Politecnico di Torino

Corso di Laurea in Mechatronic Engineering

Master's Degree Thesis

ROS-Based Data Structure for Service Robotics Applications

Supervisor

Marcello CHIABERGE

Candidate

Simone RAPISARDA

April 2019

Ringraziamenti

Arrivato alla fine di questo percoso accademico non posso fare a meno di ringraziare alcune persone senza le quali quest'impresa sarebbe stata molto più ardua se non, addirittura, quasi impossibile.
Innanzitutto ringrazio il professor Marcello Chiaberge per avermi dato la possibilità di partecipare a questo progetto accogliendomi all'interno del team del PIC4SeR dove ho potuto lavorare a questa tesi in un ambiente caratterizzato da persone competenti ed entusiaste del proprio lavoro, capaci di saper trasmettere una passione unica ed autentica nei confronti delle tematiche affrontate quotidianamente. A tal proposito, voglio esprimere la mia più sincera gratitudine nei confronti degli altri ragazzi del LIM senza i quali questi ultimi mesi non sarebbero stati i migliori mesi da me vissuti al Politecnico di Torino.
Con la vostra simpatia, gentilezza e disponibilità nel momento del bisogno, mi avete fatto capire cosa significa lavorare con persone competenti e al tempo stesso definibili veri e propri amici. Per questo e per molti altri motivi, grazie Jurgen, Angelo, Lorenzo, Luca e Gianluca.
Infine il più importante dei "grazie" è per la mia famiglia: i miei genitori e mia sorella. In un percorso arduo come quello da me intrapreso difficilemente si può arrivare alla fine senza aver passato dei momenti difficili o semplicemente aver dovuto affrontare dei fallimenti. La mia più grande fortuna, tuttavia, non è stata non aver mai fallito un esame, aver mancato un obiettivo o essermi sentito "perso" durante dei momenti di crisi, la mia più grande fortuna è stata sapere di poter sempre contare su di voi che, con un frase, una parola o semplicemente un sorriso riuscivate a farmi sentire di nuovo felice e "ritrovare" la strada giusta. Grazie.

S.R.

"Not all those who wander are lost
J.R.R Tolkien

Abstract

The cooperation and communication between different robotic agents is a very powerful tool useful for a lot of implementations, from the mapping of areas (SLAM) to the exchange of data to achieve tasks in a shorter time or with better solutions that can allow to save resources.
In particular, the possibility of having an Unmanned Ground Vehicle (UGV) communicating with an Unmanned Air Vehicle (UAV) is something that is in greater demand than ever before.
For instance, in agriculture applications this cooperation is quite interesting since the mapping of vineyards can be accomplished by a rover which then shares the data with a drone that can use them to define the parameters needed for its mission planning.
The purpose of this thesis is to build a common data structure able to receive information from an agent, regardless the nature of the robot (UGV or UAV), and make these data available for other robots that need to work together to achieve a common task.
The first part is an introduction to the ROS environment and explains how to use this tool in order to program the robots and organize the data structure that will be used for the thesis.
The second part discusses about UGVs, in particular about the two TurtleBot3 rovers which have been used for this project, the Waffle and the Burger models, and the Jackal rover. In this section it is also explained how these rovers can be controlled using the Pixhawk autopilot, which is usually used with drones and therefore needs to be properly programmed in order to be utilized with UGVs.
Finally, the last part focuses on MAVROS and MAVLink protocol and explains how these two tools work and how they can be used as interface to allow the communication between the Pixhawk and the rover.

Contents

List of Tables

List of Figures

Introduction

Overview and State of the art analysis

In the last years, our life and daily routine are increasingly more conditioned by the network of information that, even if invisible, is all around us. We use these information and share them between us every day for a lot of different reasons, and some tasks would be simply impossible to achieve without this process of data sharing.
If we link this analysis to the fact that, nowadays, robots are always more present and utilized in a lot of different scenarios in our lives, the direct consequence is that this process of data sharing can be extended to robots aswell, in order to improve their efficiency and be more valuable for their user.
One example of such benefit is the project ended in 2014 by RoboEarth [13] which consisted in disposing four different robots in an experimental setup with two hospital rooms [12]. Each of these four robots have their own structure and have been built accordingly to their specific purpose: the first (Ari) is a mobile sensive platform that takes care about the mapping of the area where the other robots have to work, the second (Amigo) is an advanced humanoid with a rich set of sensors and two robot arms, the third (Pico) is a much more simple humanoid robot with just a tray attached to it and the last (Pera) is a fixed robotic manipulator mounted on a table.

Figure 1: RoboEarth robots

During the demonstration the robots receive vocal instructions by the patients and proceed to accomplish their tasks working all together and sharing data through a cloud system. The humanoid robots can move around the rooms thanks to the map elaborated by Ari, the mobile sensive platform, and they can grab or move objects with the help of the robotic manipulator that can reach and reposition items.
This example can prove the importance of multi-agent cooperation and data sharing in such a relevant domain as the medical field.

While working in a multi-agent application, one of the hardest obstacle to be overcome is the different kind of approach to be used to instruct the robots about their tasks. More precisely while programming the TurtleBot3 [15] UGVs tasks, the communication exists directly between the user and the robot whithout any mediation involved.
On the other hand, while programming a task for a drone using Pixhawk [8] autopilot, the procedure is more complex and there are more steps involved to achieve the final goal. The code written by the user is elaborated by the mission planner that generates the right inputs and coordinates needed by the Ground Control Station (GCS) which will produce the instructions that the drone will follow in order to complete the task.

The MAVLink [3] communication protocol and MAVROS [5] package are useful to translate the istructions from the human-readable representation to the machine-readable format used by the drones.

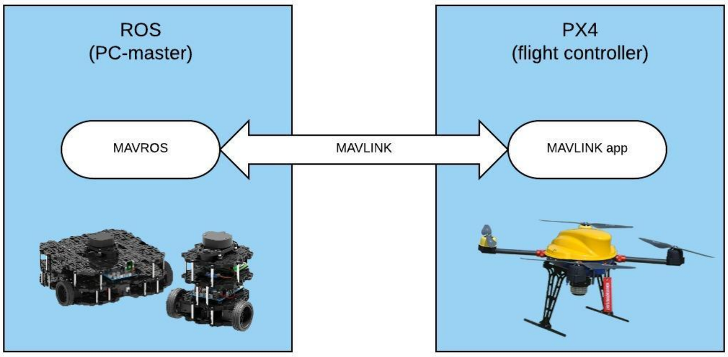


Figure 2: MAVLINK and MAVROS system architecture

Objectives

The objective of the thesis is to build an efficient data structure able to work as central node of communication for a network of different type of robots that can share and store information regardless of the protocol they use.
This goal has been reached addressing the problem with a step-based approach:

- first, the algorithms are tested in a simple navigation application without using any external autopilot as mission controller. In this phase, no interface is needed between the data structure and the agents that use it. In particular, some navigation algorithms have been tested with Turtlebot3 and Jackal rovers inside the ROS framework.

- secondly, another set of tests have been made using a Pixhawk autopilot to control and pilot a UGV giving the inputs with a Radio Controller (RC tests phase). In this case, no ROS interface is utilized, instead, a Ground Control Station has been used to initialize and manage all the parameters for the tests.

- lastly, the final procedure is performed using the same hardware set up of the previous one, with the exception of the RC. As a matter of fact in order to have the full availability of the data and the possibility to share them, the ROS framework it has to be used, and so the input commands have been sent with the MAVROS and MAVLink protocol system.
  In this case a sort of interface comes into play and allows the correct exchange of data.

Chapter 1

ROS software framework

Even if there are a lot of different kinds of robots and a large variety of tasks that robots can accomplish, some common traits are often present while working in this environment. For this reason, a group of programmers of the Stanford University in the mid-2000s created a common platform where is possible to share codes and ideas regarding the robotic field, the Robotic Operating System (ROS) [14] framework.
ROS consists in a set of libraries useful to control and program robots and provides different services [9]:

- a set of drivers that allows you to read data from sensors and send commands and instructions to motors and other actuators

- a large collection of robotics algorithms such as SLAM algorithms, autonomous navigation algorithms, sensor data interpretation algorithms, and much more

- various computational infrastructures that allows you to move data around, to connect the various components of a complex robot system, and to incorporate your own algorithms

- a large set of tools that make it easy to visualize the state of the robot and the algorithms, debug faulty behaviours, and record sensor data

The strength of this platform lies in his versatility since through ROS the user can focus on the peculiarities of his project while the basics, for the majority, have already been taken care of by the platform itself.
This is accomplished thanks to the variety of nodes and topics that are linked together and which can store and exchange data. This is really important since when trying to program a robot to achieve a task, even the simplest one, starting from zero, the amount of work and steps required would be too large for a single person or a small group of people to handle.
With ROS and its big community of users is possible to exploit the numerous repositories to lighten the load of work needed.

1.1 ROS architecture

ROS systems are composed by a large number of independent programs that are constantly communicating with each other by exchanging *messages*. The whole system can be represented with a *graph* where the programs are the *nodes* and the *messages* that link them are the *edges*.

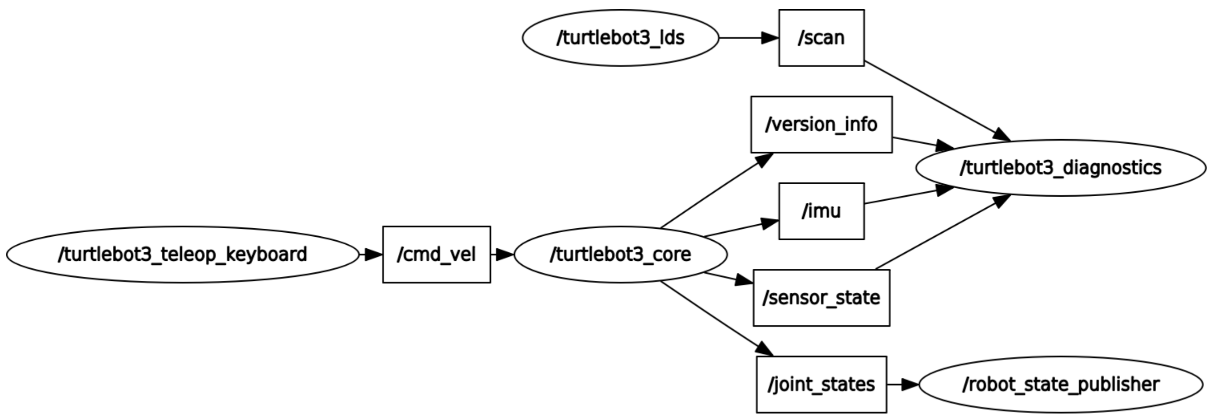


Figure 1.1: ROS graph for a teleop task

The example shown in figure 1.1 is the graph related to a simple teleop task, in which the robot is remotely-controlled with the keyboard and can be moved around. The oval-shaped blocks are the *nodes*, corresponding to the processes that perform computation. The rectangular-shaped ones, on the other hand, are the *topics*, that consist in the "channels" or "buses" through which the *messages* (represented by the *edges* in the graph) are exchanged.
The first important concept to understand while using ROS is the concept of *node*.

1.2 Nodes

A node is a process that performs computation. While dealing with robot applications, every component of the robot is controlled by a node. For example, one node controls the robot's wheel motors, one node performs localization, and so on. A useful command that allows to know all information about nodes is **rosnode info** followed by the name of the node. With this command is possible to know which subscribers and publishers are linked to that node. Each node uses *messages* to exchange information within the ROS network.

1.3 Messages

When a node needs to communicate with another node it publishes *messages* to *topics*.
A *message* is a simple data structure that has a type (integer, floating point, boolean, etc.) and can be used once the related library is imported inside the code.
With the command **rosmsg** is possible to inspect a *message* and understand which data structure it uses.

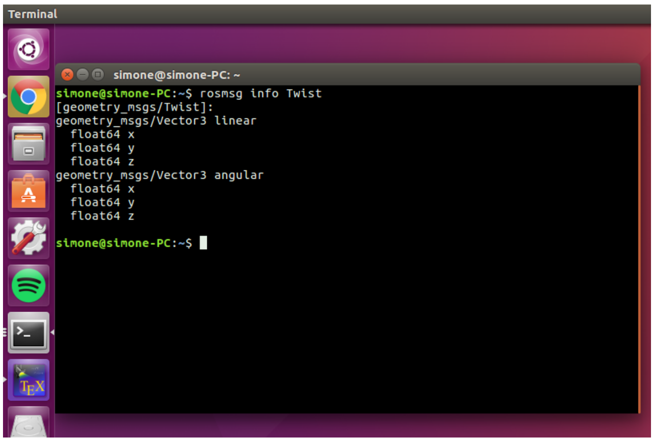


Figure 1.2: rosmsg command example

In figure 1.2 it is represented an example of this command launched on the **Twist** message. In this case this message has a data structure divided in two parts, one concerning the linear velocity of the robot and the other one concerning the angular one. Both the structures use three float numbers (x,y,z) that correspond to the velocity wanted for the robot along x,y and z axis.
All this exchange of data and information between *nodes* through *messages* is possible thanks to the usage of *topics*.

1.4 Topics

A topic is a stream of messages with a defined type that implements a *publish/subscribe* communication process. Nodes that want to receive messages from a topic can subscribe to that topic by making a request to **roscore**.
The following lines of code (extracted from the code reported in Appendix A) are an example of a typical subscriber/publisher mechanism concerning a multi-agent system, where the task for the "follower" robot is to copy the movements of the "driver" robot:

```
1 #! /usr/bin/env python
2
3 import rospy
4 from geometry_msgs.msg import Twist
5
6 rospy.init_node("follower_controller")
7 sub = rospy.Subscriber("driver/odom", Odometry, newOdom)
8 pub = rospy.Publisher("follower/cmd_vel", Twist, queue_size=1)
```

The first line is necessary to choose the right interpreter for the code that follows. In this case the information needed is the speed of the "driver" robot that is obtained by reading the **Twist** message type belonging to **geometry_msgs** messages, hence the code in lines 3 and 4. After that, a new node for the application is created and called **follower_controller** (line 6), this initialization is really important since allows communication with the MASTER node of the system. This node will then subscribe to the **driver/odom** topic and read messages of the type **Odometry** (line 7). These messages contain all the information needed to move the TurtleBot. Finally, in line 8, these information just read from the driver node are sent to the **follower/cmd_vel** topic by publishing them on that topic through a **Twist** type message.
Even if all programs can freely communicate with each other, there is one process that needs to be launched before all others and which has the task to sort all messages in the ROS network, the **roscore** program.

1.5 roscore

Every time a new node is created the **roscore** service program provides it with all the information needed in order to form peer-to-peer connection with the other nodes. Since the **roscore** has such an important role his presence is mandatory in every ROS system and that is why the **roscore** is always the first program that shall be ran. When the **roscore** program is ran, the MASTER node is created.
In a multi-agent application the MASTER can be choosen freely between all robots even if, usually, this role is covered by the user's PC. Once the MASTER node is created, all other nodes tell **roscore** which messages they provide and which they would like to subscribe to, then **roscore** shares the addresses of the relevant message producers and consumers.

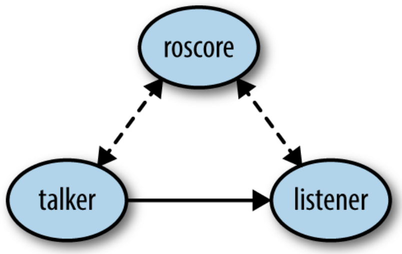


Figure 1.3: roscore connections with other nodes in the system

In the example shown in figure 1.3 the **listener** node is subscribed to the **talker** node, meaning that it reads the messages produced by the **talker**, while both periodically make calls to the **roscore** node.
Another relevant problem that comes into play both when dealing with robots that work alone and, especially, in multi-robots applications, is the management of *coordinate frames* and, for this purpose, the **tf** package has been created.

1.6 tf

In ROS the coordinate frames and the transforms between them are handled with a distributed approach. Any node can publish information about some transforms and any node can subscribe to transform data and in this way a complete picture of the robot is gathered by the various authorities.
This process is implemented by the **tf** (short for transform) topic, which uses messages of type **tf/tfMessage**. Each **tf/tfMessage** message contains a list of transforms, stating for each one the names of the frames involved, their relative position and orientation, and the time at which that transform was measured or computed.

Another important aspect regarding ROS systems , especially when the application involves multiple robotic agents, is the definition of *names* and *namespaces*.

1.7 Names and Namespaces

Names are a crucial concept in ROS, since nodes, message streams ("topics") and parameters must have unique names. Nevertheless, namespace collision are really common in robotic environments, where is frequent to find similar or identical components on the same robot, such as arms, wheels or cameras.
In order to address this issue, ROS provides the *namespaces* mechanism, with which can launch identical nodes into separate namespaces. The procedure entails the copy of a node that suffers of namespace collision into another namespace that will differ in the path definition.
For instance, when there are two similar or identical robots (e.g. the TurtleBot3 *Waffle Pi* and *Burger* models) that need to be controlled by the MASTER node, the same instruction or code can be used for both the robots but ROS needs to know which messages are sent to which node and so a namespace collision occurs.

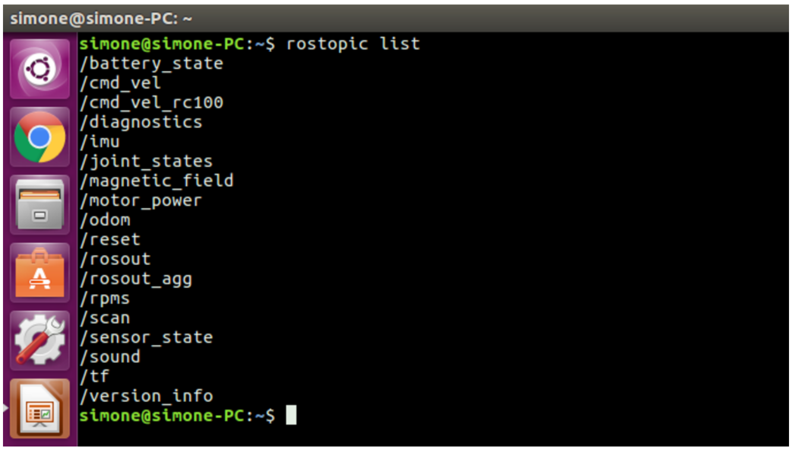


Figure 1.4: Topic list for a single robot

As shown in figure 1.4 while working with one set of unique-named topics there is no namespace collision and therefore the nodes can send and receive messages without any ambiguity.

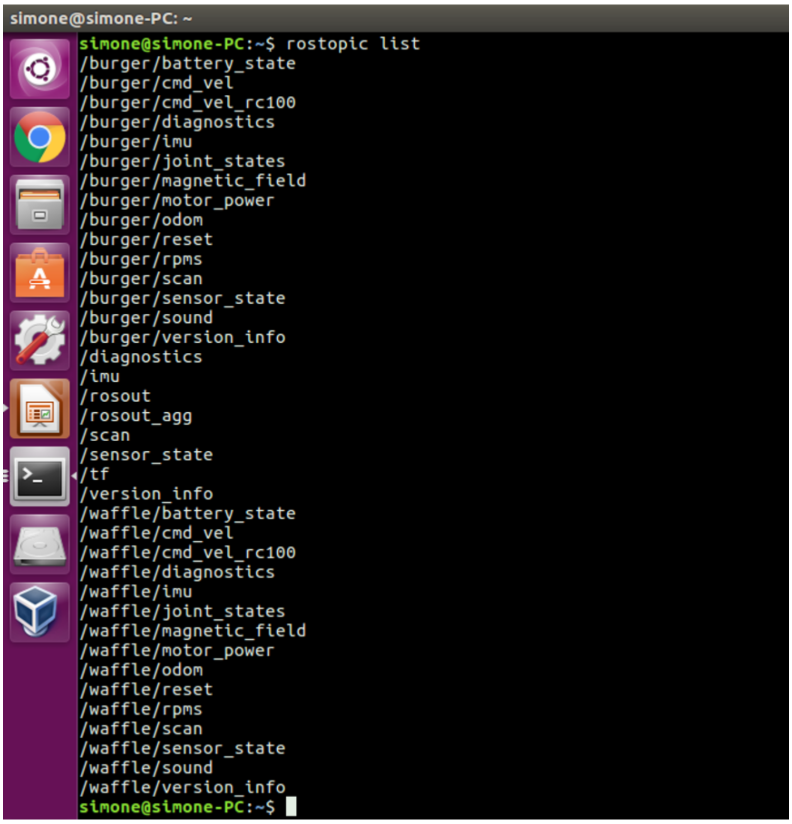


Figure 1.5: Topic list for two robots using NAMESPACE feature

However, when more than one robot is involved, as shown in figure 1.5, since the same set of topics exists for both robots, the NAMESPACE mechanism becomes necessary. For instance, the topic **/cmd_vel** used to control the speed of the rover has been renamed as **/burger/cmd_vel** for the *burger* model and as **/waffle/cmd_vel** for the *Waffle Pi* model. In this way, it is possible to control both speeds subscribing to the relative topic.

1.8 rosparam

Finally, a command that is really helpful and allows to save time when running scripts is **rosparam**. When the application needs a script that receive parameters as arguments for one or more functions this command can be used to set and change that parameter even while the script is running and, maybe, it is in a ros.spin loop. The structure for this command is :

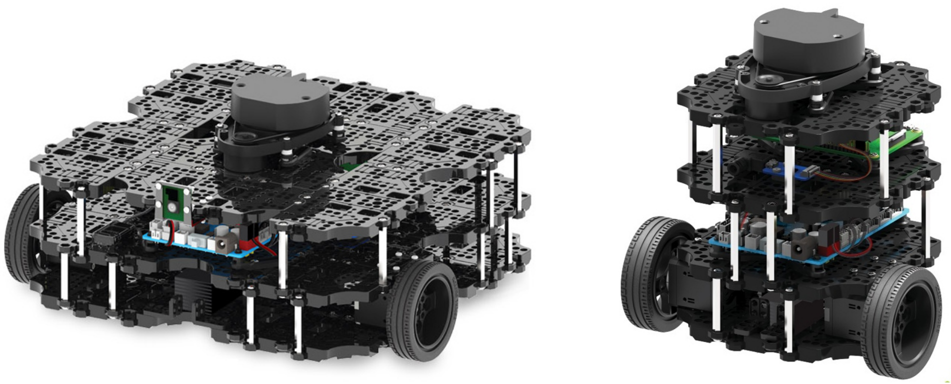
```
rosparam set "parameter_name"
```

This is helpful because allows the user to change the intended parameter without re-compiling the whole script with the new values and thus interrupting the ros loop.

Chapter 2

UGVs - TurtleBot3 and Jackal

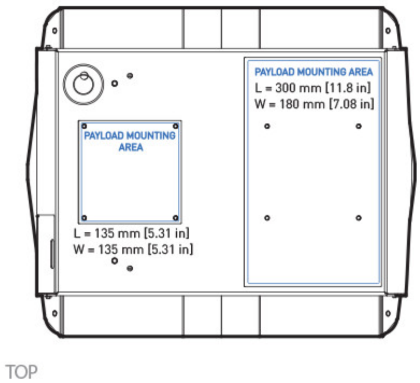
As mentioned before, through ROS software framework it is possible to control and program a large variety of robots, both UGVs and UAVs. With regards to UGVs, the models used for this thesis project are the TurtleBot3 rovers (Waffle and Burger, figure 2.1(a) and figure 2.1(b) and the Jackal by Clearpath Robotics [1] (figure 2.2).
The TurtleBots3 rovers are fully customizable robots thanks to their modular structure. As a matter of fact they support the installation of LIDAR scanners, cameras and other sensors useful for applications involving Simultaneous Localization and Mapping (SLAM) and autonomous navigation.
However, because of their structure and their specs, the TurtleBot3 robots are suitable for indoor operations only.
The Jackal, on the other hand, is a robotic research platform able to accomplish tasks in outdoor scenarios, thanks to its sturdy aluminum chassis made with a high torque 4×4 drivetrain that allows the robot to move around rugged terrains.



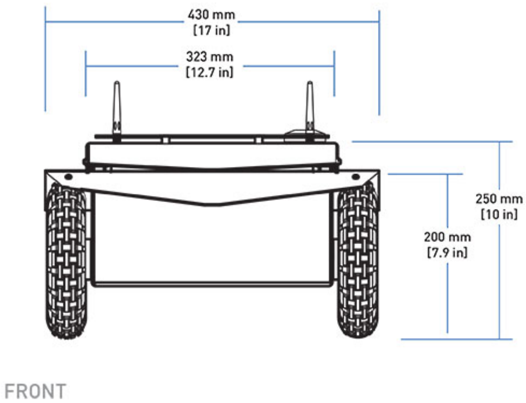
(a) Waffle model (b) Burger model

Figure 2.1: Turtlebot3 UGVs

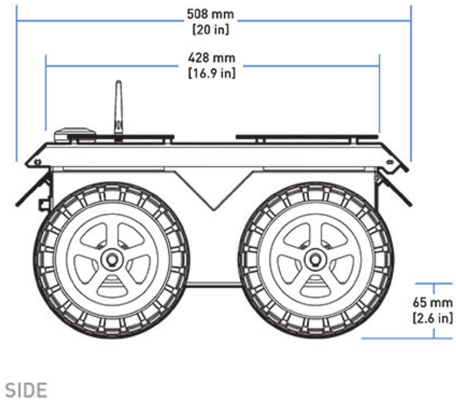
Figure 2.2: Jackal rover



(a) Jackal - top view



(b) Jackal - front view



(c) Jackal - side view

Figure 2.3: Jackal rover views

2.1 UGVs hardware specifications

In the following tables are shown the specifications of these UGVs:

Items	Specifications
Maximum translational velocity	0.26 m/s
Maximum rotational velocity	1.82 rad/s (104.27 deg/s)
Maximum payload	30kg
Size (L x W x H)	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1.8kg
Threshold of climbing	10 mm or lower
Expected operating time	2h
Expected charging time	2h 30m
SBC (Single Board Computers)	Intel® Joule™ 570x
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Actuator	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01
Camera	Intel® Realsense™ R200
IMU	3 Axis Accelerometer 3 Axis Gyroscope 3 Axis Magnetometer
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3
Programmable LEDs	User LED x 4
Status LEDs	Board status LED Arduino LED Power LED
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh/19.98Wh 5C
PC connection	USB
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Table 2.1: TurtleBot3 Waffle Hardware specifications

Items	**Specifications**
Maximum translational velocity	0.22 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)
Maximum payload	15kg
Size (L x W x H)	138mm x 178mm x 192mm
Weight (+ SBC + Battery + Sensors)	1kg
Threshold of climbing	10 mm or lower
Expected operating time	2h 30m
Expected charging time	2h 30m
SBC (Single Board Computers)	Raspberry Pi 3 Model B and B+
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Actuator	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01
IMU	3 Axis Accelerometer 3 Axis Gyroscope 3 Axis Magnetometer
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3
Programmable LEDs	User LED x 4
Status LEDs	Board status LED Arduino LED Power LED
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh/19.98Wh 5C
PC connection	USB
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Table 2.2: TurtleBot3 Burger Hardware specifications

Items	Specifications
External dimensions	508 x 430 x 250 mm (20 x 17 x 10 in)
Internal dimensions	250 x 100 x 85 mm (10 x 4 x 3 in)
Weight	17 kg
Maximum payload	20 kg
Max speed	2.0 m/s
Run time (basic usage)	4 hours
User power	5V at 5A, 12V at 10A, 24V at 20A
Drivers and APIs	ROS, Mathworks
SBC (Single Board Computers)	Raspberry Pi 3 Model B and B+

Table 2.3: Jackal Hardware specifications

2.2 Driver/follower application example

Thanks to ROS and the tools that comes with it, it is possible to make these different kind of robots work together at the same time and on the same application. For instance, thanks to the *namespace* tool the rovers can cooperate and move around sharing the same input commands or using different data, whichever case is needed for the application.
In the following code is presented a case of "driver/follower" application, where a robot (doesn't matter which one) is chosen to cover the role of *driver* and the others are the *followers*. The procedure for this task is the following: the master sends the inputs command to the *driver* which not only follows the master instructions, but also shares these inputs with the *followers*. At this point the *followers* that are subscribed to the *driver* node can use the same inputs and therefore execute the same function.

```
#! /usr/bin/env python

import rospy
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import Point, Twist

x = 0.0
y = 0.0
theta = 0.0

def newOdom (msg) :

global x
global y
global theta
```

```
17 x = msg.pose.pose.position.x
18 y = msg.pose.pose.position.y
19 rot_q = msg.pose.pose.orientation
20 (roll, pitch, theta ) = euler_from_quaternion ([rot_q.x,rot_q.y,rot_q.z,rot_q.w])
21
22 rospy.init_node("driver_controller")
23 sub = rospy.Subscriber("driver/odom", Odometry, newOdom)
24 pub = rospy.Publisher("driver/cmd_vel", Twist, queue_size=1)
25
26 speed = Twist()
27 r = rospy.Rate(4)
28 while not rospy.is_shutdown() :
29   angle_to_goal = 0.4
30 while (x < 0.2 and y < 0.2) :
31   if abs (angle_to_goal - theta) > 0.1 :
32     speed.linear.x = 0.0
33     speed.angular.z = 0.2
34   else :
35     speed.linear.x = 0.1
36     speed.angular.z = 0.0
37     pub.publish(speed)
38     r.sleep()
39 speed.linear.x = 0.0
40 speed.angular.z = 0.0
41 pub.publish(speed)
42
```

This part of code regards the *driver* part of the application. The most important part of this code is from line 26 to line 28. In these three lines the *driver* node is created (line 26), the subscription to the **Odometry** node is made in order to get the odometry data from the sensors (line 27) and finally the publisher on the `driver/cmd_vel` topic is placed to send commands to the robot and make it move accordingly (line 28).
In the rest of the code, in particular after line 28, the parameter of the task are set and both the linear and angular velocity for the robot are chosen.

Concerning the *followers*,on the other hand, the following code is utilized:

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist

x = 0.0
y = 0.0
theta = 0.0

def newTwist(msg) :
global x
global y
global theta

x = msg.linear.x
y = msg.linear.y
theta = msg.angular.z

rospy.init_node("follower_controller")
sub = rospy.Subscriber("driver/cmd_vel", Twist, newTwist)
pub = rospy.Publisher("follower/cmd_vel", Twist, queue_size=1)

speed_waffle = Twist()
r = rospy.Rate(45)
while not rospy.is_shutdown() :
  while (x != 0 or y != 0 or theta != 0) :
    speed_waffle.linear.x = x
    speed_waffle.linear.y = y
    speed_waffle.angular.z = theta
    pub.publish(speed_waffle)
    r.sleep()

  else:
    speed_waffle.linear.x = 0
    speed_waffle.linear.y = 0
    speed_waffle.angular.z = 0
    pub.publish(speed_waffle)
    r.sleep()

```

In this case, a *follower* node is created (line 21), the *driver* node is subscribed to (line 22) and the publisher is placed with respect to the node `follower/cmd_vel` that takes care of the velocity commands.
After that, a *while* loop is created (from line 34 to the end) that has the task of checking whether the *driver* robot is moving or not, in the first case this loop will command the follower to copy these movements, otherwise it shuts down the *follower* and stop any kind of velocity inputs.
The following graph is the ROS graph related to this kind of application. In order to avoid clarity issues, the graph is divided in two parts :

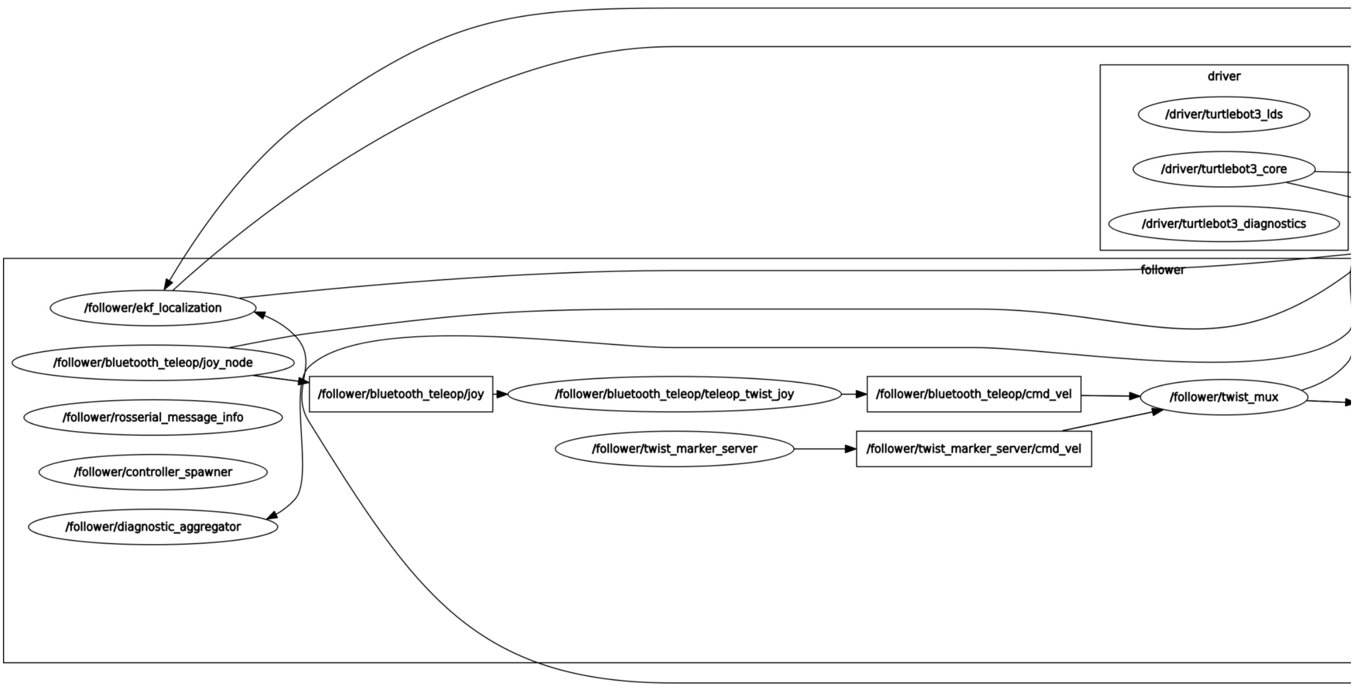


Figure 2.4: ROS graph for the driver/follower application (first half)

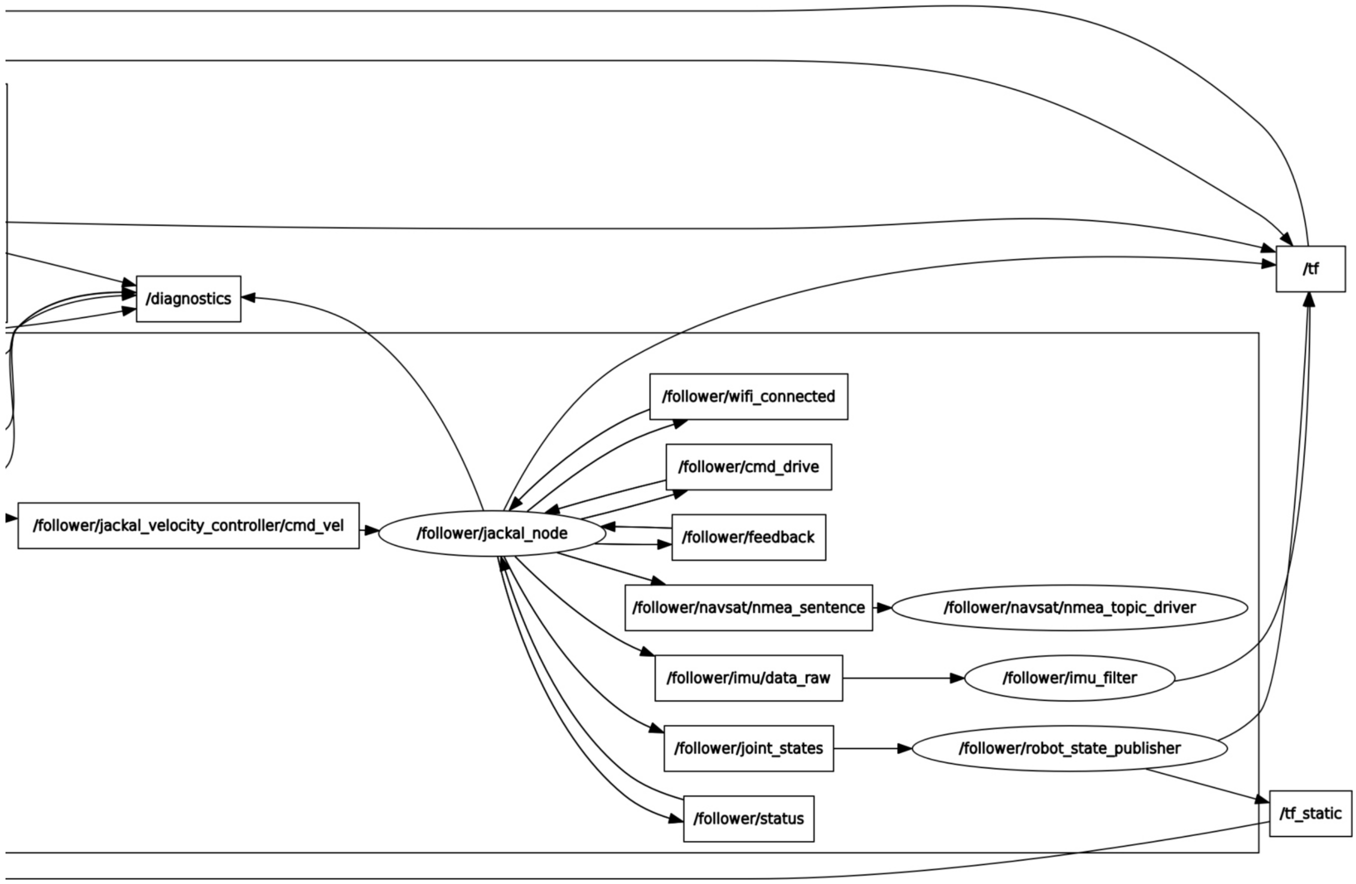


Figure 2.5: ROS graph for the driver/follower application (second half)

As it is shown in figure 2.4 on the left and bottom part of the graph there are all the nodes and topics concerning the follower robot. On the other hand, in figure **??** in the center at the top are placed the nodes and topics for the driver robot and all these nodes are linked to the **tf** topic, that, as said before, manages the coordinate transformations.

Chapter 3

QGroundControl GCS

When dealing with drones mission planning and setting, the usage of a Ground Control Station (GCS) is almost mandatory since these software allow to have the full control of the robot and to customize the parameters according to the type of application that is being implemented.
A GCS communicates with the UAV or UGV via wireless telemetry (for parameters setting a USB connection is also suitable but in this case the arming of the motors wont be possible since, for obvious reasons, a robot linked with a USB cable is not safe to move freely without risks).
These software are able to display real-time data about the robot position and performances and can be seen as a "virtual cockpit". On the most advanced and complex missions, a GCS can also be used to control the robot during the flight (UAV applications) uploading new mission commands and modifying the mission parameters.
The choice when dealing with such software is wide, some examples of GCS platforms are *Mission Planner*, *QGroundControl*,*APM Planner 2.0*, *MAVProxy*, *UgCS* etc.
As concerns this thesis project, QGroundControl [10] has been chosen since it is one of the most stable and reliable GCS and it is available for every kind of platform, both desktop and mobile.

3.1 QGroundControl GUI

In this part the various sections of QGroundControl are taken into consideration and analysed.
The main section that is showed when the autopilot is connected is represented in figure 3.1:

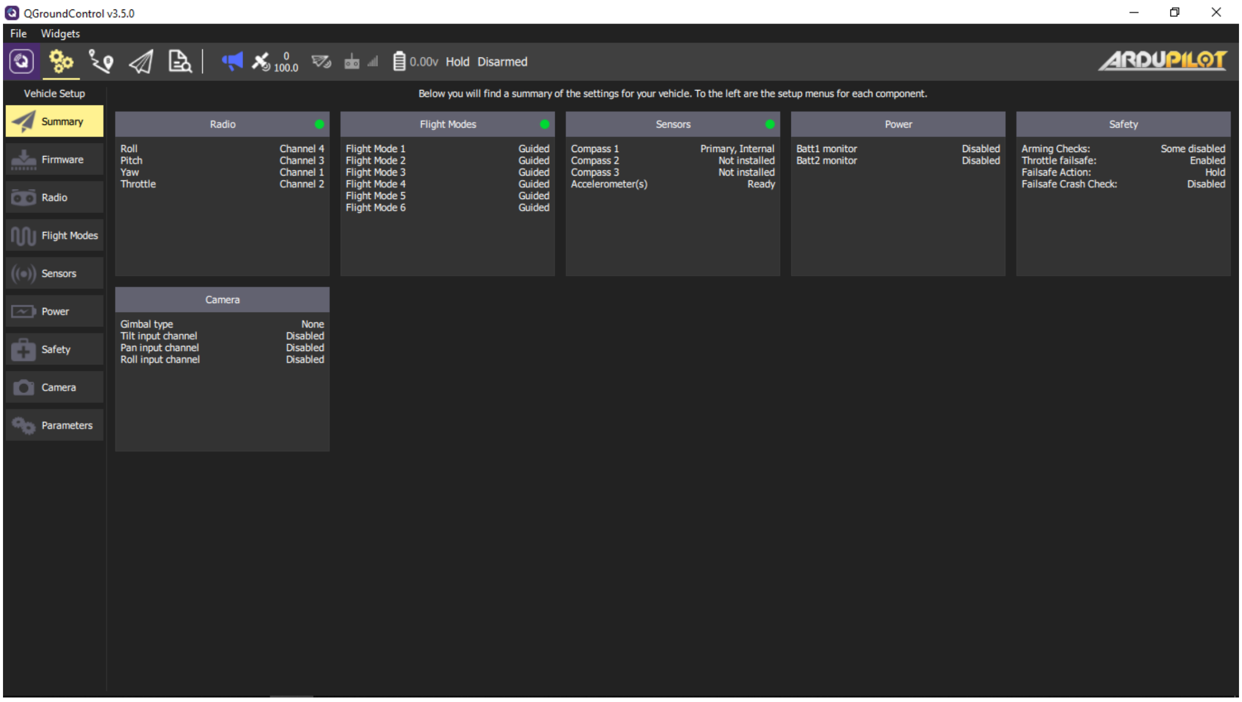


Figure 3.1: QGroundControl main section

This section serves as a sort of "recap" for the settings that are currently utilized for the definition of the mission. In, particular:

- **Radio** panel gathers all the channels configured for the raw, pitch, roll and throttle control.
- **Flight modes** panel shows which kind of mode is set for each possible flight configuration. For a detailed explanation of each modes, refer to Chapter 5, "MAVLink messages from GCS to autopilot" section.
- **Sensors** panel reports which sensor are currently operative on the autopilot.
- **Power** panel can be used to check, if needed, the status of the battery pf the vehicle used.
- **Safety** panel shows all the settings chosen for the various safety control parameters. Some of these safety blocks can be disables by changing the corresponding

parameter in the dedicated section of QGroundControl or by utilizing a safety switch linked to the autopilot that enable or disable the outputs to motors and servos.

- **Camera** panel gathers all the information about the kind of camera used, if any is present.

The next section is the one that regards the firmware selection for the autopilot:

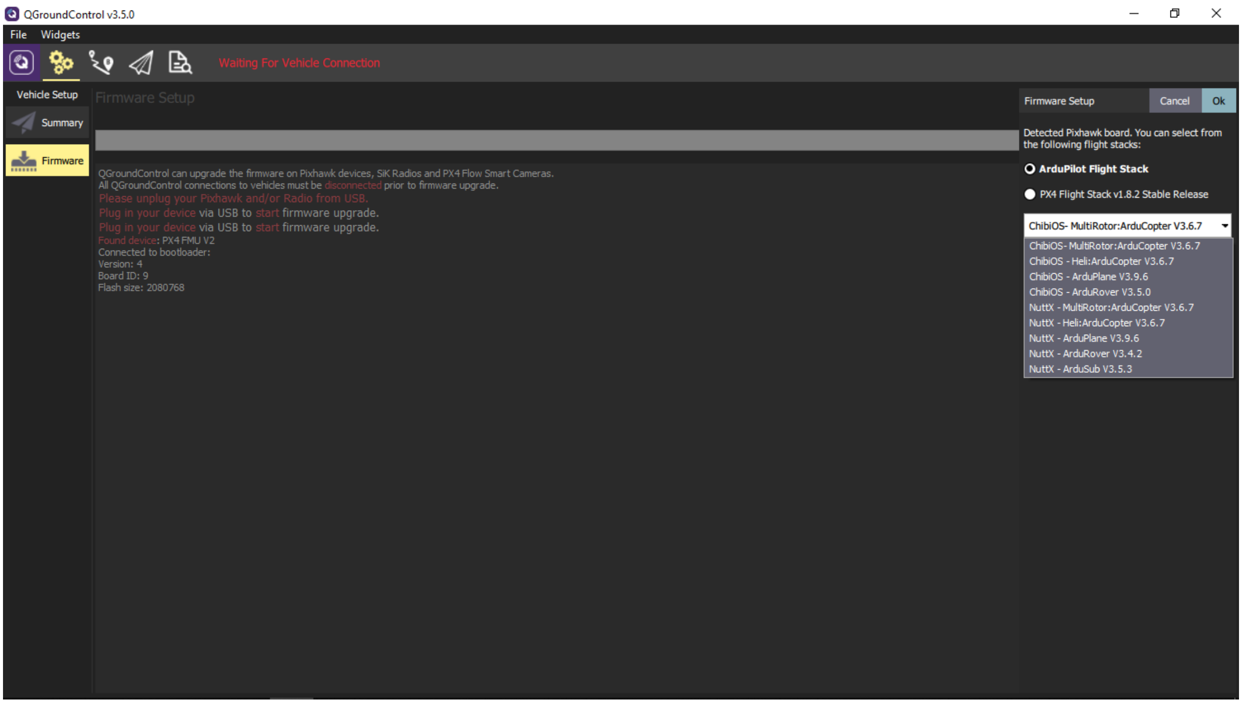


Figure 3.2: QGroundControl firmware section

As shown in figure 3.2 (on the right side), once the autopilot is connected via USB port, it is possible to chose between ArduPilot Flight Stack or PX4 Flight Stack, and each of these firmware has multiple sub-choices that change according to the kind of robot that the user is going to utilize for his application.

After this section, it comes the one that takes care of radio calibration and radio parameters setting:

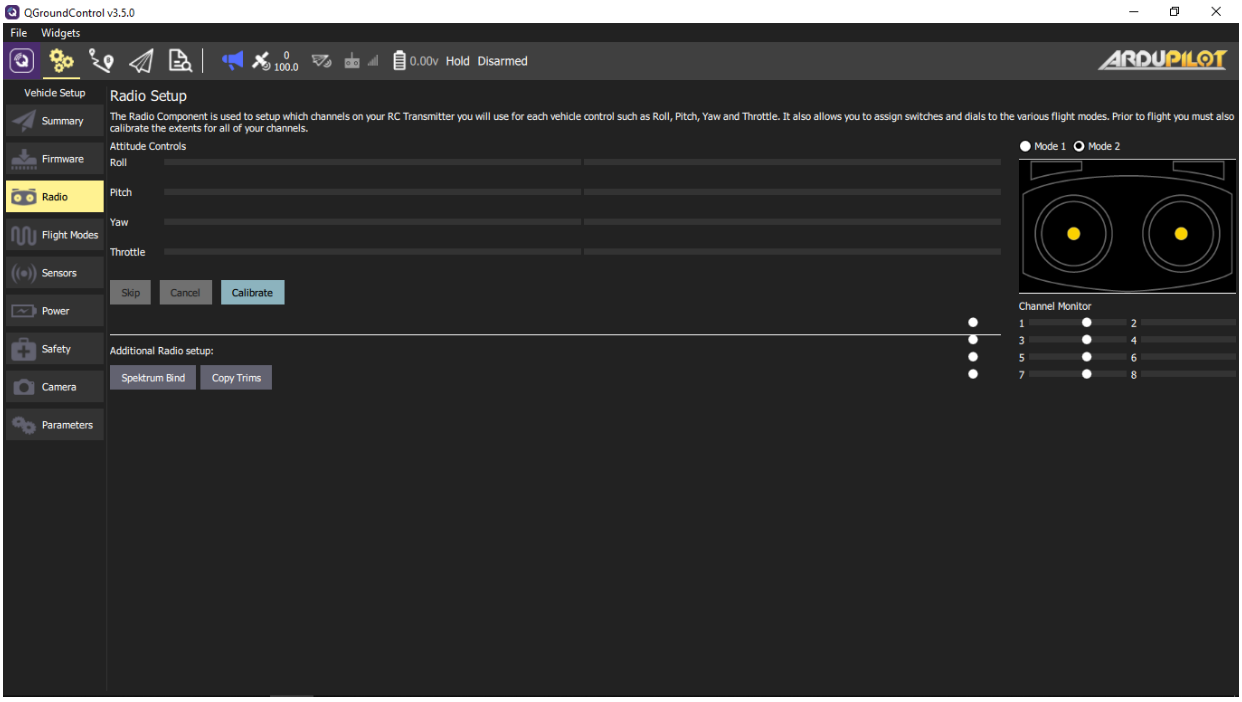


Figure 3.3: QGroundControl radio control section

This section is used during the calibration of the Radio Controller (RC) used to control the robot during the mission.
For this thesis project, a DX8 Spektrum RC (shown in figure 3.4) has been used:

Figure 3.4: DX8 Spektrum radio controller

The flight modes that the GCS provides to set the parameters of the mission are displayed in the next section of the GUI:

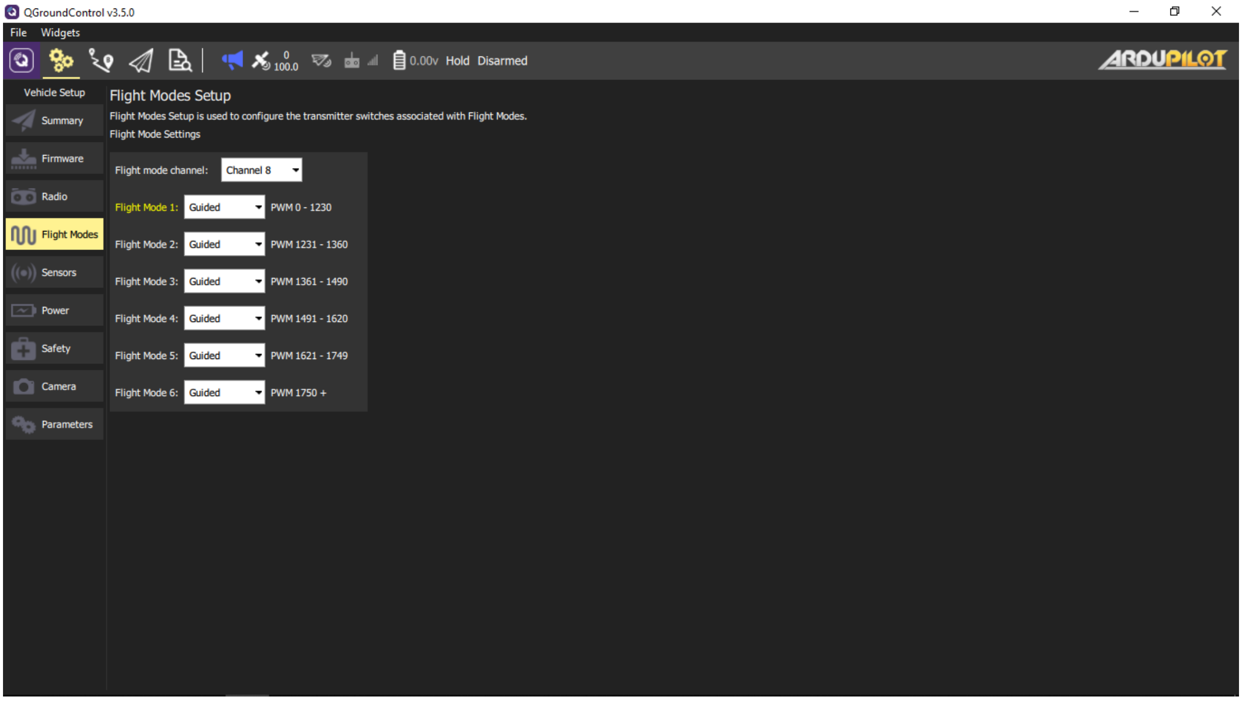


Figure 3.5: QGroundControl flight modes section

As shown in figure 3.5 all the modes has been set to GUIDED in order to have the full control of the vehicle without any of the restriction that other modes can be affected with.
The next relevant section is the one that allows to change the parameters for the mission and the robot. Some parameters can be changed from the GCS during the mission, while others need the vehicle to be disarmed and have to be set before the mission has started.
For this thesis application the most important parameters that have been changed are:

- **ARMING_CHECK**:allow to enable or disable various checks before the arming of the vehicle. Has ben set to NONE in order to speed up the arming procedure.
- **MODE 1-6**:allow to chose the flight mode for the mission. Changed to GUIDED for the reasons explained before.
- **SERIAL1_BAUD**:allow to set the right baud rate for serial port communication, the value has been set to 57600.
- In addition to that, several changes has been done with regards to the SERVO1 and SERVO3 parameters. More information about this topic will be provided below.

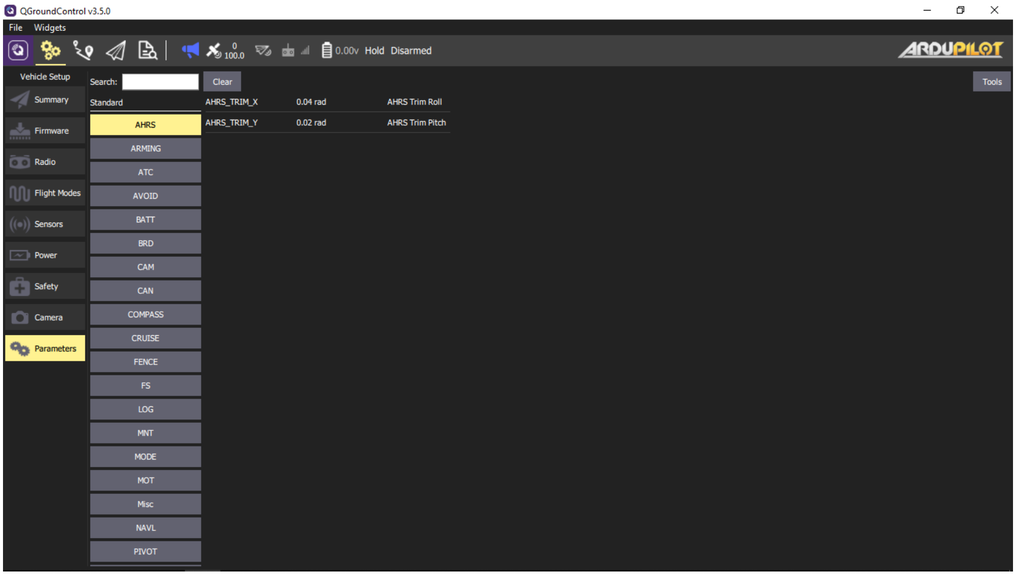


Figure 3.6: QGroundControl parameters section

Another important section is the map page where it is possible to control and check the position of the robot at any time, thanks to the fact that (usually) the vehicle is provided with a GPS system. This section of QGroundControl is showed in figure 3.7.

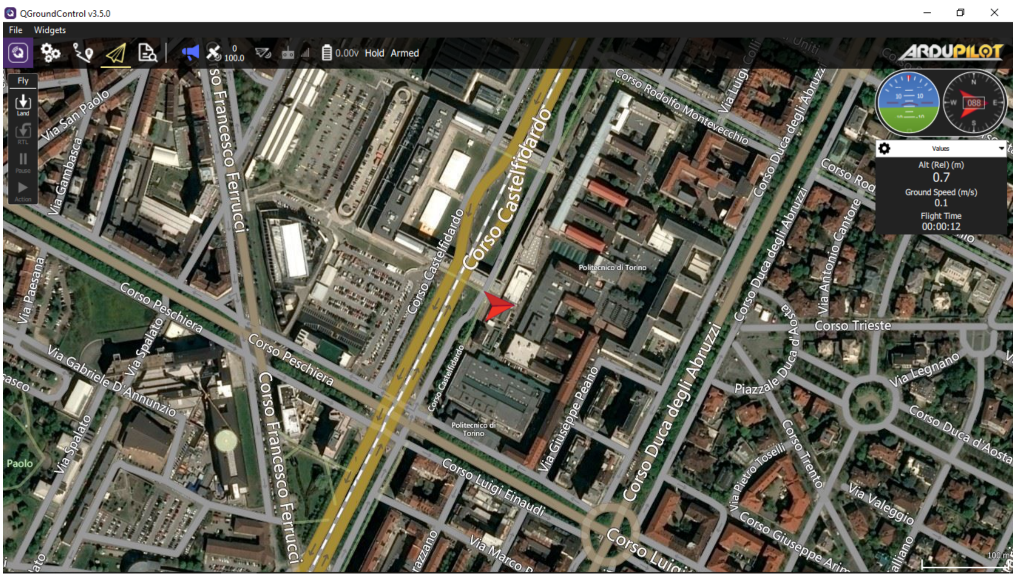


Figure 3.7: QGroundControl map section

3.2 SERIAL_BAUD and SERVO_FUNCTION parameters

Among the most important parameters to check and change according to the kind of robot and applications there are **SERIAL1_BAUD** and **SERVO_FUNCTION**.
With **SERIAL1_BAUD** the user can set the correct baud rate for the serial port where the autopilot is connected, both via USB or via radio telemetry.
Usually the standard value for this parameter is 57600 and can be changed taking into consideration that the lower is the value chosen, the lower is the chance for errors to occur, although it will also be slower the update of the GCS.
This important parameter can be changed in the apposite section as shown in figure 3.8

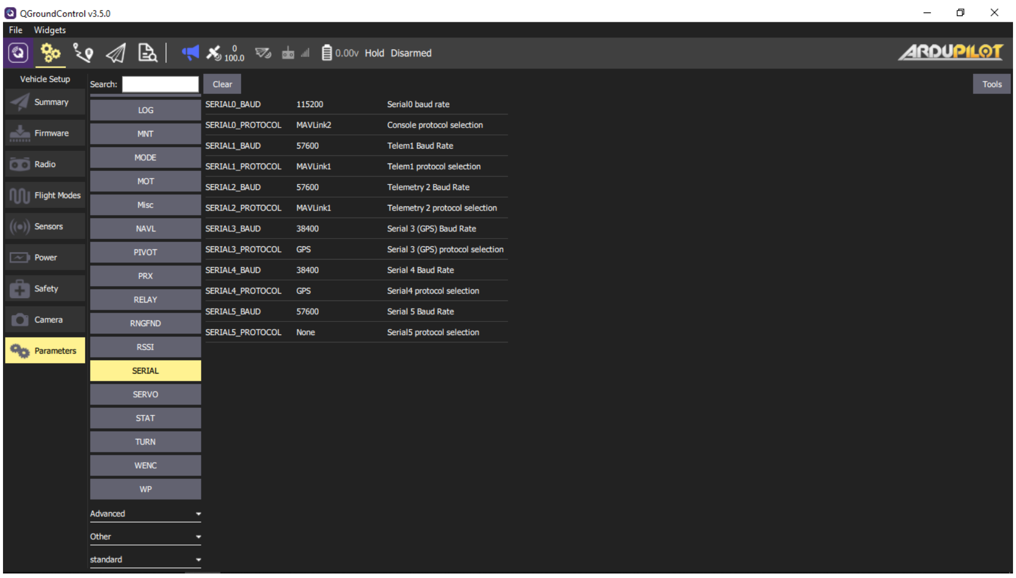


Figure 3.8: Baud rate parameters section

On the other hand, **SERVO_FUNCTION** allows to choose the function for the servo output. This is crucial in order to obtain the desired behaviour when trying to move the robot. For instance, if the application involves drones this parameter may be set to 4 to obtain an *Aileron* function.
When dealing with rovers, however, the *GroundSteering* and *Throttle* functions are required and so the correspondent values for **SERVO_FUNCTION** parameter have been chosen (26 and 70 respectively), as shown in figure 3.9.
For the list of values to assign to this parameter, table 3.1 can be referred.

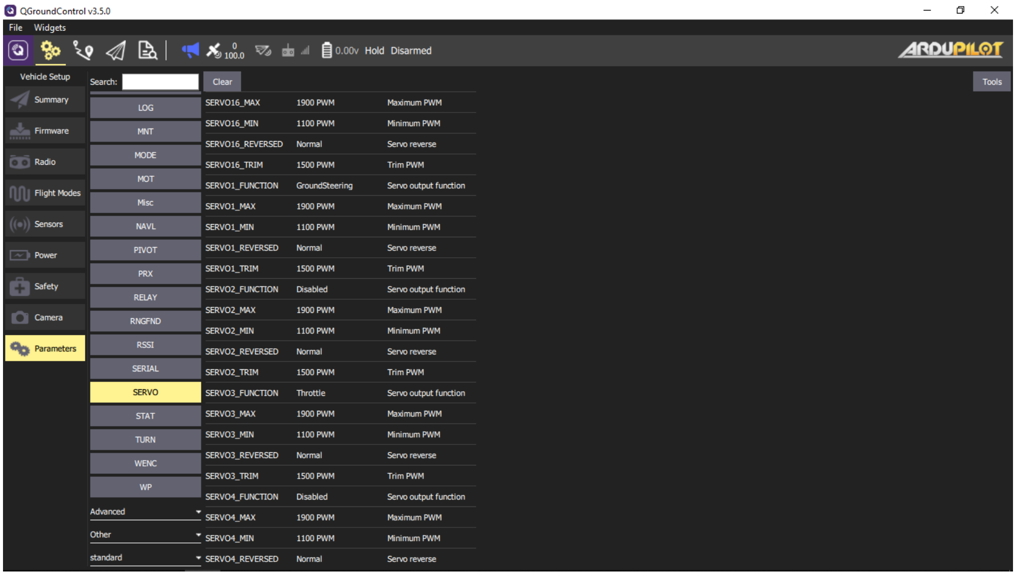


Figure 3.9: SERVO parameters section

FUNCTION	**VALUE**
FLAP	2
AILERON	4
ELEVATOR	19
RUDDER	21
STEERING	26
THROTTLE	70
THROTTLE_LEFT	73
THROTTLE_RIGHT	74
ELEVON_LEFT	77
ELEVON_RIGHT	78
VTAIL_LEFT	79
VTAIL_RIGHT	80

Table 3.1: List of values for SERVO_FUNCTION parameter

Chapter 4

Pixhawk autopilot and RC testing phase

Even if autopilots like Pixhawk or NAVIO2 [6] are usually used to manage applications regarding drones, it is possible to use them also with UGVs.
In particular, the Pixhawk is an advanced open-hardware autopilot capable of powering all kinds of vehicles from racing and cargo drones through to ground vehicles and submersibles. This autopilot can be flashed with different type of firmware (e.g. PX4, APM) and when used along with a Ground Control Station (GCS) can be utilized to control a large variety of frames.
Another hardware component utilized for this application is the OpenCR1.0 [7] robot controller that is embedded with a powerful MCU from the ARM Cortex-M7 line-up. This board supports RS-485 and TTL to control the Dynamixels (the UGV wheels motors), and offers UART, CAN and a variety of other communication environment. Moreover, development tools such as Arduino IDE are available as well. This board has been used in order to traduce the PWM signals produced by the Pixhawk and control the motors actuators sending the inputs via TTL serial ports.

4.1 Pixhawk and OpenCR hardware specifications

In the following pages are reported the specifics for the Pixhawk autopilot and the OpenCR1.0 board:

Figure 4.1: Pixhawk connectors

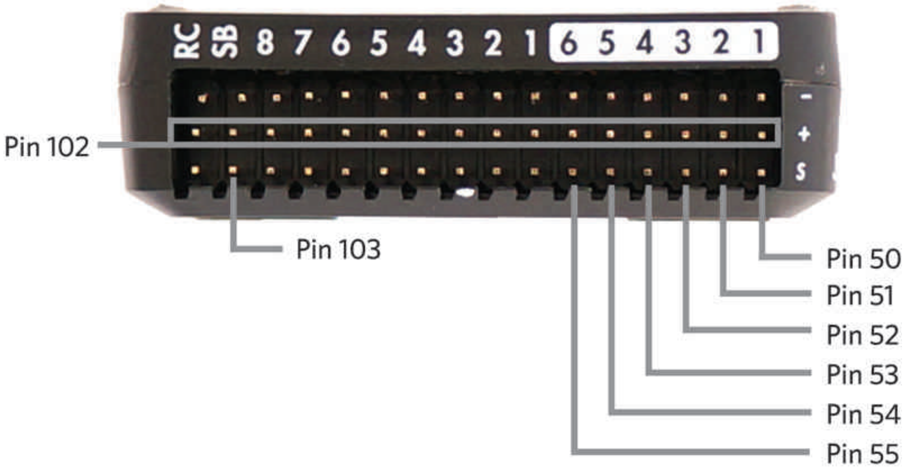


Figure 4.2: Pixhawk pinout

Items	**Specifications**
Processor	32-bit ARM Cortex M4 core with FPU 168 Mhz/256 KB RAM/2 MB Flash 32-bit failsafe co-processor
Sensors	MPU6000 as main accel and gyro ST Micro 16-bit gyroscope ST Micro 14-bit accelerometer/compass (magnetometer) MEAS barometer
Power	Ideal diode controller with automatic failover Servo rail high-power (7 V) and high-current ready All peripheral outputs over-current protected, all inputs ESD protected
Interfaces	5x UART serial ports, 1 high-power capable, 2 with HW flow control Spektrum DSM/DSM2/DSM-X Satellite input Futaba S.BUS input (output not yet implemented) PPM sum signal RSSI (PWM or voltage) input I2C, SPI, 2x CAN, USB 3.3V and 6.6V ADC inputs
Dimensions	Power module output: 4.9 5.5V USB Power Input: 4.75 5.25V Servo Rail Input: 0 36V
Weight	Weight 38 g (1.3 oz) Width 50 mm (2.0”) Height 15.5 mm (.6”) Length 81.5 mm (3.2”)

Table 4.1: Pixhawk Hardware specifications

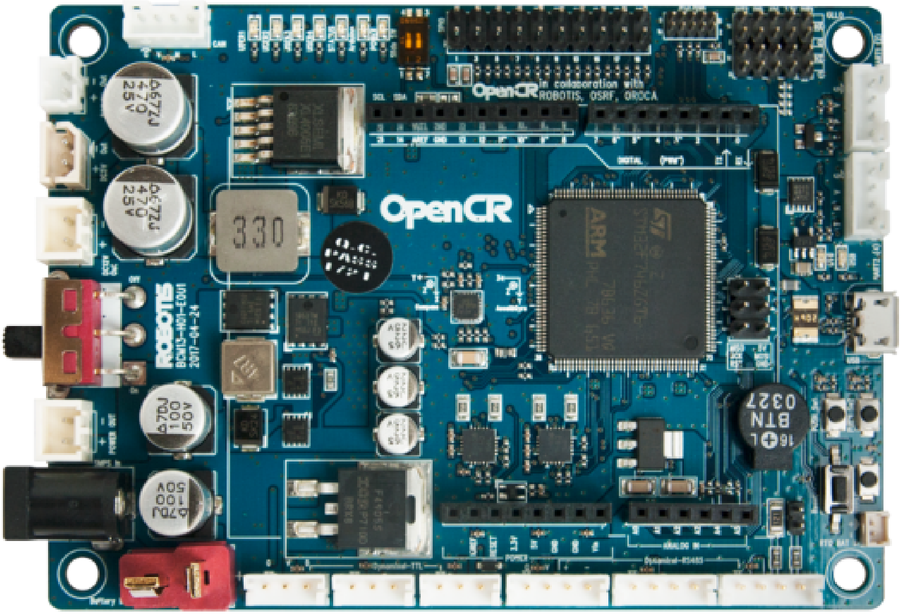


Figure 4.3: OpenCR1.0 controller

GPIO

1	3.3V	6	D53	11	D58	16	D63
2	GND	7	D54	12	D59	17	D64
3	D50	8	D55	13	D60	18	D65
4	D51	9	D56	14	D61	19	D66
5	D52	10	D57	15	D62	20	D67

JTAG

1	3.3V	6	JTDO_SWO
2	JTMS_SWDIO	7	GND
3	GND	8	JTDI
4	JTCK_SWCLK	9	GND
5	GND	10	MCU_NRESET

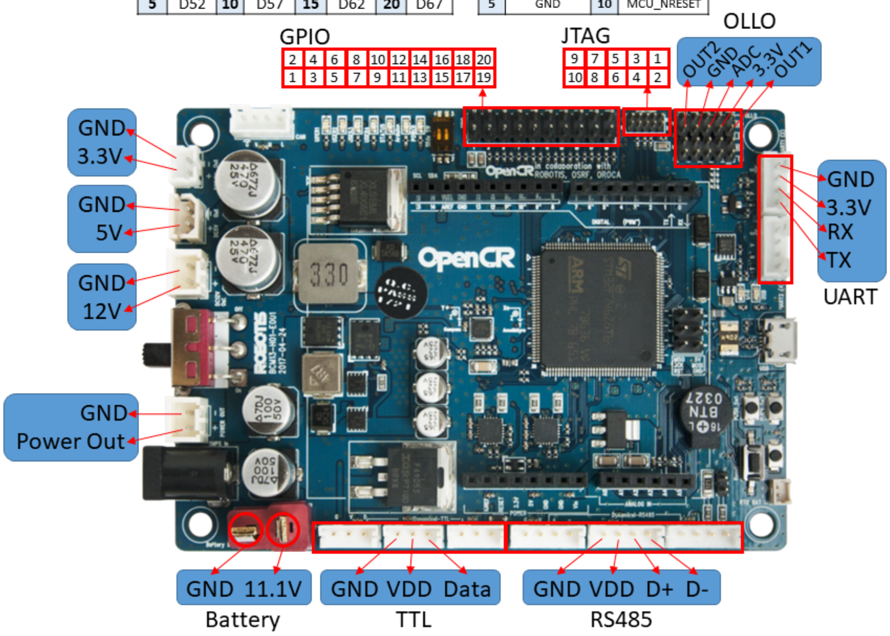


Figure 4.4: OpenCR1.0 pin map

Items	Specifications
Microcontroller	STM32F746ZGT6 / 32-bit ARM Cortex®-M7 with FPU (216MHz, 462DMIPS)
Sensors	Gyroscope 3Axis Accelerometer 3Axis Magnetometer 3Axis
Programmer	ARM Cortex 10pin JTAG/SWD connector USB Device Firmware Upgrade (DFU) Serial
Digital I/O	32 pins (L 14, R 18) *Arduino connectivity 5Pin OLLO x 4 GPIO x 18 pins PWM x 6 I2C x 1 SPI x 1
Analog INPUT	ADC Channels (Max 12bit) x 6
Communication Ports	USB x 1 (Micro-B USB connector/USB 2.0 /Host/Peripheral/OTG) TTL x 3 (B3B-EH-A / Dynamixel) RS485 x 3 (B4B-EH-A / Dynamixel) UART x 2 (20010WS-04) CAN x 1 (20010WS-04)
LEDs and buttons	LD2 (red/green) : USB communication User LED x 4 : LD3 (red), LD4 (green), LD5 (blue) User button x 2 Power LED : LD1 (red, 3.3 V power on) Reset button x 1 (for power reset of board) Power on/off switch x 1
Input Power Sources	5 V (USB VBUS), 7-24 V (Battery or SMPS) Default battery : LI-PO 11.1V 1,800mAh 19.98Wh Power LED : LD1 (red, 3.3 V power on) Default SMPS : 12V 4.5A External battery Port for RTC (Real Time Clock) (Molex 53047-0210)
Output Power Sources	12V max 4.5A(SMW250-02) 5V max 4A(5267-02A), 3.3V@800mA(20010WS-02)
Dimensions	105(W) X 75(D) mm
Weight	60g

Table 4.2: OpenCR1.0 Hardware specifications

4.2 QGroundControl GCS

In order to use the Pixhawk with rovers, the first step is to flash the correct firmware using a Ground Control Station (GCS) software. For this thesis project, *QGroundControl* has been chosen.
As explained in Chapter 3, these kind of software allow the user to interface with the autopilot, to change the parameters, to flash the desired firmware and to control the flight or path of the robots. The procedure to set up the task environment is the following:

- Open the GCS and connect the Pixhawk via USB or via wireless telemetry
- Select the desired firmware to upload on the autopilot. As concerns this application, the choice was between PX4 and ArduPilot firmware, and the latter has been used.
- Select the airframe that better represents the typology of the robot used for the mission. As shown in figure 4.5 there are plenty of options for drones and each robot frame has multiple sub-frames available to be chosen.
  For the sake of this project, the rover frame has been selected.

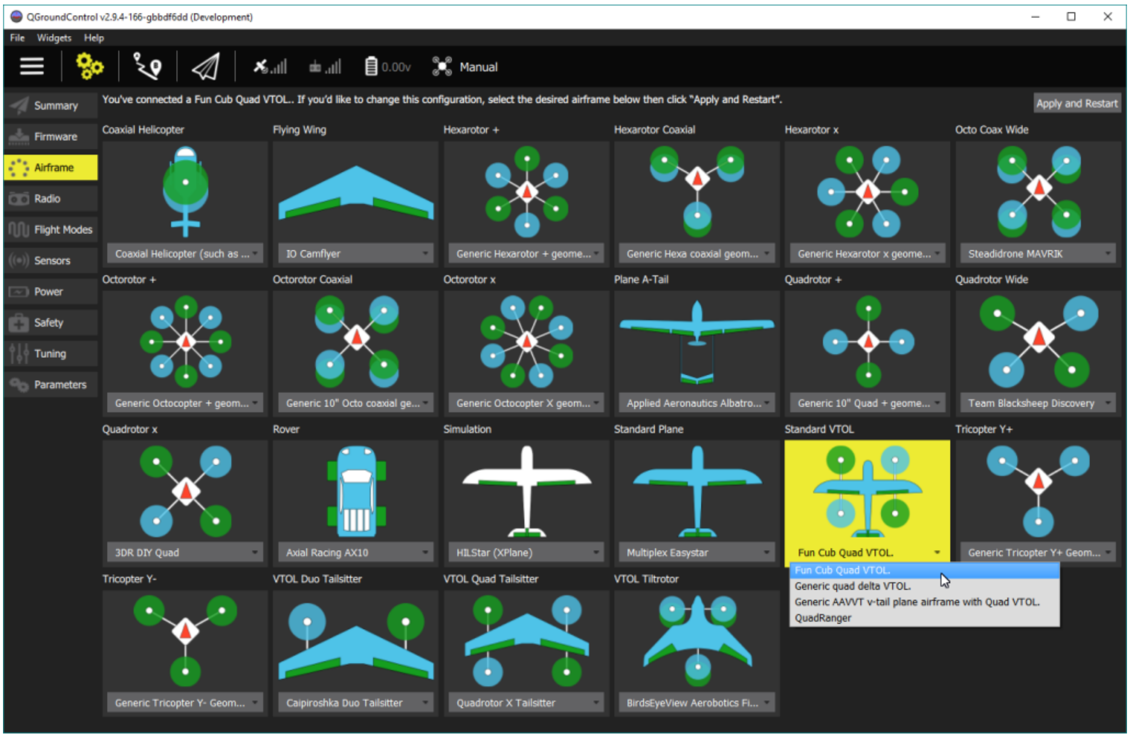


Figure 4.5: Frames selection section of QGroundControl

- The next step is to complete the calibration procedure. With this process the GCS collects all the data needed to initialize all the parameters for the mission. Some of these parameters regards the sensors (accelerometer and gyroscope) while others are for the radio calibration. With regard to the sensors, the procedure requires to rotate the autopilot around all its axis and then to keep it still in certain positions. On the other end, for the radio calibration, the various switches of the RC have to be placed in every possible position so that the GCS can register the values for the min/max range.

- Finally,in the *Parameters* section of the GCS the user can modify the various parameters for the mission in order to customize the environment accordingly with his goals.

It is important to notice the difference between the high number of frames available for drones against the single one present for rovers.
This, as reported before, is due to the fact that usually these hardware and software are utilized for applications regarding drones and only a few times are taken into consideration for the control of rovers.

4.3 Control signals procedure and circuit design

Once the rover firmware has been flashed in the Pixhawk, the autopilot knows that wont have to deal with a drone airframe (meaning a structure with motors and helices) but will have to manage a system with motors and wheels, which means dealing with different types of inputs and different directions for the motion.
In order to proceed with the task as intended a further modification for the mission parameters is needed. In particular the two parameters SERVO1_FUNCTION and SERVO3_FUNCTION have to be set to values 20 and 76 respectively, meaning that pin n.1 (see figure 4.2 for the autopilot pinout) of the Pixhawk will generate the PWM that manages the *STEERING* and pin n. 3 the *THROTTLE*.
Once these preliminary steps are completed, the Pixhawk is connected to the controller board of the rover. The autopilot now sends PWM signals that are collected by the UART pins of the OpenCR1.0 (see figure 4.4) and translated by the Arduino that is embedded inside the board. In the following picture (figure 4.6) it is shown the circuit used to accomplish this task:

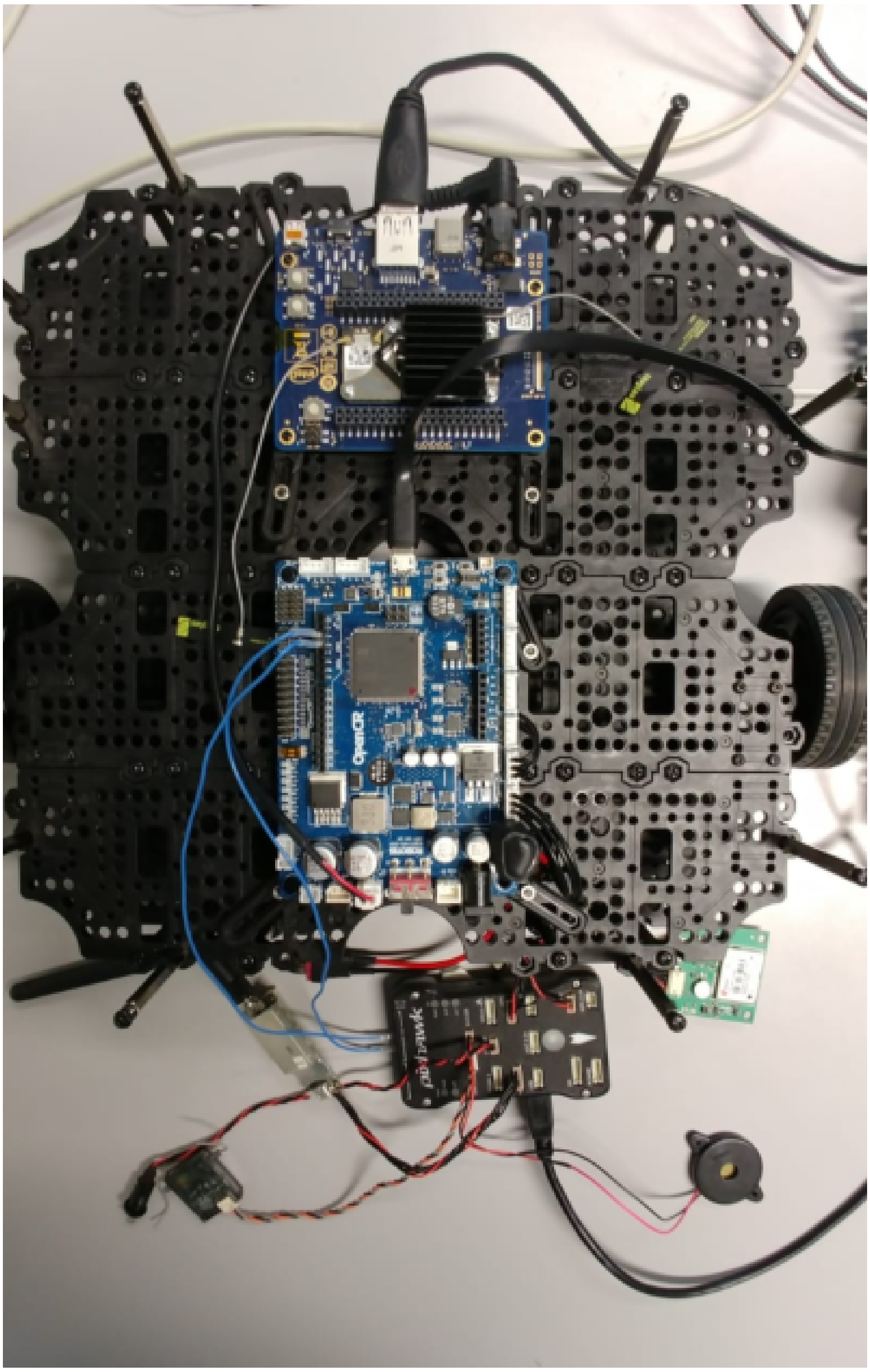


Figure 4.6: Circuit used for signals exchange between the Pixhawk and the motors

The components shown in figure 4.6 are:

- Turtlebot3 SBC (Single Board Computer) on the top
- OpenCR board embedded with MCU from the ARM Cortex-M7 line-up in the center
- Pixhawk autopilot (connected with jumper cables to the OpenCR) on the bottom

These signals are then redirected via TTL serial port to the servo motors that control the actuators of the wheels.
The full code that manages this translation is available on the Turtlebot3 official GitHub [16], here below there are the most significant lines that have been added or modified for this application:

```

#include "turtlebot3_core_config.h"
int durationLeft,durationRight;
int pinLeft=2;
int pinRight=3;

...

void loop()

{

durationLeft = pulseIn(pinLeft, HIGH);
durationRight = pulseIn(pinRight, HIGH);

DEBUG_SERIAL.print("durationLeft:");
DEBUG_SERIAL.println(durationLeft);
DEBUG_SERIAL.print("durationRight:");
DEBUG_SERIAL.println(durationRight);

goal_velocity[ANGULAR]=durationRight;
goal_velocity[LINEAR]=durationLeft;

DEBUG_SERIAL.print("Linear:");

DEBUG_SERIAL.println( goal_velocity[LINEAR]);

DEBUG_SERIAL.print("Angular:");

```

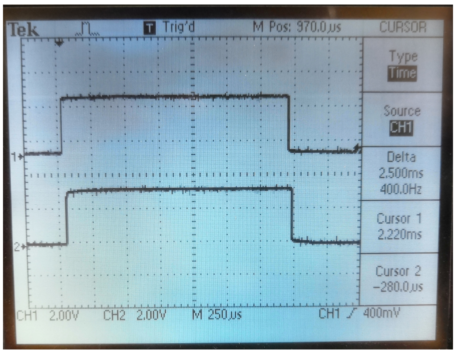
```
30 DEBUG_SERIAL.println(goal_velocity[ANGULAR]);
31
32
33 goal_velocity[ANGULAR]=map(goal_velocity[ANGULAR], —3484, 3484, —18, 18);
34
35 goal_velocity[LINEAR] =map(goal_velocity[LINEAR], 1500, 2000, —26, 26);
36
37
38 if( durationLeft==0 &&  durationRight==0 ) goal_velocity[LINEAR]=0;
39
40 goal_velocity[ANGULAR]=goal_velocity[ANGULAR]/10;
41
42 goal_velocity[LINEAR]=goal_velocity[LINEAR]/100;
43
44 ...
```

8

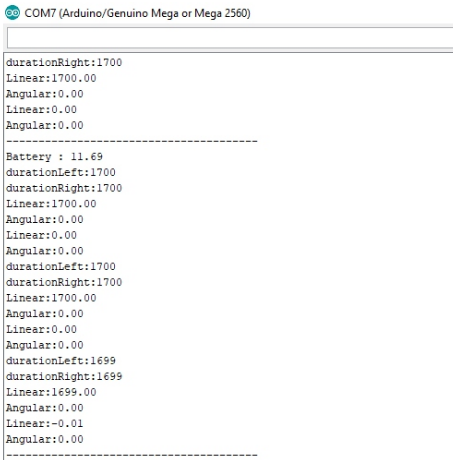
In lines 4 and 5 the input pins are selected (pins 2 and 3 in this case), those pins are the one that will be connected to the Pixhawk and will receive the PWM signals. In the loop starting at line 8, the values for the PWM signals are used to initialize the variables "durationLeft" and "durationRight". These two variables are then mapped (lines 38 and 40) in order to see the right range for the velocities, from -0.26 m/s to 0.26 m/s for the linear velocity and from -1.8 rad/s to 1.8 rad/s for the angular velocity (which are the minimum and maximum values reachable by the motors).
For this application the PWM values go from 1500 ms to 2000 ms (for practical purposes these two values have been rounded to 1600 and 1900, respectively).

4.4 RC control testing phase

Once the circuit is prepared it is possible to test it with a Radio Controller in order to understand if the signals sent and received are correct and to verify that the actuators work like intended.
With the purpose of having a second feedback along with the oscilloscope, an Arduino board has been used to monitor the signals passing through the serial port.
The results of this test are shown in the following figures:



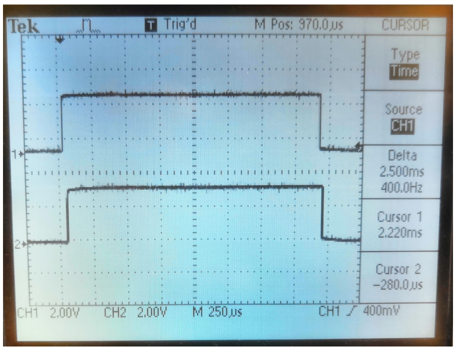
(a) PWM signals when no speed input is given



(b) PWM values and mapped values read by Arduino board (no speed input)

Figure 4.7: PWM values and Arduino readings when no speed input is given

In figure 4.7 it is shown the case in which no input signal is sent and so the rover is standing still. In this case on both pins is present a PWM of 1750 µs that is the "zero" value for this example. Moreover, in the Arduino serial monitor it can be seen that the linear and angular speed values are both to zero as expected.



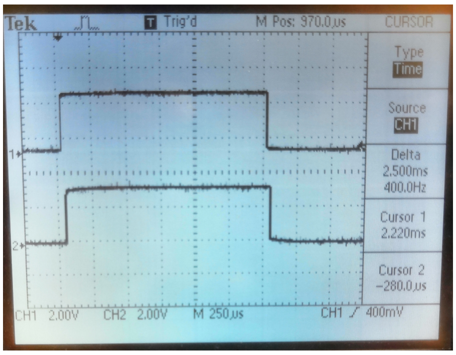
(a) PWM signals for maximum speed forward RC input

COM7 (Arduino/Genuino Mega or Mega 2560)

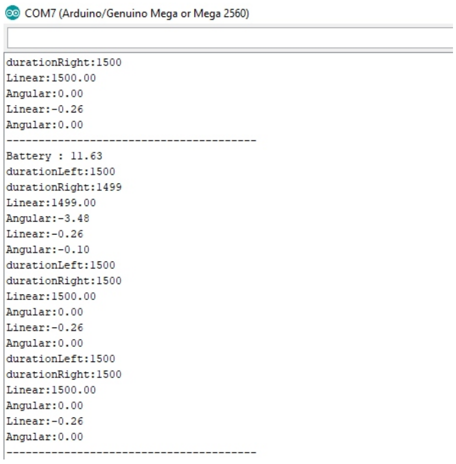
```
Angular:0.00
Linear:0.26
Angular:0.00
durationLeft:1900
durationRight:1900
Linear:1900.00
Angular:0.00
Linear:0.26
Angular:0.00
---------------------------------------
Battery : 11.65
durationLeft:1900
durationRight:1900
Linear:1900.00
Angular:0.00
Linear:0.26
Angular:0.00
durationLeft:1900
durationRight:1900
Linear:1900.00
Angular:0.00
Linear:0.26
Angular:0.00
durationLeft:1899
durationRight:1899
Linear:1899.00
Angular:0.00
Linear:0.25
Angular:0.00
---------------------------------------
```

(b) PWM values and mapped values read by Arduino board (maximum speed forward input)

Figure 4.8: PWM values and Arduino readings for pure linear forward speed



(a) PWM signals for maximum speed backwards RC input

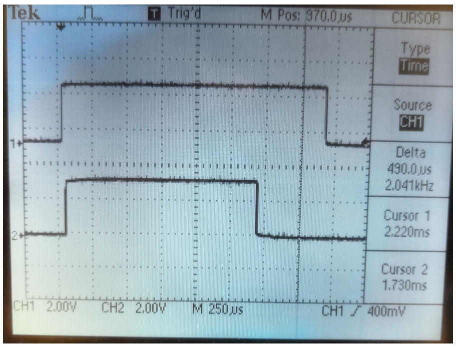


COM7 (Arduino/Genuino Mega or Mega 2560)

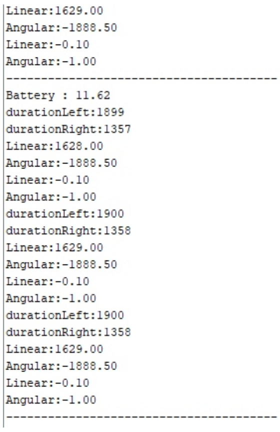
```
durationRight:1500
Linear:1500.00
Angular:0.00
Linear:-0.26
Angular:0.00
---------------------------------------
Battery : 11.63
durationLeft:1500
durationRight:1499
Linear:1499.00
Angular:-3.48
Linear:-0.26
Angular:-0.10
durationLeft:1500
durationRight:1500
Linear:1500.00
Angular:0.00
Linear:-0.26
Angular:0.00
durationLeft:1500
durationRight:1500
Linear:1500.00
Angular:0.00
Linear:-0.26
Angular:0.00
---------------------------------------
```

(b) PWM values and mapped values read by Arduino board (maximum speed backwards input)

Figure 4.9: PWM values and Arduino readings for pure linear backwards speed



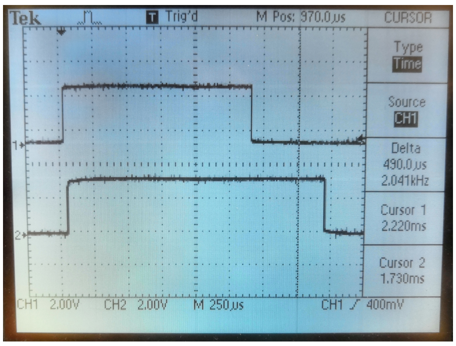
(a) PWM signals for right turn RC input



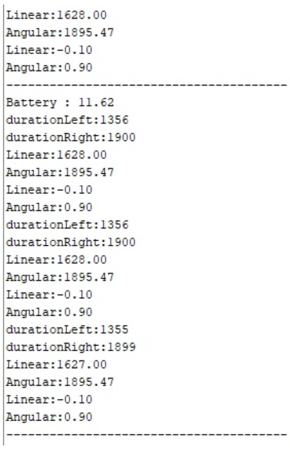
```
Linear:1629.00
Angular:-1888.50
Linear:-0.10
Angular:-1.00
----------------------------------------
Battery : 11.62
durationLeft:1899
durationRight:1357
Linear:1628.00
Angular:-1888.50
Linear:-0.10
Angular:-1.00
durationLeft:1900
durationRight:1358
Linear:1629.00
Angular:-1888.50
Linear:-0.10
Angular:-1.00
durationLeft:1900
durationRight:1358
Linear:1629.00
Angular:-1888.50
Linear:-0.10
Angular:-1.00
----------------------------------------
```

(b) PWM values and mapped values read by Arduino board (turn right input)

Figure 4.10: PWM values and Arduino readings for angular speed (turn right)



(a) PWM signals for left turn RC input



```
Linear:1628.00
Angular:1895.47
Linear:-0.10
Angular:0.90
----------------------------------------
Battery : 11.62
durationLeft:1356
durationRight:1900
Linear:1628.00
Angular:1895.47
Linear:-0.10
Angular:0.90
durationLeft:1356
durationRight:1900
Linear:1628.00
Angular:1895.47
Linear:-0.10
Angular:0.90
durationLeft:1355
durationRight:1899
Linear:1627.00
Angular:1895.47
Linear:-0.10
Angular:0.90
----------------------------------------
```

(b) PWM values and mapped values read by Arduino board (turn left input)

Figure 4.11: PWM values and Arduino readings for angular speed (turn left)

In figure 4.8 and 4.9 are represented the two cases in which the rover moves at maximum speed forward (4.8) and at maximum speed backwards (4.9). In the first case the PWM value reaches 1900 µs (rounded value for 2000 µs) while in the second case the value is 1600 µs (rounded for 1500 µs). In both cases the corresponding mapped values can be seen in the Arduino serial monitor (0.26 m/s is the maximum speed reachable by the rover used for the test).
Finally, in figure 4.10 and 4.11 are exhibited the two cases in which the rover performs a turn right (4.10) or left (4.11). For this purpose it can be seen that the wheels are spinning in different direction and thus the rotation is performed (one PWM is above the "zero" threshold of 1750 µs while the other one is below it). Once again, the Arduino serial monitor shows the remapped values for the linear and angular velocities.

Chapter 5

MAVROS package and MAVLink protocol

The next step is to use the MAVLink protocol along with MAVROS libraries instead of using the RC to control the rover. The goal is to be able to override the RC controls and use only the computer to set the parameters and send the commands.
This procedure requires MAVROS package and MAVLink protocol, which are tools that, when used together, allow to build a communication bridge between the computer, the GCS and the autopilot.

5.1 MAVROS

The MAVROS package provides communication driver for different kinds of autopilots that use MAVLink protocol. In particular, allows the user to program and customize the parameters of the autopilot utilizing the ROS environment and replacing by all means any kind of GCS software.
The first step after the installation of MAVROS package is to run the right command to launch the main node, and this command changes depending on which kind of firmware is present on the autopilot that is being used.
If a PX4 stack has been chosen, the corresponding launch command is:

```
roslaunch mavros px4.launch
```

On the other hand, if an ArduPilot/APM stack has been chosen the command becomes:

```
roslaunch mavros apm.launch
```

At this point, the ROS environment is set and all the topics and nodes are ready to be used, as shown in the following figure:

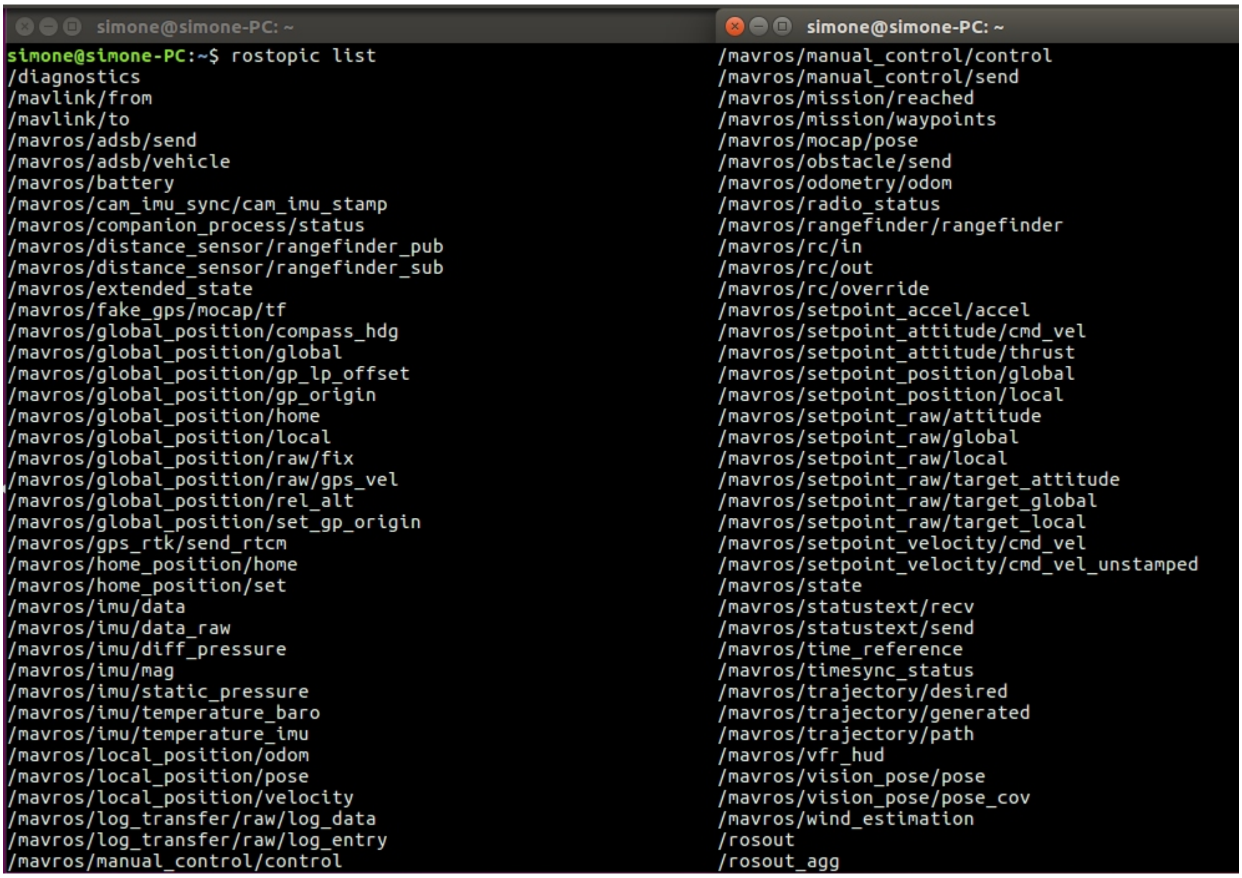


Figure 5.1: MAVROS topics list

In figure 5.1 is is represented all the topics that are generated once the MAVROS node is launched.
The most significant topics are:

- **/mavlink/from** and **/mavlink/to** that enables the Mavlink stream from and to the autopilot
- **/mavros/state** that allow to check the state of some parameters of the autopilot (connection, arming, mode)

- **/mavros/set_point_velocity/cmd_vel** that is useful to track the values sent to the autopilot for the velocity control of the robot

- **/mavros/rc/override** that enables the override for the RC control and allow to send the input commands via MAVROS console.

5.2 MAVROS commands

In addition to these tools, another useful section of MAVROS package regards its commands, in particular **mavsafety** and **mavsys** commands.
mavsafety allows to manipulate safety parameters on MAVLink devices. One significant example is the command used to arm (or disarm) the drone or the rover:

```
rosrun mavros mavsafety arm
```

This command is essential and shall be used every time that the application requires the robot to move or take any action.
On the other hand, **mavsys** command changes the mode and the rate on MAVLink devices. This is useful when the user has to directly control the autopilot with customized inputs that do not pass through a GCS, like in the case of MAVROS console control.
This command can be used with different arguments depending on which type of firmware has been flashed on the autopilot. For PX4 stack firmware the command is:

```
rosrun mavros mavsys mode -c OFFBOARD
```

For APM stack firmware the command becomes:

```
rosrun mavros mavsys mode -c GUIDED
```

Another crucial aspect of MAVROS procedures is the order with which the commands are invoked. As a matter of fact some commands have prerequisites without which they wont work properly and, in some cases, will return an error and wont be executed.
For instance, if the user needs to change the *mode* parameter with the **mavsys** command, this has to be done before arming the robot, and so before calling the **mavsafety** command.

5.3 MAVLink protocol

MAVLink (Micro Aerial Vehicle Link) is a protocol for communication that uses as messages a stream of bytes that has been encoded and is sent to the autopilot via USB serial or telemetry. The encoding procedure puts the packet into a data structure and sends it via the selected channel in bytes, adding some error correction alongside.
"MAVLink follows a modern hybrid publish-subscribe and point-to-point design pattern: data streams are sent / published as topics while configuration sub-protocols such as the mission protocol or parameter protocol are point-to-point with retransmission."[3].

5.4 Structure of the MAVLink message

Each MAVLink packet has a length of 17 bytes and the following structure:

The software is in charge of verifying that each message is valid by checking the *checksum* and in case of negative response, the message is discarded.
For this reason another important parameter that has to be checked is the baud rate. As a matter of fact, the baud rate value has to be the same for every component of the system utilized for reading or sending messages or commands. Concerning this thesis project, the baud rate utilized is 57600.
The most important elements among the ones showed in figure 5.2 are:

- **System ID**: it's the source message sent from the GCS or the MAVROS console to the autopilot via wireless telemetry or USB port. The presence of this message is regularly checked by the software.
- **Component ID**: it's the identification code of the component that is sending that message within the system.
- **Message ID**: it contains the topic of the message sent.
- **Payload**: it's the actual data sent through the message.

5.5 MAVLink function

The real purpose of MAVLink protocol is to exchange messages between the various elements that work together in the architecture just explained, i.e. the ROS/MAVROS and autopilot environment.
This messages are data bundles that contains a fixed number of bytes (i.e. 17). The autopilot gets the streaming bytes forwards it to the hardware interface (e.g. via UART serial OR Telemetry) and decodes the message in software. The goal of this procedure is to extract the payload contained into the message.
In order to be sure that the messages are being sent to the correct component or system, every time a message is sent the first things to be checked by the code are the **System**

Byte Index	Content	Value	Explanation
0	Packet start sign	v1.0: 0xFE (v0.9: 0x55)	Indicates the start of a new packet.
1	Payload length	0 - 255	Indicates length of the following payload.
2	Packet sequence	0 - 255	Each component counts up his send sequence. Allows to detect packet loss
3	System ID	1 - 255	ID of the SENDING system. Allows to differentiate different MAVs on the same network.
4	Component ID	0 - 255	ID of the SENDING component. Allows to differentiate different components of the same system, e.g. the IMU and the autopilot.
5	Message ID	0 - 255	ID of the message - the id defines what the payload "means" and how it should be correctly decoded.
6 to (n+6)	Data	(0 - 255) bytes	Data of the message, depends on the message id.
(n+7) to (n+8)	Checksum (low byte, high byte)	ITU X.25/SAE AS-4 hash, **excluding packet start sign, so bytes 1..(n+6)** Note: The checksum also includes MAVLINK_CRC_EXTRA (Number computed from message fields. Protects the packet from decoding a different version of the same packet but with different variables).	

Figure 5.2: MAVLink message structure

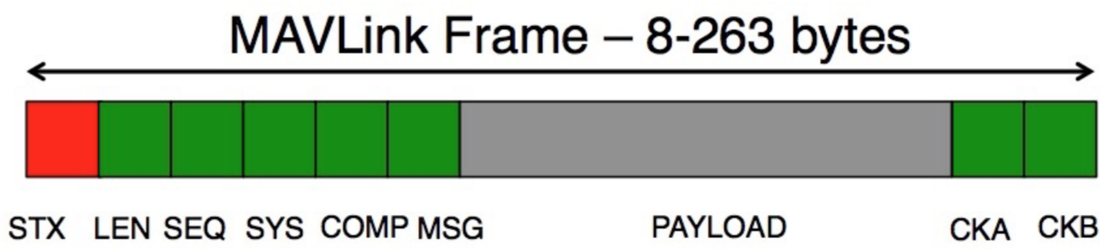


Figure 5.3: MAVLink bytes composition

ID and the **Component ID**. These two IDs are usually hardcoded to be the same.
The payload data is extracted from the message and put into a packet that is then placed into an appropriate data structure. These data structures define the different parameters of the robot, e.g. attitude (pitch, roll, yaw orientation), GPS, RC channels, etc.
Another feature of MAVLink messages is that they are bi-directional. In particular, they can be exchanged from the Ground Station Control to the APM/PX4 autopilot or vice versa.

5.6 MAVLink messages from GCS to autopilot

The main messages exchanged between GCS and the autopilot have the *MAVLINK_MSG_ID_* type and then, in addition, a sub-message type that changes according to the category of the message taken into consideration. A list of the main messages is reported below:

- **MAVLINK_MSG_ID_HEARTBEAT**: it is the most important message. The GCS keeps sending this message to the autopilot with a frequency of 1 Hz to check whether it is connected to it or not. This is crucial to make sure the GCS is synchronized with the PX4/APM stack firmware when the parameters are updated. In the event that a certain number of heartbeats are missed, a failsafe it is triggered and the current mission is aborted. The failsafe option is one of the parameters that can be enabled or disabled using the GCS options section.
- **MAVLINK_MSG_ID_REQUEST_DATA_STREAM**: it is used to request data from sensors, RC channels, GPS position, etc.
- **MAVLINK_MSG_ID_COMMAND_LONG**: it manages loiter mode, RTL (Return To Launch), landing procedure, mission start, arm and disarm.
- **SET_MODE**: used to change mode for the current application. Among the most used modes there are:
  - **ACRO**: holds attitude, no self-level
  - **AUTO**: holds altitude and self-levels the roll and pitch
  - **GUIDED**: navigates to precise coordinates in space taking the inputs from GCS or companion computer
  - **LOITER**: holds altitude and position, uses GPS for movements
  - **RTL**: returns above take-off location, may also include landing

  If a PX4 stack firmware is used, the OFFBOARD mode is also present and shall be used whenever a MAVROS console control is implemented. An example of code useful to enable this mode is reported in Appendix C.
  The GUIDED mode is the equivalent of the OFFBOARD mode, but shall be used along with a APM/ArduPilot stack on the autopilot.

- **MAVLINK_MSG_ID_MISSION_REQUEST_LIST**: requests the overall list of mission items from the system/component.
- **MAVLINK_MSG_ID_MISSION_REQUEST**: requests the information of the mission item with the sequence number indicated in the message.
- **MAVLINK_MSG_ID_MISSION_ACK**: acknowledge message during mission handling.
- **MAVLINK_MSG_ID_MISSION_SET_CURRENT**: this message is used to change active command during a mission. This means that the robot will continue to this mission item on the shortest path.
- **MAVLINK_MSG_ID_MISSION_ITEM**: this message allows to take real-time action like, for instance, setting waypoints and advanced features.
- **MAVLINK_MSG_ID_PARAM_SET**: sets a parameter value temporarily to RAM memory. It will be reset to default on system reboot. The receiving component should acknowledge the new parameter value by sending a param_value message to all communication partners. This will also ensure that multiple GCS all have an up-to-date list of all parameters. If the sending GCS did not receive a PARAM_VALUE message within its timeout time, it should re-send the PARAM_SET message.
- **MAVLINK_MSG_ID_RC_CHANNELS_OVERRIDE**: overrides RC channel values in order to allow the GCS or the MAVROS console to have the full control of the inputs commands needed to control the robot.

5.7 MAVLink messages from autopilot to GCS

In this case the code is structured so that every function has its own running time that wont change throughout the task. This predictability makes these protocol really safe for Real-time systems. The following code is an example of communication from autopilot (APM stack) to GCS :

```
...
static void gcs_data_stream_send(void)
{
  for (uint8_t i=0; i<num_gcs; i++) {
    if (gcs[i].initialised) {
    gcs[i].data_stream_send();
    }
  }
}
...
```

This code sends data streams in the given rate range on both Telemetry and USB links. This is just an extract of the *GCS_Mavlink.pde* [4] script that is used to configure the MAVLink environment when an APM stack is used along with a GCS.

Chapter 6

MAVROS velocity control procedure and final results

Once all the test with the RC has been done, the final goal is to obtain the same results in terms of PWM outputs from the autopilot, but without utilizing the RC.
As a matter of fact, in order to use the ROS environment as common ground for robotics applications independently of the kind of UAV or UGV taken into consideration, all the commands and inputs shall be exchanged outside the GCS systems. This is the reason why instead of a RC that requires GCS in order to properly work, for this thesis project a different approach that uses MAVROS has been chosen.
The first step before proceeding inside MAVROS environment, is to set the parameters of the autopilot that regards inputs control to the right values. In particular, in some cases (for instance, when a PX4 stack has been chosen) the RC override parameter should be enabled so that the autopilot can recognize not only the inputs coming from the RC (that in this part of the application is no longer being used) but also the commands coming from the MAVROS console.
This procedure can be accomplished by changing the RC_OVERRIDE and the SYS_COMPANION parameters. The first one allows to override the default RC privilege on other kind of inputs, while with the second one the user can set his computer as source of inputs and commands taking effectively the place of the RC. In this case like in the previous one regarding communication, it is mandatory to chose the right baud rate for the companion computer. In particular the baud rate shall be the same of the telemetry module utilized to communicate with the autopilot.

6.1 Velocity control with MAVROS and MAVLink

After these preliminary steps have been completed, the real control procedure can be started by connecting the Pixhawk to the OpenCR1.0 board of the rover and to the radio transmitter that will be in communication with the companion computer.
At this point the autopilot is not armed and is set on *HOLD* mode that does not allow the control with a MAVROS console. In order to prepare the Pixhawk and have the full control of the inputs the following instructions shall be executed in the order in which they are indicated:

- **roscore**: as explained in Chapter 1, when working in a ROS environment, this command shall be always the first to be executed.
- **roslaunch mavros apm.launch fcu_url:="/dev/ttyUSB0:57600"**: with this command the MAVROS environment is initialized and the Pixhawk begins to exchange heartbeats messages via MAVLink and waits for a mission. Moreover, the desired baud rate for the communication is also spelt out and given to the *fcu_url* parameter of the launch file.
- **rosrun mavros mavsys mode -c GUIDED**: with this instruction the **MODE** parameter is changed to **GUIDED**, allowing to have full control of the PWM outputs of the autopilot that means being able to control the motors of the rover via MAVROS console.
- **rosrun mavros mavsafety arm**: this command is crucial since it arms the vehicle, allowing the velocity control of its motors. If the vehicle is not armed a lot of features will be forbidden to use and some parameters modification wont be available for the changing.

At this point it is possible to write a script to manage the values sent to the **/mavros/setpoint_velocity/cmd_vel** topic that is the topic utilized to write numerical values in a range from -1 to 1 that will be remapped and traduced into PWM values that will be then generated by the autopilot and found as outputs in the main pins of the Pixhawk. For this purpose, the following script has been used:

```
#include <ros/ros.h>
#include <geometry_msgs/TwistStamped.h>

int main(int argc, char *argv[])
{
ros::init(argc, argv, "cmd_vel_fusion");
ros::NodeHandle nh;
ros::Publisher send_velocity_pub = nh.advertise<geometry_msgs::TwistStamped>("/mavros/setpoint_velocity/cmd_vel", 1000);
ros::Rate loop_rate(100);

```

```
11 geometry_msgs::TwistStamped send_velocity_msg;
12
13 double ros_roll=0.0;
14 double ros_pitch=0.0;
15 double ros_yaw=0.0;
16 double ros_throttle=0.0;
17 int count = 1;
18
19 while (ros::ok())
20 {
21   nh.param<double>("ros_roll", ros_roll, 0.0);
22   nh.param<double>("ros_pitch", ros_pitch, 0.0);
23   nh.param<double>("ros_yaw", ros_yaw, 0.0);
24   nh.param<double>("ros_throttle", ros_throttle,0.0);
25
26   send_velocity_msg.header.stamp = ros::Time::now();
27   send_velocity_msg.header.seq = count ;
28   send_velocity_msg.header.frame_id = 1 ;
29
30   send_velocity_msg.twist.linear.x = ros_throttle;
31   send_velocity_msg.twist.linear.y = 0.0;
32   send_velocity_msg.twist.linear.z = 0.0;
33   send_velocity_msg.twist.angular.x = ros_pitch;
34   send_velocity_msg.twist.angular.y = ros_roll;
35   send_velocity_msg.twist.angular.z = ros_yaw;
36
37   send_velocity_pub.publish(send_velocity_msg);
38   ros::spinOnce();
39   count++;
40   loop_rate.sleep();
41 }
42 return 0;
43 }
```

From line 6 to 9 the node handler and the publisher are initialized. In particular, this script publishes messages on **/mavros/setpoint_velocity/cmd_vel** topic with a rate of 100 Hz (ros::Rate loop_rate(100)) .
From line 13 to line 17 the variables used in the code are initialized.
From line 21 to 24 the parameters are stored within the ROS parameter server using the node handler created in line 7. This allows the usage of **rosparam** command with which it is possible to change the values of these parameters from the terminal while the script is running.
From line 26 to 28 there are some info to be printed on the terminal while this script is running.

From line 30 to 35 the velocity variables assume the values passed with the **rosparam** command. In particular, the two variables that are used for this application are **ros_throttle** that manages the linear.x speed, and **ros_yaw** that manages the angular.z velocity.
Finally, from line 37 to 40 the publisher is set and the ros.spin loop is implemented in order to be able to change the values for the inputs at any time.
While this whole procedure is being performed, there are two topics that shall be always kept in check because of the information they provide. The first one is **/mavros/state**, a topic that allow to check if the autopilot is connected, armed and which kind of mode is active for the current application.
The second one, on the other hand, is **/mavros/setpoint_velocity/cmd_vel** itself, the topic used to write the values for the input commands. This control can be done by invoking the **rostopic echo** command, followed by the name of the topic that the user wants to check.
The two terminals used to check **/mavros/state** topic and **/mavros/setpoint_velocity/cmd_vel** topic are reported below in figure 6.1 and in figure 6.2 , respectively:

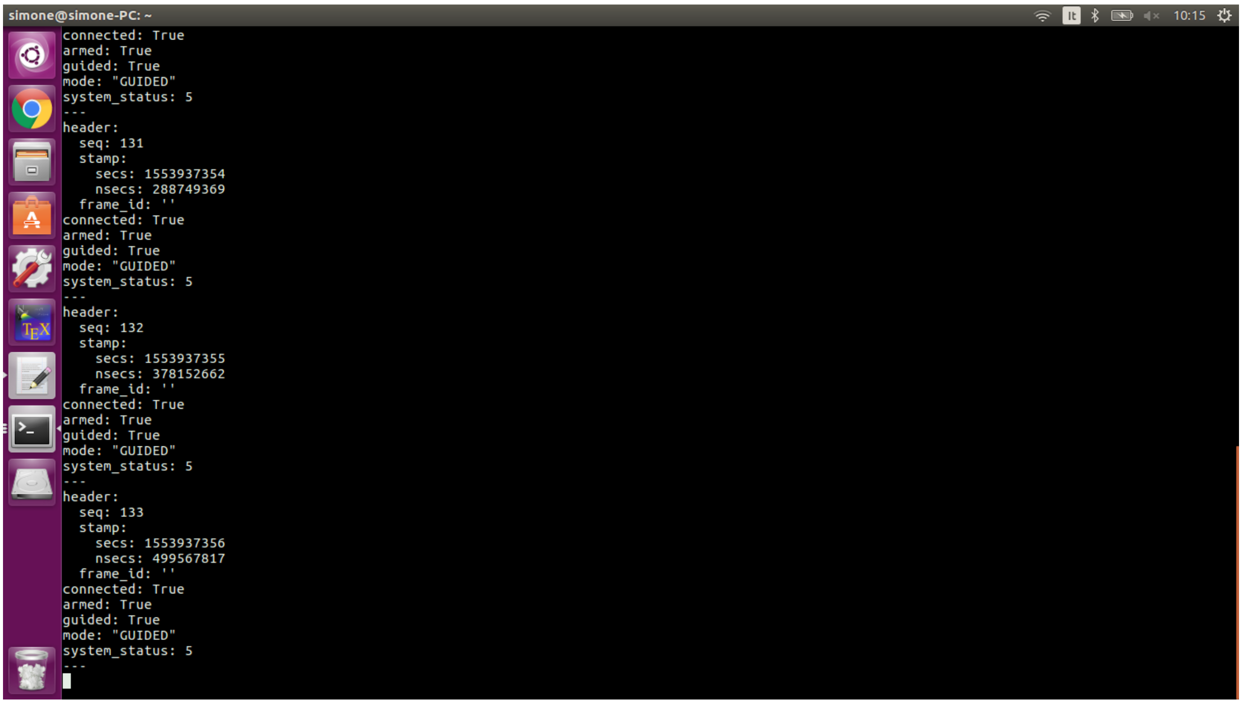


Figure 6.1: mavros/state topic echo terminal

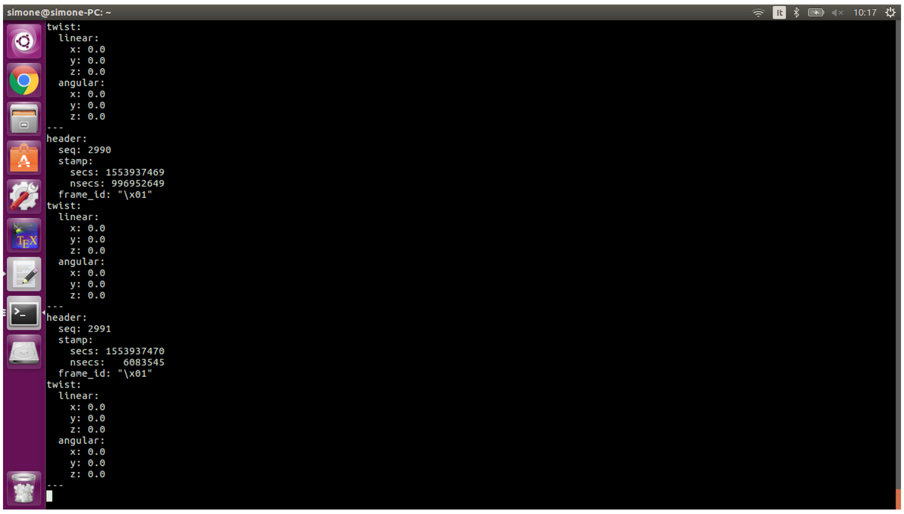


Figure 6.2: mavros/setpoint_velocity/cmd_vel topic echo terminal

As it is shown in figure 6.2, initially the values for linear.x and angular.z velocities are set to zero. In order to change these values, the **rosparam** command has been used with the following syntax:

```
rosparam set ros_throttle 1
rosparam set ros_yaw 1
```

After these commands, the new values for the ros_throttle and ros_yaw parameters have been updated, as it is shown in the echo terminal of the **/mavros/setpoint_velocity/cmd_vel** topic reported below :

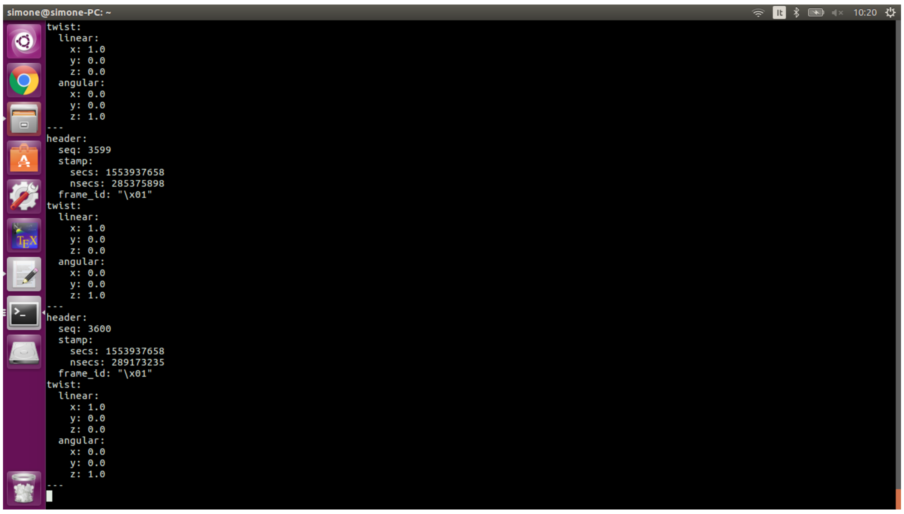
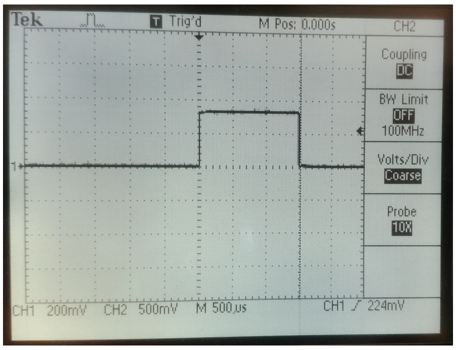


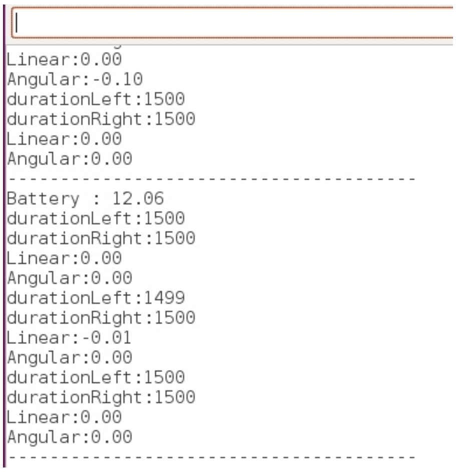
Figure 6.3: Updated throttle and yaw values after the usage of rosparam command

6.2 Final results

The final goal for this application was to make the robot move and gather data following the commands provided through a MAVROS console through the MAVLink communication protocol.
With the following pictures it is showed the achieving of this goal, by means of PWM signals obtained from the autopilot after that the corresponding values were being chosen and provided through the MAVROS console in a ROS environment.
These pictures show the PWM output of the main pins of the Pixhwak (pins 1 and 3 for steering and throttle functions) measured with an oscilloscope. These values are the almost the same reported in Chapter 4 during the RC testing phase and, like in that case, the PWM were eventually transmitted to the OpenCR1.0 board that used them to pilot and control the motors of the rover.
The only difference in this case is the PWM regarding the linear backwards speed control (ros_throttle=-1). As a matter of fact, the *Throttle* parameter that controls the linear speed of the rover did not drop under the trim value of 1500 µs. This may be due to a glitch of the firmware code that has not been optimized for rovers application yet or due to some parameter that does not allow negatives values to be imposed to the *Throttle* function.
In order to fix this problem, a different mapping was performed for this parameter, setting the negative values for the linear velocity in the PWM values range going from 1500 µs to 1750 µs.
As consequence of this mapping, the positive values for the linear forward velocity have been taken from the remaining range, that means from 1750 µs to 2000 µs.
In order to get a stable PWM signal, an important modification to the standard parameters MOT_SLEWRATE needs to be performed. As a matter of fact, this parameter that controls the throttle slew rate as a percentage of total range per second has to be set to zero otherwise the PWM keeps going from the minimum value to the maximum one and makes the speed of the robot not constant in time.
In addition to the oscilloscope readings, each picture comes along with the corresponding Arduino serial monitor output, which reports the numerical values for the PWM and for the traduced linear and angular velocities.

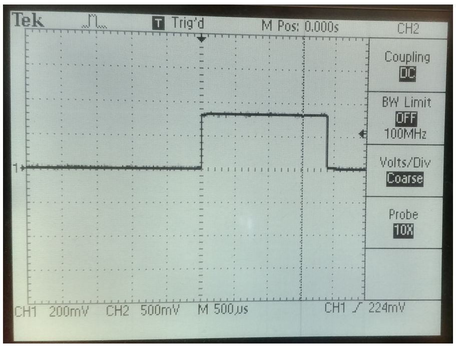


(a) PWM signals for zero linear and angular velocity (ros_throttle=0, ros_yaw=0)

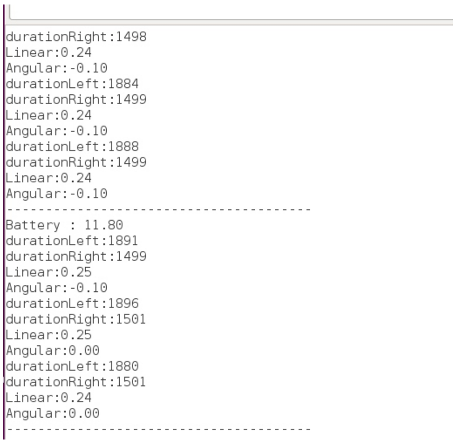


(b) Arduino serial readings for zero linear and angular velocity

Figure 6.4: PWM and velocity values when no speed input is given

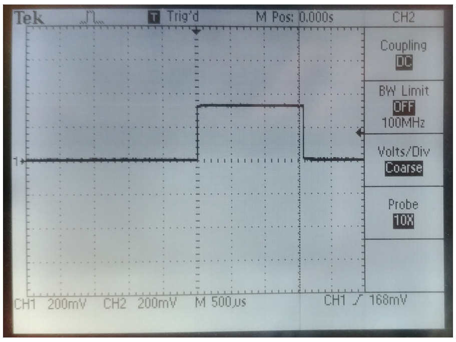


(a) PWM signals for maximum speed forward input (ros_throttle=1)

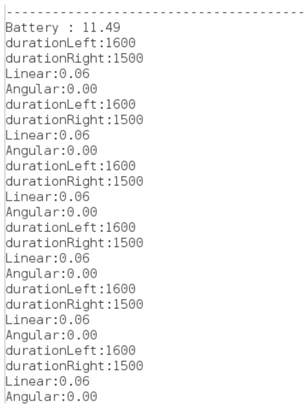


(b) Arduino serial readings for maximum linear forward velocity

Figure 6.5: PWM and velocity values for pure linear forward speed

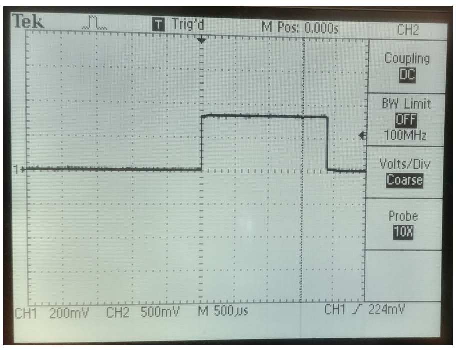


(a) PWM signals for maximum speed backwards input (ros_throttle=-1)

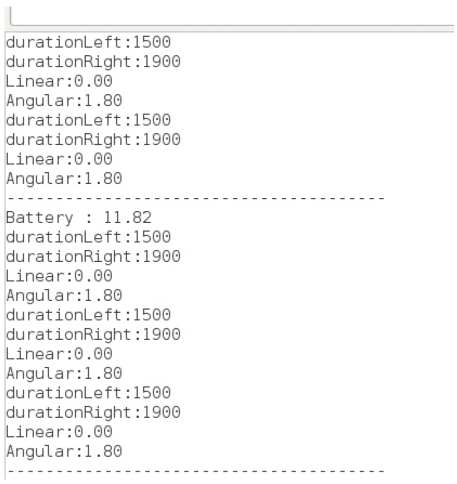


(b) Arduino serial readings for maximum linear backwards velocity

Figure 6.6: PWM and velocity values for pure linear backwards speed

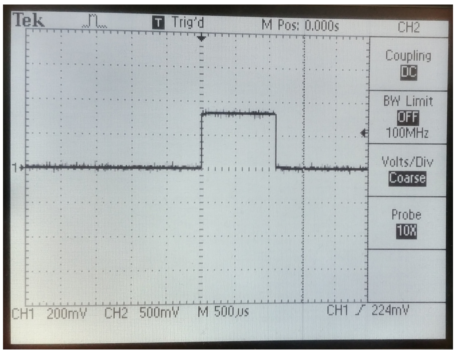


(a) PWM signals for right turn input (ros_yaw=1)

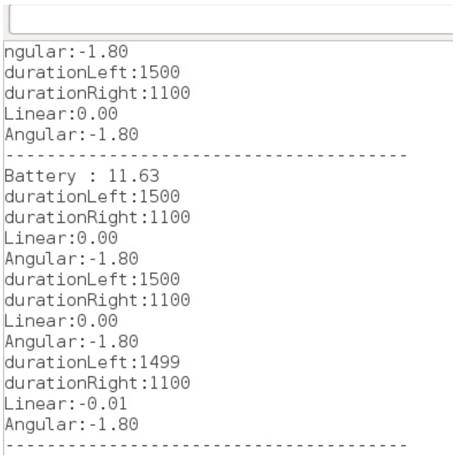


(b) Arduino serial readings for maximum angular velocity (turn right)

Figure 6.7: PWM values and Arduino readings for angular speed (turn right)



(a) PWM signals for left turn input (ros_yaw=-1)



```
ngular:-1.80
durationLeft:1500
durationRight:1100
Linear:0.00
Angular:-1.80
---------------------------------------
Battery : 11.63
durationLeft:1500
durationRight:1100
Linear:0.00
Angular:-1.80
durationLeft:1500
durationRight:1100
Linear:0.00
Angular:-1.80
durationLeft:1499
durationRight:1100
Linear:-0.01
Angular:-1.80
---------------------------------------
```

(b) Arduino serial readings for maximum angular velocity (turn left)

Figure 6.8: PWM values and Arduino readings for angular speed (turn left)

6.3 Conclusion and future work

After some preliminary study regarding the general ROS architecture and intermediary tests with the GCS software, the challenge of this project was to obtain the same results but in a different, more general environment, with a working procedure that would have been easy to extend and reply with different robotic agents and in different scenarios. The understanding and the usage of such powerful tools as MAVLink protocol and MAVROS package have been the most important assets in this thesis, and in the near future will be undoubtedly deepened.
The importance of this project is highlighted if it is placed inside the PIC4SeR context, where a lot of different UGVs are designed and utilized for different applications. Hence, the design and testing of a common database structure is crucial in order to simplify and quicken the upcoming work of those who will have to deal with this kind of procedures and applications.
Concerning the future, the step that would place this thesis project in an even more interesting application scenario would be the design of a GUI that allows the user to set all the mission parameters with simple clicks and without knowing anything about the MAVROS and MAVLink environments.
With this GUI the user selects the type of robot (UAV or UGV), the type of firmware utilized by the autopilot, the type of sensors available for the mission and choose the desired mission parameters.
Furthermore, another interesting development for this project would be the design of a cloud architecture that would be able to treat and redistribute raw data between the various robotic agents that are working on the same application. Moreover, this kind of data sharing procedure can be implemented quite easily inside a working environment like ROS, due to its intrinsic predisposition to share information and data among the components connected to the same network.

Appendix A

follower_controller

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist

x = 0.0
y = 0.0
theta = 0.0

def newTwist(msg) :
    global x
    global y
    global theta

    x = msg.linear.x
    y = msg.linear.y
    theta = msg.angular.z

rospy.init_node("follower_controller")
sub = rospy.Subscriber("driver/cmd_vel", Twist, newTwist)
pub = rospy.Publisher("follower/cmd_vel", Twist, queue_size=1)

speed_waffle = Twist()
r = rospy.Rate(45)
while not rospy.is_shutdown() :
   while (x != 0 or y != 0 or theta != 0) :
      speed_waffle.linear.x = x
      speed_waffle.linear.y = y
      speed_waffle.angular.z = theta
      pub.publish(speed_waffle)
```

```
        r.sleep()

    else:
        speed_waffle.linear.x = 0
        speed_waffle.linear.y = 0
        speed_waffle.angular.z = 0
        pub.publish(speed_waffle)
        r.sleep()
```

Appendix B

driver_controller

```
#! /usr/bin/env python

import rospy
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import Point, Twist

x = 0.0
y = 0.0
theta = 0.0

def newOdom (msg) :

    global x
    global y
    global theta
    x = msg.pose.pose.position.x
    y = msg.pose.pose.position.y
    rot_q = msg.pose.pose.orientation
    (roll, pitch, theta ) = euler_from_quaternion ([rot_q.x,rot_q.y,rot_q.z,rot_q.w])

rospy.init_node("driver_controller")
sub = rospy.Subscriber("driver/odom", Odometry, newOdom)
pub = rospy.Publisher("driver/cmd_vel", Twist, queue_size=1)

speed = Twist()
r = rospy.Rate(4)
while not rospy.is_shutdown() :
   angle_to_goal = 0.4
   while (x < 0.2 and y < 0.2) :
```

```
        if abs (angle_to_goal - theta) > 0.1 :
            speed.linear.x = 0.0
            speed.angular.z = 0.2
        else :
            speed.linear.x = 0.1
            speed.angular.z = 0.0
        pub.publish(speed)
        r.sleep()
speed.linear.x = 0.0
speed.angular.z = 0.0
pub.publish(speed)
```

Appendix C

turtlebot3_core

```
#include <ros/ros.h>
#include <geometry_msgs/PoseStamped.h>
#include <mavros_msgs/CommandBool.h>
#include <mavros_msgs/SetMode.h>
#include <mavros_msgs/State.h>

mavros_msgs::State current_state;
void state_cb(const mavros_msgs::State::ConstPtr& msg){
    current_state = *msg;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "offb_node");
    ros::NodeHandle nh;

    ros::Subscriber state_sub = nh.subscribe<mavros_msgs::State>
            ("mavros/state", 10, state_cb);
    ros::Publisher local_pos_pub = nh.advertise<geometry_msgs::PoseStamped>
            ("mavros/setpoint_position/local", 10);
    ros::ServiceClient arming_client = nh.serviceClient<mavros_msgs::CommandBool>
            ("mavros/cmd/arming");
    ros::ServiceClient set_mode_client = nh.serviceClient<mavros_msgs::SetMode>
            ("mavros/set_mode");

    //the setpoint publishing rate MUST be faster than 2Hz
    ros::Rate rate(20.0);

    // wait for FCU connection
    while(ros::ok() && current_state.connected){
```

```
        ros::spinOnce();
        rate.sleep();
    }

    geometry_msgs::PoseStamped pose;
    pose.pose.position.x = 0;
    pose.pose.position.y = 0;
    pose.pose.position.z = 2;

    //send a few setpoints before starting
    for(int i = 100; ros::ok() && i > 0; --i){
        local_pos_pub.publish(pose);
        ros::spinOnce();
        rate.sleep();
    }

    mavros_msgs::SetMode offb_set_mode;
    offb_set_mode.request.custom_mode = "OFFBOARD";

    mavros_msgs::CommandBool arm_cmd;
    arm_cmd.request.value = true;

    ros::Time last_request = ros::Time::now();

    while(ros::ok()){
        if( current_state.mode != "OFFBOARD" &&
            (ros::Time::now() - last_request > ros::Duration(5.0))){
            if( set_mode_client.call(offb_set_mode) &&
                offb_set_mode.response.mode_sent){
                ROS_INFO("Offboard enabled");
            }
            last_request = ros::Time::now();
        } else {
            if( !current_state.armed &&
                (ros::Time::now() - last_request > ros::Duration(5.0))){
                if( arming_client.call(arm_cmd) &&
                    arm_cmd.response.success){
                    ROS_INFO("Vehicle armed");
                }
                last_request = ros::Time::now();
            }
        }

        local_pos_pub.publish(pose);
```

```
        ros::spinOnce();
        rate.sleep();
    }

    return 0;
}
```

Appendix D

set_velocity

```
#include <ros/ros.h>
#include <geometry_msgs/TwistStamped.h>

int main(int argc, char *argv[])
{
    ros::init(argc, argv, "cmd_vel_fusion");
    ros::NodeHandle nh;
    ros::Publisher send_velocity_pub = nh.advertise<geometry_msgs::TwistStamped>("/mavros/
     setpoint_velocity/cmd_vel", 1000);
    ros::Rate loop_rate(100);

    geometry_msgs::TwistStamped send_velocity_msg;

    double ros_roll=0.0;
    double ros_pitch=0.0;
    double ros_yaw=0.0;
    double ros_throttle=0.0;
    int count = 1;

    while (ros::ok())
    {
        nh.param<double>("ros_roll", ros_roll, 0.0);
        nh.param<double>("ros_pitch", ros_pitch, 0.0);
        nh.param<double>("ros_yaw", ros_yaw, 0.0);
        nh.param<double>("ros_throttle", ros_throttle,0.0);

        send_velocity_msg.header.stamp = ros::Time::now();
        send_velocity_msg.header.seq = count ;
        send_velocity_msg.header.frame_id = 1 ;

```

```
        send_velocity_msg.twist.linear.x = ros_throttle;
        /*send_velocity_msg.twist.linear.y = 0.0;
        send_velocity_msg.twist.linear.z = 0.0;
        send_velocity_msg.twist.angular.x = ros_pitch;
        send_velocity_msg.twist.angular.y = ros_roll;*/
        send_velocity_msg.twist.angular.z = ros_yaw;

        send_velocity_pub.publish(send_velocity_msg);
        ros::spinOnce();
        count++;
        loop_rate.sleep();
    }
    return 0;
}
```

Bibliography

[1] *Clearpath Robotics - Jackal webpage.* URL: https://www.clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/.

[2] *MAVLink console commands.* URL: https://dev.px4.io/en/middleware/modules_command.html.

[3] *MAVLink Developer Guide.* URL: https://mavlink.io/en/.

[4] *MAVLink script for communication with GCS.* URL: https://github.com/GaloisInc/ardupilot-mega/blob/master/APMrover2/GCS_Mavlink.pde.

[5] *MAVROS wiki.* URL: http://wiki.ros.org/mavros.

[6] *Navio2 website.* URL: https://emlid.com/navio/.

[7] *OpenCR1.0 e-Manual.* URL: http://emanual.robotis.com/docs/en/parts/controller/opencr10/.

[8] *Pixhawk webpage.* URL: http://pixhawk.org/.

[9] YoonSeok Pyo et al. *ROS robot programming.* " ROBOTIS Co.", 2015.

[10] *QGroundControl user guide.* URL: https://dev.qgroundcontrol.com/en/.

[11] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System.* " O'Reilly Media, Inc.", 2015.

[12] *RoboEarth multi-agent demonstration.* URL: https://www.youtube.com/watch?list=PL8Jt2DDNgBLo2-XxRStWTlMhwGcwdanfh&v=mgPQevfTWP8.

[13] *RoboEarth website.* URL: http://roboearth.ethz.ch/index.html.

[14] *ROS wiki.* URL: http://wiki.ros.org/.

[15] *Turtlebot3 webpage.* URL: http://www.robotis.us/turtlebot-3/.

[16] *turtlebot3.ino full script.* URL: https://github.com/ROBOTIS-GIT/OpenCR/blob/master/arduino/opencr_arduino/opencr/libraries/turtlebot3/examples/turtlebot3_waffle/turtlebot3_core/turtlebot3_core.ino.