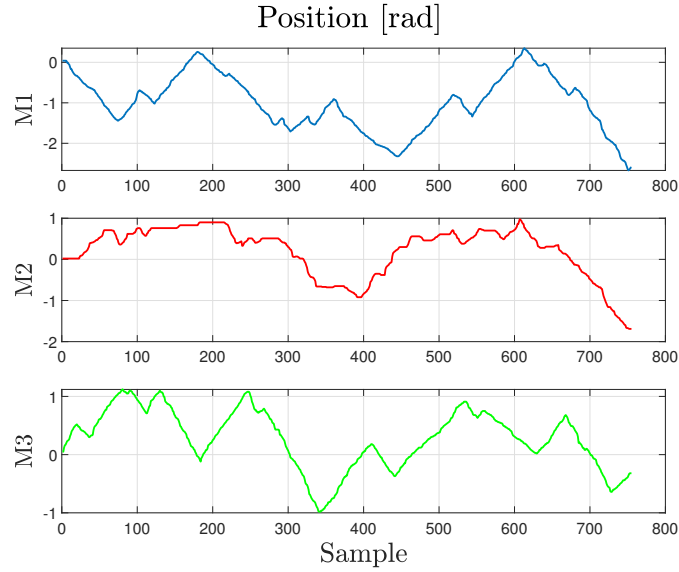


Figure 11.5: Motors feedback - position

The speed of the three motors is shown in the Fig. 11.6, which is relatively low in order to have better control over the robot's movement; the maximum speed that the motors can achieve can be set using the GUI.



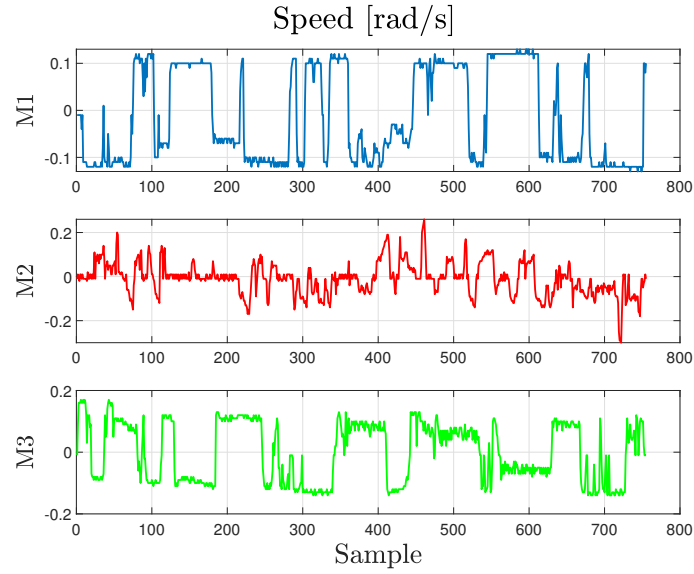


Figure 11.6: Motors feedback - speed

Finally, the current consumed by the motors shown in the FIG. 11.7 is low because the robot is very light and hence great torque is not required.

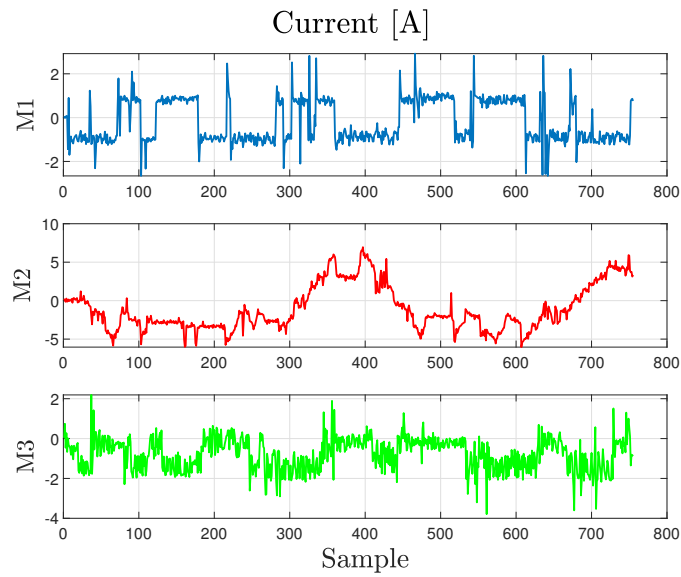


Figure 11.7: Motors feedback - current



## Chapter 12

# Conclusion

The purpose of this thesis is to use the robotic software ROS 1.0 to control a three-degree-of-freedom robot. The POPUP robot, which will be operated, is a soft manipulator created for space applications in the laboratories of Politecnico di Torino. It is composed of two inflatable links and three rigid joints.

A similar project was done in the following publication [12], in which ROS middleware was also utilised to drive a 7 dof robotic arm. The main difference between this project and the one described in the paper is that no ST microcontroller was used for communication with the motors, and the connection between PC and robot was created over ethernet to reduce communication latency (the roserial protocol was always used, as it also supports ethernet communication).

The usage of ROS 1.0 middleware, in this dissertation, enabled the POPUP robot to be controlled in a variety of ways, beginning with position control and progressing to speed control. ROS 1.0 offers several services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message passing between processes, and package management. Using this library tool, it was feasible to create complete robot control, allowing for the integration of other devices and the creation of a modular framework. The application and the entire system have been configured and designed to be versatile and modular, allowing for easy integration both in terms of hardware, with the addition of sensors and motors on the manipulator, and software with the use of tools for simulating and monitoring the POPUP robot.

The POPUP robot ecosystem is made up of three major components: a PC, a microcontroller, and a robot. The microcontroller is in charge of "mediating" communication between the PC and the motors of the robot. It uses the CAN protocol to communicate with the three motors, while it is connected to the PC through USB and communicates with the ROS node via the roserial protocol (communication protocol present in ROS 1)

The first step in this project was to connect the motors and microcontroller through CAN. To enable this communication, a firmware that provides complete control of the motor was developed, allowing the microcontroller to supply input to the motors and receive responses.

Subsequently, the ROS middleware was integrated into the project, allowing the direct connection with the microcontroller and consequently with the three motors. It was very important to set up the microcontroller in such a way as to allow simultaneous communication both with the PC via USB and with the motors via CAN. Then the manipulator control system was developed using nodes and publisher-subscriber communication, which is the main communication paradigm of ROS. The use of the python programming language for node development enabled the implementation of algorithms to control and visualise the manipulator's data.

The combination of ROS middleware and the python programming language for developing nodes provides the project with great versatility and the possibility to be upgraded. Using the publisher and subscriber communication it is possible to control and modify every single data of the POPUP robot. Furthermore, with this configurations, the motors may be controlled in various modes (position, speed, torque) by simply running the appropriate script, without the need to replace the firmware or other components, which underlines the modularity of the project. ROS offers a wide range of robotics software development tools, that allows the development of advanced applications with just a few lines of code.

A future development of this project could be the reception of data from sensors mounted on the robot, which can be done simply by adding a publisher that publishes the data received from the sensor on a defined topic and then live-printed using a python library. This aspect allows great modularity to the project, as the code that implements the functions is not device-bound, but can be ported to any device, maintaining correct operation.

Finally, another step forward for this project would be to make better use of the robotics tools supplied by ROS, such as simulation and data representation software, in order to create an accurate software for operating and monitoring the POPUP robot

# Bibliography

- [1] Emmanouil Tsardoulas and Pericles Mitkas. *Robotic frameworks, architectures and middleware comparison*. 2017. DOI: [10.48550/ARXIV.1711.06842](https://arxiv.org/abs/1711.06842). URL: <https://arxiv.org/abs/1711.06842>.
- [2] L. Joseph and J. Cacace. *Mastering ROS for Robotics Programming - Third Edition: Best Practices and Troubleshooting Solutions when Working with ROS*. Packt Publishing, 2021. ISBN: 9781801071024. URL: <https://books.google.it/books?id=08GQzgEACAAJ>.
- [3] Marwane Ayaida et al. “TalkRoBots: A Middleware for Robotic Systems in Industry 4.0”. In: *Future Internet* 14.4 (Mar. 2022), p. 109. DOI: [10.3390/fi14040109](https://doi.org/10.3390/fi14040109). URL: <https://doi.org/10.3390/fi14040109>.
- [4] Pablo Iñigo-Blasco et al. “Robotics software frameworks for multi-agent robotic systems development”. In: *Robotics and Autonomous Systems* 60.6 (June 2012), pp. 803–821. DOI: [10.1016/j.robot.2012.02.004](https://doi.org/10.1016/j.robot.2012.02.004). URL: <https://doi.org/10.1016/j.robot.2012.02.004>.
- [5] Ayssam Elkady and Tarek Sobh. “Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography”. In: *Journal of Robotics* 2012 (2012), pp. 1–15. DOI: [10.1155/2012/959013](https://doi.org/10.1155/2012/959013). URL: <https://doi.org/10.1155/2012/959013>.
- [6] Leon Jung Yoonseok Pyo Hanchol Cho and Darby Lim. *ROS Robot Programming (English)*. ROBOTIS, Dec. 2017. ISBN: 9791196230715. URL: <http://community.robotsource.org/t/download-the-ros-robot-programming-book-for-free/517D>.
- [7] Henrik Andreasson et al. *Software Architecture for Mobile Robots*. 2022. DOI: [10.48550/ARXIV.2206.03233](https://arxiv.org/abs/2206.03233). URL: <https://arxiv.org/abs/2206.03233>.
- [8] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. “Exploring the performance of ROS2”. In: *Proceedings of the 13th International Conference on Embedded Software*. ACM, Oct. 2016. DOI: [10.1145/2968478.2968502](https://doi.org/10.1145/2968478.2968502). URL: <https://doi.org/10.1145/2968478.2968502>.

- [9] Jaeho Park, Raimarius Delgado, and Byoung Wook Choi. “Real-Time Characteristics of ROS 2.0 in Multiagent Robot Systems: An Empirical Study”. In: *IEEE Access* 8 (2020), pp. 154637–154651. DOI: [10.1109/access.2020.3018122](https://doi.org/10.1109/access.2020.3018122). URL: <https://doi.org/10.1109%2Faccess.2020.3018122>.
- [10] J. A. Cook and J. S. Freudenberg. “Controller Area Network (CAN)”. In: *EECS 461, Fall 2008* ().
- [11] C. Novello. *Mastering STM32: A Step-by-step Guide to the Most Complete ARM Cortex-M Platform, Using a Free and Powerful Development Environment Based on Eclipse and GCC*. Leanpub, 2016. URL: <https://books.google.it/books?id=ZZnfzQEACAAJ>.
- [12] Guojun Zhang et al. “A Real-time Robot Control Framework Using ROS Control for 7-DoF Light-weight Robot”. In: *2019 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*. IEEE, July 2019. DOI: [10.1109/aim.2019.8868488](https://doi.org/10.1109/aim.2019.8868488). URL: <https://doi.org/10.1109%2Faim.2019.8868488>.