

POLITECNICO DI TORINO

Master's degree course in Mechatronic Engineering

Master's Degree Thesis

END-OF-LINE TEST AUTOMATION USING A LOW COST 6-DOFs ROBOTIC ARM AND ROS



Supervisor:

Prof.re Massimo Violante

Candidate:

Cristiano Poggi

Company Tutor:

Ing. Adolfo Mastroberardino

Academic Year 2019-2020

Glossary


BLE	Bluetooth Low Energy
Bluepy	Bluetooth library for Python
BT	Bluetooth protocol
COTS	Commercial Off-the-Shelf
EOL	End-Of-Line
DOF	Degrees Of Freedom
DUT	Device Under Test
FDM	Fused Deposition Material
GUI	Graphical User Interface
OEM	Original Equipment Manufacturer
OS	Operating System
OTS	Off-The-Shelf
ROS	Robot Operating System
ROS Kinetic	a ROS distribution
STL	3D CAD file format
TCP	Tool Centre Point
URDF	Unified Robot Description Format

Contents

GLOSSARY	2
CONTENTS	3
LIST OF FIGURES	5
1 INTRODUCTION	6
1.1 THESIS PURPOSE.....	6
1.2 WORKFLOW.....	7
1.3 THESIS OUTLINE.....	8
2 INDUSTRY 4.0 STATE OF THE ART	10
2.1 AUTOMATED TEST METHODS.....	10
2.2 OFF-THE-SHELF COMPONENTS.....	12
2.3 OPEN-SOURCE TECHNOLOGIES.....	14
3 ROBOT OPERATING SYSTEM (ROS)	17
3.1 A BRIEF HISTORY OF ROS	17
3.2 WHY ROS: A DISTRIBUTED FRAMEWORK ARCHITECTURE	18
3.3 ROS NODES AND TOPICS: HOW THE DISTRIBUTED PARADIGM WORKS	21
4 NIRYO ONE	25
4.1 OVERVIEW.....	25
4.2 SOFTWARE ARCHITECTURE	29
4.3 SIMULATION MODE.....	32
4.4 NIRYO ONE SETUP.....	35
5 DEVICE UNDER TEST (DUT).....	37
5.1 OVERVIEW.....	37
5.2 GESTURE TECHNOLOGY: THE CONTROLLER MGC3130	39
5.3 BLUETOOTH FUNCTIONALITY: THE RN4678 MODULE	44
5.4 CODE DEVELOPMENT: “DIAGNOSTIC SESSION” IMPLEMENTATION	46

6	TESTER APPLICATION	49
6.1	OVERVIEW.....	49
6.2	CODE DEVELOPMENT AND IMPLEMENTATION.....	52
6.2.1	CREATING THE ROS PACKAGE.....	52
6.2.2	PYTHON SCRIPTS	54
6.2.3	BLUEZ AND BLUEPY LIBRARIES.....	59
6.2.4	THE ROS ACTIONLIB LIBRARY.....	62
6.3	TESTING AND VALIDATION	ERRORE. IL SEGNA LIBRO NON È DEFINITO.
7	CONCLUSION AND FUTURE DEVELOPMENTS	66
7.1	RESULTS	66
7.2	CONCLUSION	68
7.3	FUTURE DEVELOPMENTS	69
	BIBLIOGRAPHY	70
	ANNEX A: DUT MODIFIED SOURCE CODE.....	72
A-I.	THE APP_MAIN.C FILE:	72
A-II.	THE CTL_RN4678.C FILE:.....	73
A-III.	THE MGC_DECODE.C FILE:	77
	ANNEX B: TEST APPLICATION SOURCE CODE	79
B-I.	THE “CMAKELIST.TXT” FILE:.....	79
B-II.	THE “PACKAGE.XML” FILE:.....	81
B-III.	THE “SETUP.PY” SCRIPT:.....	81
B-IV.	THE “DUT_CONFIG.PY” SCRIPT:.....	82
B-V.	THE “DEVICE_BLE.PY” SCRIPT:.....	83
B-VI.	THE “TEST_APP.PY” SCRIPT:	84
B-VII.	THE “TESTER_COMMANDER.PY” SCRIPT:.....	89
B-VIII.	THE “TESTER_CONFIG.PY” SCRIPT:.....	94

List of Figures

FIGURE 2-1: THE ROS POWERED FANUC M710 ON DISPLAY AT AUTOMATICA 2016	16
FIGURE 3-1: BRIEF TIMELINE OF ROS.	17
FIGURE 3-2: GENERAL OS LAYERS ARCHITECTURE.	19
FIGURE 3-3: ROS LAYERS ARCHITECTURE.	20
FIGURE 3-4: ROS COMMUNICATION INSTAURATION PROCESS.	23
FIGURE 4-1: MECHANICAL SPECIFICATION OF THE ROBOTIC ARM.	25
FIGURE 4-2: THE ROTATIONAL ANGLE RANGE OF EACH JOINT ARE REPORTED	27
FIGURE 4-3: THE 6-DOFs OF THE ROBOTIC ARM ARE SHOWN.	27
FIGURE 4-4: A) THE GRIPPER 3D PRINTED AND ASSEMBLED. B) THE GRIPPER MOUNTED ON THE ROBOTIC ARM 	28
FIGURE 4-5: NIRYO ONE ROS STACK PACKAGES ARCHITECTURE.	30
FIGURE 5-1: THE DEVICE UNDER TEST (DUT).....	38
FIGURE 5-2: EQUIPOTENTIAL LINES OF AN UNDISTORTED E-FIELD.	39
FIGURE 5-3: EQUIPOTENTIAL LINES OF A DISTORTED E-FIELD.	40
FIGURE 5-4: ELECTRODE WEIGHTING PROCEDURE.....	42
FIGURE 5-5: E-FIELD LINEARIZATION PROCEDURE.....	43
FIGURE 5-6: RN4678 BLUETOOTH@4.2 DUAL-MODE MODULE.	44
FIGURE 6-1: THE STRUCTURE OF THE ROS PACKAGE "NIRYO_ONE_TEST_APP"	53
FIGURE 6-2: CONCISE LOGICAL FLOWCHART OF "TEST_APP.PY" SCRIPT	57
FIGURE 6-3: THE ROS ACTION PROTOCOL.	62

CHAPTER 1

Introduction

In a European climate of the fourth industrial revolution and with the plan of the Italian government for Industry 4.0, which encourages companies to adapt, this thesis proposes itself as an industrial demonstrator aimed to small and medium enterprises that want to automate end-of-line testing.

1.1 Thesis purpose

This thesis responds primarily to the need to test a device that can interact with the user through simple gestures of the hand; it born from the complications due to the execution of tests by a human operator. For a human being it is realistically impossible to guarantee the reliability, accuracy and repeatability required by procedures such as testing, calibration and configuration of a device equipped with 3D gesture recognition and motion tracking controller chips. Besides, these procedures are characterized by a sequence of simple mechanical actions and are usually repeated many times, both during development and production, making them the ideal candidate for automation. The modularity and dexterity of the arm, together with the interchangeability of the end-effector, make the robotic arm the best choice for this project. Although the price of an industrial robot is still very high, the development of new technologies such as 3D printing and the wide spread of open source technology such as ROS have significantly reduced the costs and time required for the realization of prototype projects. Therefore, today it is possible to find numerous solutions at more affordable prices, mainly aimed at educational and semi-professional market.

After an in-depth analysis, the Niryo One robotic arm, powered by ROS (Robot Operating System), was evaluated as the most suitable for the purpose for its technical specifications, open-source code, and affordable cost. Thus, this thesis wants to ultimately prove the advantages and the capability to perform accurate automated tests even with off the shelf components and open-source software.

1.2 Workflow

The main objective of this thesis is to realize a 3D gestural test bench consisting of the DUT (Device Under Test), the robotic arm Niryo One and a PC for the supervision of the arm connected via wi-fi. Niryo One is controlled by a Raspberry Pi 3 B with Xubuntu for ARM as operating system, where ROS Kinetic is installed and which supports both Bluetooth and Wi-Fi communication. Therefore, since both the device to be tested and the robotic arm supports both Bluetooth Classic and Bluetooth Low Energy (BLE) communication, the latter has been chosen as the communication channel between the arm and the test application developed for the robot.

Thanks to the support of the company Ema s.r.l., the various parts of the arm were printed using a 3D FDM (Fused Deposition Material) printer; then the arm was assembled, wired and configured. So, finally, it was possible to realize this industrial demonstrator.

With this configuration in mind, the test application was designed, developed, implemented and tested. Niryo One offers several programming possibilities but to remain faithful to the purpose it was chosen to use ROS. Hence, the resulting test application consists of several scripts written in Python and enclosed in a ROS package within an architecture that will be presented later. The application is started from the PC

and, briefly, takes care of: connect itself with the DUT via BLE, make the robot perform a sequence of test gestures collecting feedback via Bluetooth, move the DUT according to the test result and provides a report to the PC.

In parallel, to collect the information processed by the DUT, an additional operating mode, named “Diagnostic Session” has been implemented in the device itself, in order to send data about the gestures recognized to the robot via BLE.

Finally, the application was tested and then utilised to perform the developed test on some devices produced by the company Ema.

1.3 Thesis Outline

All the technologies, tools, software and libraries utilised to realise this project are presented and discussed in this thesis.

In chapter 2 the state of the art of industry 4.0 is presented. In particular, among the new technologies promoted, those that have been exploited to realize this industrial demonstrator are introduced.

In chapter 3 a key point of this thesis is brought into, the Robot Operating System (ROS). It is the most widespread open source robotics framework and is also adopted by the “Niryo One” robotic arm that will perform the test.

In chapter 4, the six-axis (six degrees of freedom) anthropomorphic robotic arm “Niryo One” is presented. Then, the robot's software architecture and some of its features are described. The chapter ends with a first preview of the test-bench that will be realised.

In chapter 5, instead, the other main protagonist of this work is described, the device to be tested (DUT). To protect Ema's intellectual property, only those elements / modules that are involved in this work were discussed. After, the changes made to the original source code to implement the new operational mode, to be adopted during the test, are exposed.

In chapter 6, the test application developed for the robotic arm, the heart of this work, is finally produced. The ROS package created is analysed in detail, together with the difficulties encountered and the choices made during the design and development of the application.

In chapter 7, lastly, the results obtained from the test and the conclusions drawn are reported. This thesis concludes with some further work to improve the robot's performance in the execution of gestures and to extend the test to other features of the device under test.

CHAPTER 2

Industry 4.0 State of the Art

Industry 4.0 is an initiative that embraces new technology innovations such as the Internet of Things (IoT), big data, cloud computing, artificial intelligence, machine learning, automation and 3D printing.

2.1 Automated Test Methods

An automatic system can be considered a sum of a decision system (control system) and a decision implementation system (power system). The control system always includes a sensor system that continually detects the effects of decisions made. In the industrial field, the scenario of realisations is hugely vast and varied. Based on conventional mechanical technology, modern industrial automation uses all the other known technologies:

- electrical;
- electronic;
- fluid: hydraulic and pneumatic.

In the specific sector of industrial automation of production processes, the need to operate according to the principles of assembly flexibility and product quality in a highly productive context implies the request for something that goes slightly beyond the concept of robots: the integrated line. To typologically specify the field of interest, consider a simplifying example: that of a station for the automatic composition (assembly) of a mechanical group. Up to now, for assembly operations in industries, the following two solutions were considered:

- Automatic assembly using a particular single-purpose machine
- Manual assembly entrusted to skilled workers

The first solution is a classic example of dedicated automation. An extreme specialisation distinguishes each workstation; only one operation is carried out in sequence, and the motion is controlled by a device specifically built and not convertible. The set up is pushed until the desired performances are obtained, and the cost-effectiveness of the solution is linked to mass production. Thus, for over 20 years, more and more specialised robots with proprietary software have been developed in the industrial sector to perform always the same type of tasks, such as welding and handling of parts.

However, in the manufacturing industry, we are currently witnessing a continuous introduction of automatisms able to replace man in critical functions that they consider their own. According to the human functions replaced, the classification of the corresponding types of automation systems is obtained. Finally, the advent of collaborative robots knocked down the wall between human and machine, opening the way to hybrid forms of automation even in those fields where it seemed impossible to replace human intervention. Today, considering that quality standards must be maintained throughout the production line, each test/control segment is a crucial factor to avoid common mistakes that could be very expensive. In such a dynamic and modern environment, therefore, a robot has to be able to adapt quickly and economically to frequent changes, such as avoiding collisions with nearby machinery or performing pre-programmed tasks even when moved.

2.2 Off-the-Shelf components

In any sector of the industry, the terms Commercial Off-the-Shelf (COTS) and Off-the-Shelf (OTS) components are widely used but mostly in low-volume programs. The idea is that the lower the volume-production, the lower the customised content. So, COTS parts are mainly used to save money, that combined with their immediate availability makes them an ideal solution for low-volume projects. Although COTS is a commonly used term, it often refers to a commonly misunderstood concept as explained in the following. Many programs start with this approach, in order not to pay for the entire realisation of a new customised product, and thus the initial prototypes are designed using OTS components. However, most of them usually abandon COTS solutions before production. This frequently results in higher than expected costs and, sometimes, can be more expensive than a custom design carried out right from the start.

Furthermore, it is also true that there are some pitfalls in designing a platform with COTS parts without doing extensive research on the actual component. While a COTS component may be readily available when the design starts, it is most likely to undergo some revision or change along the way. Although this modification may work well for the program in many cases, there are also some occasions when a slight modification will no longer work for the designed project. This will result in engineering changes not only for that component but potentially for the parts around it as well. Therefore, when designing a new product with COTS components, it is crucial to understand at an early stage, whether it is the right solution.

Another myth misunderstood around COTS parts is their ease of purchase and their ability to plug and play. When starting a project, as already mentioned, the aim is to have as much COTS content as possible. This saves time and total program costs, both essential

elements with which low-volume programs struggle. Nevertheless, after the COTS components are usually purchased from different sources, such as distributors or Tier I suppliers, most of the time, they will not work correctly together due to their design. At this point, in most scenarios, a company will require some kind of technical support from the manufacturer or the distributor, and this is where the situation becomes complex turning into a more complicated process than expected. On the other hand, modifying the design of a COTS part is a significant and expensive task, not as simple as buying a component from such a supplier and having it modified.

Finally, software development and integration are usually necessary because there is always an adjustment somewhere. Despite the various pieces of the platform can still be COTS, bringing a significant saving in total time and cost, the final component is no longer COTS and needs additional specific support.

In conclusion, OTS components can offer an excellent solution in terms of time and money for many platforms, while they are a great starting point for others. The key to reducing the risks surrounding the COTS parts as much as possible is to understand the project's requirements from the outset and then make an in-depth analysis of the component prior to purchase.

2.3 Open-source technologies

A crucial point of this Thesis is the adoption of the Robot Operating System (ROS) to manipulate the robotic arm. ROS is an open-source meta-operating system that provides a set of frameworks for robot development and programming, as explained in detail in the next chapter. This project also includes other software, libraries and tools, always open-source, which will be discussed during the implementation of this demonstrator.

The most common criticism encountered when talking about open-source software is the unreliability, which is an unacceptable risk, especially in the industrial sector where the error tolerance is very low. Nonetheless, this ignores about ten years of success in mission-critical applications with open-source software, as well as the fact that many universities are using it more and more. Even if closed systems can be still suitable for mature markets, that usually consists of massive production with only a single application repeated continuously, that is unlikely to be the direction in which future robots are watching. Today, new markets are always demanding for new robot functionality, however, OEMs are becoming incapable of satisfying them because their development resources are committed to maintaining existing applications.

That is why the Robotic Operating System (ROS) is essential for the future development of robots. ROS, in fact, offers a development community that shares technologies and works to achieve mutual goals, without being burdened by the maintenance of existing activities. Therefore, all interesting peripherals and algorithms can be developed first on ROS, and only later for a specific OEM platform. In this way, manufacturers who will dispose of a robust ROS interface will have the advantage of being first in the market to offer the required new functionality. Consequently, as OEMs, instead of thinking about maintaining all the implemented applications, they should switch to a more stable version

of ROS to work on. At the same time, they would also benefit from all ROS tools that aim to improve ease-of-use, primarily via standardization.

One of the historical problems of closed proprietary systems is, in fact, the development of systems that can easily interact with each other. For this reason, one of the objectives of ROS is precisely to create an interface standard for industrial robot controllers in order to provide interoperability between industrial robot platforms. Therefore, anyone creating a ROS driver must adhere to this standard. An example of this standardization is the simple message protocol that interfaces multiple industrial robot controllers and provides a common interface at the robot operating system level.

Thereby, with a standard non-proprietary structure, on the contrary, developers can solve countless problems simultaneously. Only in this way, the development of robots could reach a pace that would be able to meet the demand for robotic applications.

Even if it has not been used in this work it is worth spending a few words on ROS-industrial, which extends the advanced capabilities of ROS software to industrial relevant hardware and applications. It practically embraces all the points of strengths presented above and already contains ROS stacks and packages developed by some of the world's leading robotics companies such as ABB, Comau, Fanuc, Kuka or Universal Robots.

Figure 2.1 reports an example of industrial robot from Fanuc company that implements ROS.



Figure 2-1: the ROS powered Fanuc M710 on display at Automatica 2016

In conclusion, despite the fact that ROS still has some limitations for its full use in industrial applications, such as those requiring real-time processes at the operating system level or secure and encrypted communications, there are already workarounds and some major companies have not been discouraged and are making their first products with ROS.

CHAPTER 3

Robot Operating System (ROS)

3.1 A Brief History of ROS

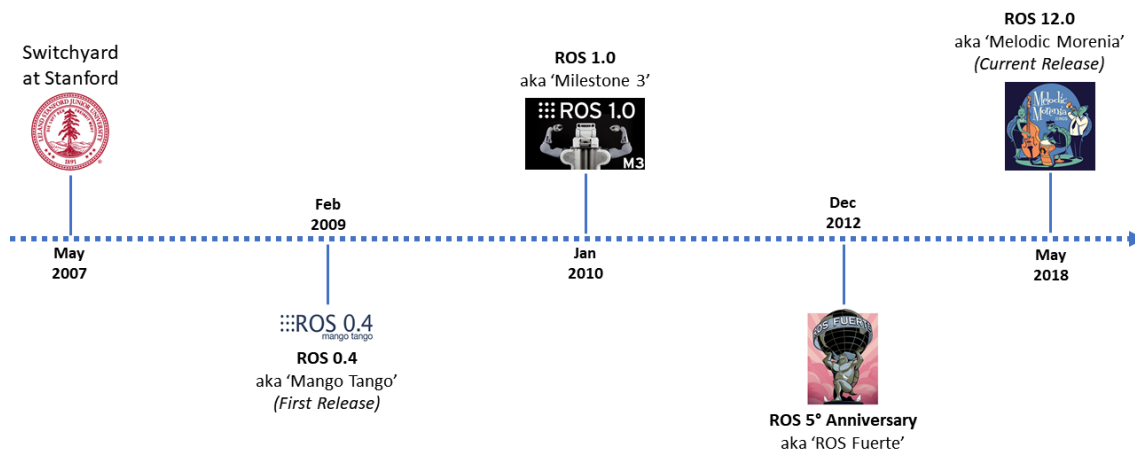


Figure 3-1: brief timeline of ROS.

During the mid-2000s the Stanford University research projects saw several efforts in the construction of software systems for robotics use, such as the STanford AI Robot (STAIR) and Personal Robotic Program (PR) projects, carried out at the Stanford Artificial Intelligence Laboratory (SAIL). The PR project in particular was led by two PhD students, Eric Berger and Keenan Wyrubek, who noticed that the innate diverse nature of robotics, needing expertise ranging from software development to specific hardware knowledge, placed a considerable difficulty to their colleagues. For this reason, they decided to set a out a system that would allow a base ground platform for other developers to build upon.

Beginning this work in a unifier robotic system, the turning point happened when they met Scott Hassan, the founder of a technology incubator named “Willow Garage” who

provided the critical funding needed to build what in his mind was the “Linux for robotics”. From there, the ROS project saw his birth and the first commit was made on SourceForge the seventh of November 2007.

3.2 Why Ros: A Distributed Framework Architecture

As highlighted in the previous section, when developing a software for robots several tasks are involved requiring a certain degree of knowledge in numerous scientific and engineering fields. As an example, motion planning applications require knowledge of coordinate geometry, artificial intelligence, soft computing and programming while robot vision applications require knowledge of signal processing, image processing, computer vision, and machine learning.

This also means that, at a deeper level, many different developing tasks are to be addressed. Logging, message infrastructure, coordinate system transformation, peripheral drivers, navigation logic, vision systems and many other subtasks should all be taken on by the application developer: this generates an immense technical debt that would quite certainly never be filled. This calls out to a layered infrastructure where the application layer is separated and all the other software services and hardware resources can be abstracted and accessed by it. This is the basic concept of an operating system and the fundamental reason to the implementation of the ROS ecosystem.

To make a parallelism, a general operating system would have the following structure:

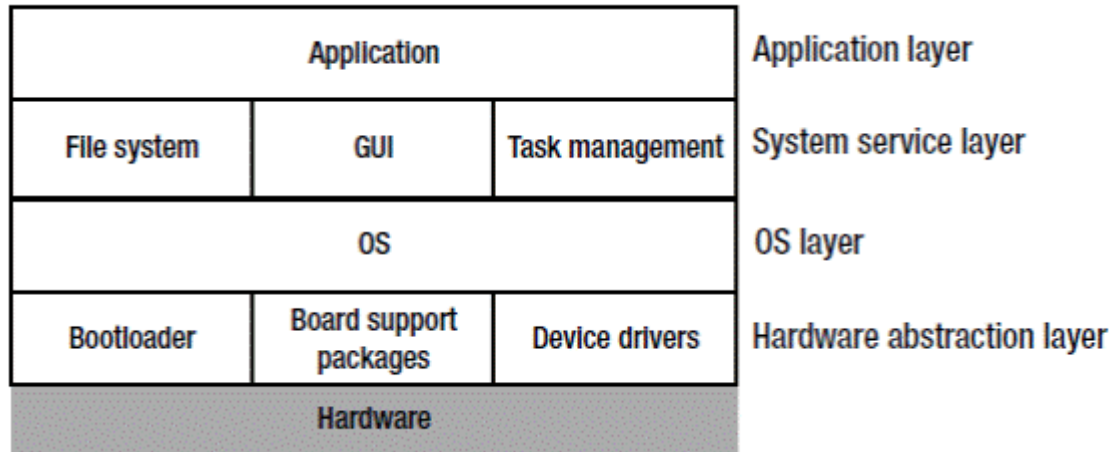


Figure 3-2: general OS layers architecture.

The Application Layer is at the top level of the hierarchy and implements the final functionality of the system. Typically, the application runs at a low-privileged state (i.e. in *user space*) and has access to the system resources through the APIs provided by the operating system which instead runs at a high-privileged state (i.e. in *kernel space*). The System Service Layer provides the top-level interfaces to the application and has the role of linking the application environment to the system environment. Graphical user interfaces (GUIs) and file system hierarchy typically take place inside this layer. The Operating System Layer abstracts all the hardware functions and present them (through the System service layer) to the application. The process scheduler, the network manager and the file synchronization process are typical examples of services abstracted by this layer. Finally, the Hardware abstraction layer (or HAL) prepares and enables the hardware environment before starting the execution of the core component of the operating system i.e. the kernel. As such, the bootloader and all the drivers responsible to the correct communication to the hardware peripherals are common examples of functions provided by the HAL.

In the following picture, the ROS architecture is shown:

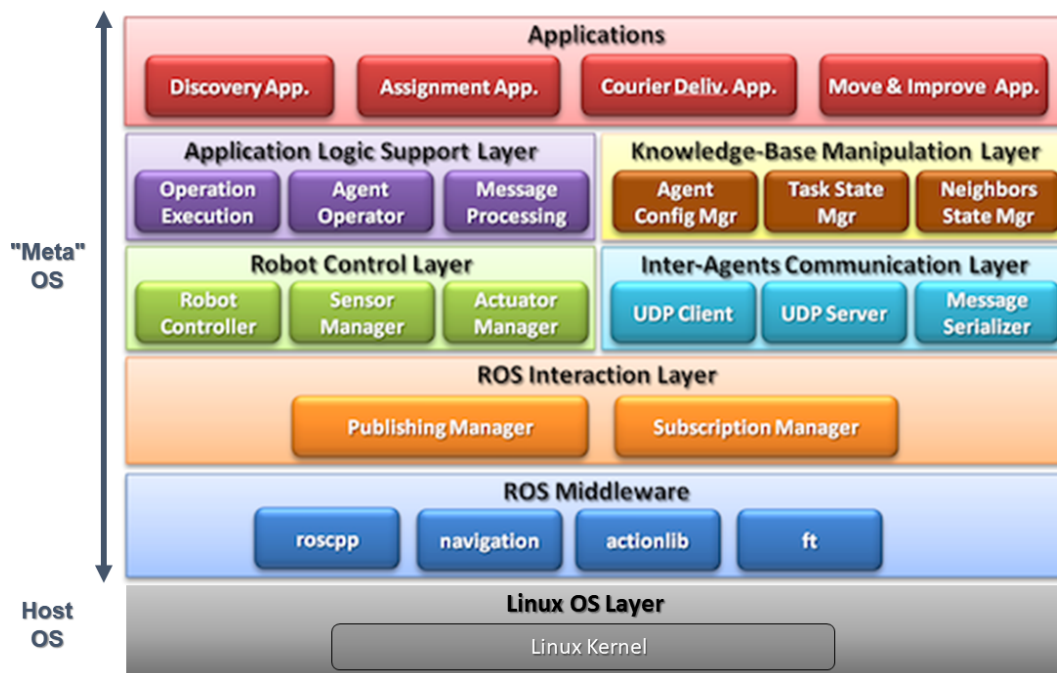


Figure 3-3: ROS layers architecture.

The main difference with respect to a traditional OS layered architecture, is that ROS actually runs on top of a host operating system, most commonly a Linux distribution, i.e. Ubuntu. ROS then is technically not an operating system but a software framework (or middleware) providing an integrated development environment to the robot application. For this reason, it has also been called a *meta operating system*.

Moreover, ROS provides a suite of packages driven by the developer communities that implement a dense set of robot functionalities such as trajectory planning, perception, vision, control and manipulation.

ROS has been thought as a *multi-lingual system*: ROS modules can be written in any language for which a client library exists. While the two most common library languages are C++ and Python, current libraries are also written in LISP, Java, Ruby and MATLAB.

All the software in ROS is arranged in *packages*. Each package can contain one or several nodes and it also exposes a ROS interface. Packages can be built by the application developer but are also independently distributed by the community and hosted on a software repository, providing a development environment with a common set of application functionalities.

3.3 ROS Nodes and Topics: How the Distributed Paradigm Works

ROS derives its core architectural philosophy from that of a micro-kernel operating system. This means that a ROS environment consists of numerous small processes that are all connected to each other and function by continuously exchange messages. As such it is presented as *a peer-to-peer (P2P) architecture*.

The ROS processes are called *nodes*. ROS nodes are written using a ROS client library. roscpp and rospy are the two most used libraries written respectively in C++ and Python. ROS nodes are then singularly compiled and executed providing the required atomic functionality: actuator drivers, sensor drivers, trajectory planner, visor and so on.

Each node can publish or subscribe to a specific *topic*, which are stream of data (i.e. messages) strictly built upon a specific type. As an example, a message carrying a velocity command is defined in the following way:

```
Vector3 linear  
Vector3 angular
```

Where Vector3 is once again a message composed in the following way:

```
float64 x
float64 y
float64 z
```

ROS messages are all built starting from *primitive types* and the framework allows the developer to define their own messages when the built-in ones are not sufficient to fully implement the application function. They are defined inside the directory of a package and are then compiled into the same language implementation chosen for the application.

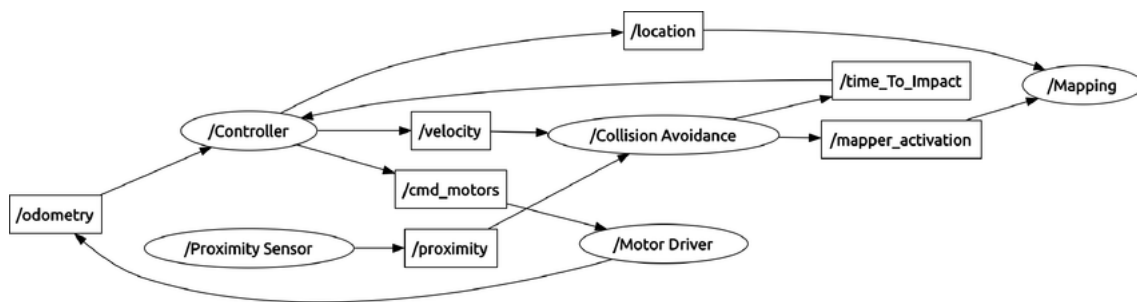


Figure 3-4: an example of ROS Graph.

Node communication is then implemented by publishing messages on topic with a many-to-many broadcasting model. All the communication relationships between the nodes are synthesized in what is called the ROS Graph that can be dynamically generated inside the environment.

While this broadcasting model can implement a flexible application logic it is not appropriate for *asynchronous communication*, which is vital in a distributed-software architecture to provide a client-server interaction. Through services, a one-to-one request-response model can be implemented.

A service is defined by a pair of messages: one is used for the request and one is used for

the response. Nodes can expose a service via a callable interface where another node sends the request message and then wait for the reply message.

The P2P distributed model is directed by an entity called the *ROS Master*. The ROS Master provides the necessary information to the nodes allowing them to transmit and receive messages between each other. Inside the ROS Master runs also another process that is called the *Parameter Server*. It provides a distributed dictionary accessible via an API that is used to share non-binary data, such as the configuration parameters needed to establish the connection.

At startup, each node connects to the master to register the message stream they intend to publish and the message stream to which they choose to subscribe. Every time a node is instantiated, the master sends to it all the necessary information that is needed to create a P2P connection with the other nodes that publish or are subscribed to the same topic. This process is explained in the following picture.

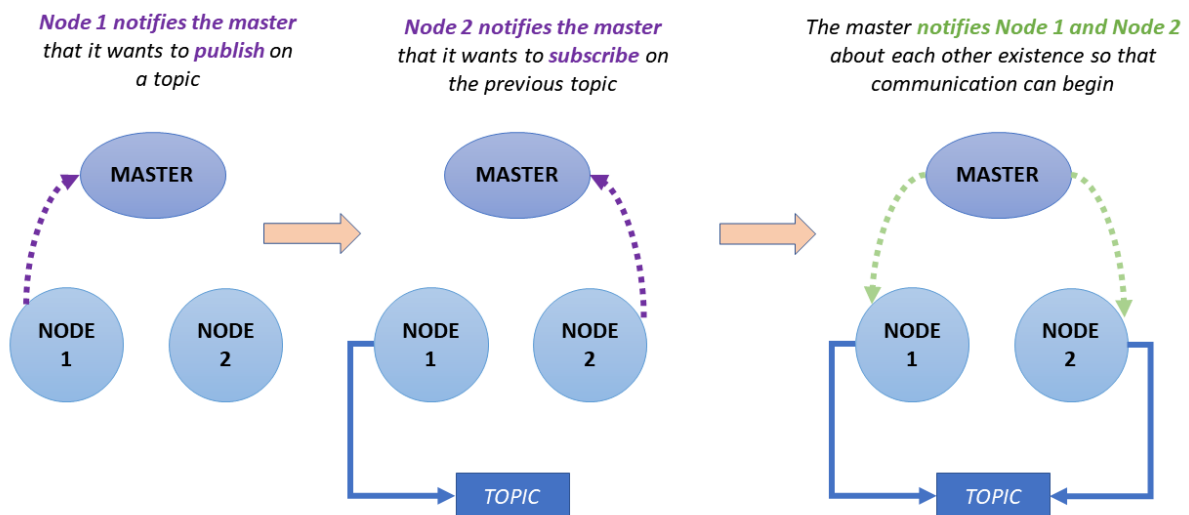


Figure 3-5: ROS communication instauration process.

As shown, before transmitting data over a topic a node must *advertise* the topic name and the message type published on the topic. Only then they can start to *publish* the actual data which other nodes can receive after they have sent to the master the request to *subscribe* upon the topic. All messages on the topic have to be of the same type which has been advertised at the begin of the communication process. The instantiation of multiple nodes and the configuration of the Parameter Server can be done through a *launch file*, which is an XML file defining both a collection of nodes and a list of parameters.

CHAPTER 4

Niryo One

4.1 Overview

Niryo One is a 6-DOFs robotic arm of the French company Niryo. It has several parts 3D printed and is powered by a Raspberry Pi and ROS, which makes it very similar to an industrial robot. Unlike the latter, however, it can be purchased at an affordable price, and it is aimed at small businesses, educational market and makers. In fact, it is designed to allow students to learn the concepts of robotics, through simple UI interfaces, but at the same time also addressed to engineers who want to specialise in this field, using the ROS interfaces directly.

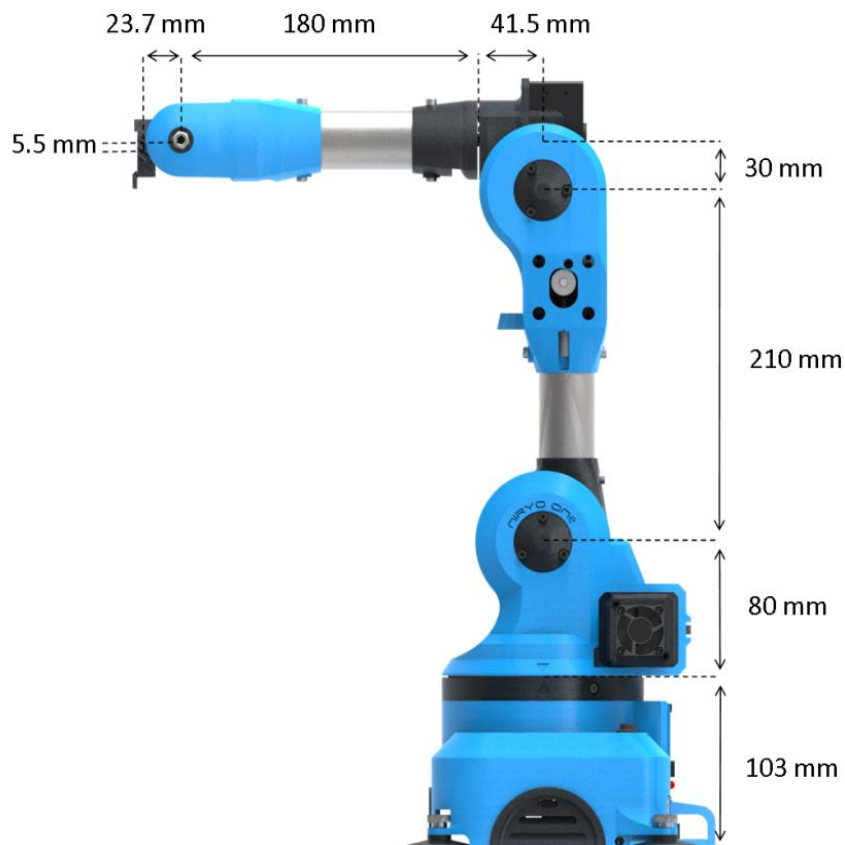


Figure 4-1: mechanical specification of the robotic arm.

Figure 4-1 shows the mechanical dimensions of the robotic arm; it weighs 3.3 kg and can reach up to 440 mm for a max payload of 300 g. STL files of printed parts and the source code of the robot are open-source and can be found on the Github project at [1]. The open-source philosophy is the great advantage of this robot; it allows anyone to make their customizations or improvements benefiting from the support of the entire community. Then these projects will most likely be shared in the community, enriching more and more the functionality of the robot.

The robotic arm is of the anthropomorphic type, i.e. it consists of 6 rotary joints that allow a high level of dexterity and a wide operating range, as can be seen from Figure 4-3. Each joint of the arm is controlled by its motor, which is equipped with an encoder to measure its angular position. Figure 4-3 illustrates all the joints of the arms and their operative range. To provide higher torques to the base (J1), the shoulder (J2) and the elbow (J3) joints, these are driven by the heavier and more powerful NEMA 17 stepper motors, all connected via CAN bus. Instead, in order to decrease the weight of the arm, the forearm (J4), the wrist (J5) and the hand (J6) joints are controlled by Robotics Dynamixel servo motors (XL-430 on J4 and J5; XL-320 on J6) daisy-chained together through a proprietary communication bus and connected to the Raspberry Pi 3.

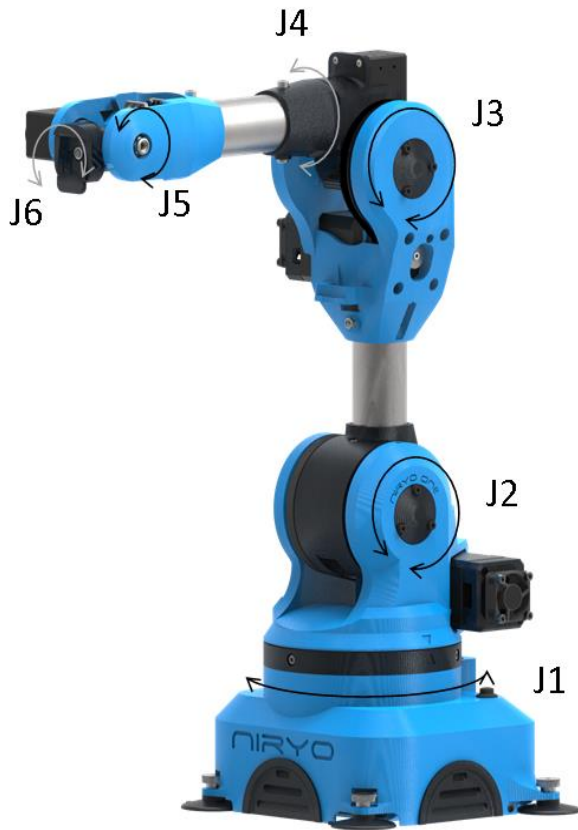
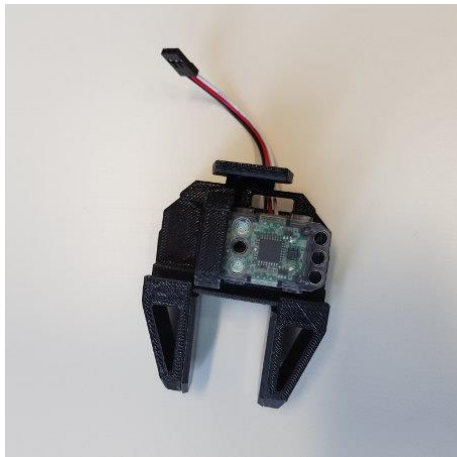


Figure 4-3: the 6-DOFs of the robotic arm are shown.

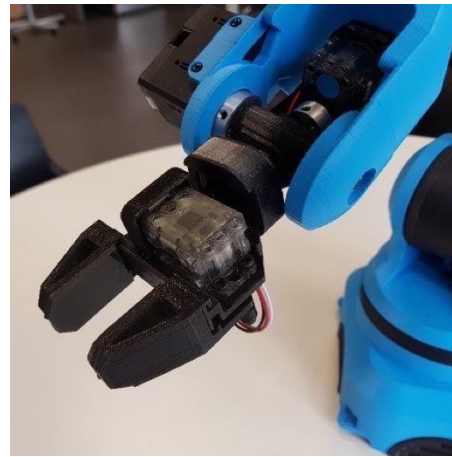
Angle	Min	Max
J1	-175°	175°
J2	-90°	36.7°
J3	-80°	90°
J4	-175°	175°
J5	-100°	110°
J6	-147.5°	147.5°

Figure 4-2: the rotational angle range of each joint are reported

The end-effector chosen for the application, shown in Figure 4-4, is the simplest gripper among those designed by Niryo. It weighs 70 g, is length 80 mm and has a maximum opening width of 27 mm. Besides, it is driven by a Robotics Dynamixel XL-320 servo motors due to its very low weight and the fact that no great forces are required to grasp the artificial hand, being made of a polystyrene parallelepiped covered with a copper sheet connected to ground.



(a)



(b)

Figure 4-4: a) the gripper 3D printed and assembled. b) The gripper mounted on the robotic arm

Niryo One can be controlled in different ways:

- Learning mode: Niryo One is a collaborative robot that can be moved by hand to save the desired positions. In this way, it is possible to teach the robot what to do without knowing anything about robotic concepts.
- Controlled with a joystick: the user can choose between two control modes (position and orientation or independent joints).
- G-Code: the robot can also interpret G-Code instructions and be used as a CNC machine with a proper end effector tool.
- Web and mobile application: thanks to the Niryo One Cloud Service, the robot can be controlled by both Web applications (any browser) and mobile applications (iOS or Android) at the same time, keeping everything synchronized.
- Blockly interface on Niryo One Studio: designed for those without programming knowledge, allows programming the robot in a visual way dealing with blocks.
- Python API: which allows programming the robot in Python by hiding all the complexity of the ROS framework.

- Use the TCP server: very convenient if one wants to send simple ASCII commands to the robot from his application.
- Develop with other languages using "rosbridge_suite": a ROS package contained in the Niryo One ROS stack that allows communication between a non-ROS system and a ROS system, via WebSocket.
- Develop directly with ROS: using the ROS interfaces (topics, services, actions) to create programs for the robot.

According to the purpose of this work, the test application has been developed directly with ROS. So, a new ROS package called "niryo_one_test_app" was developed and implemented on Niryo One ROS stack and is deeply analysed in chapter 6.

4.2 Software Architecture

The Niryo One ROS stack is a collection of ROS packages developed by the company Niryo on ROS kinetic, that provides all the robot functionality. All the files and the documentation about Niryo One ROS Stack is available on the GitHub project at [2].

Errore. L'origine riferimento non è stata trovata. shows the entire ROS Stack packages architecture (Niryo One packages plus used ROS Kinetic packages), from hardware control to the high-level user interface:

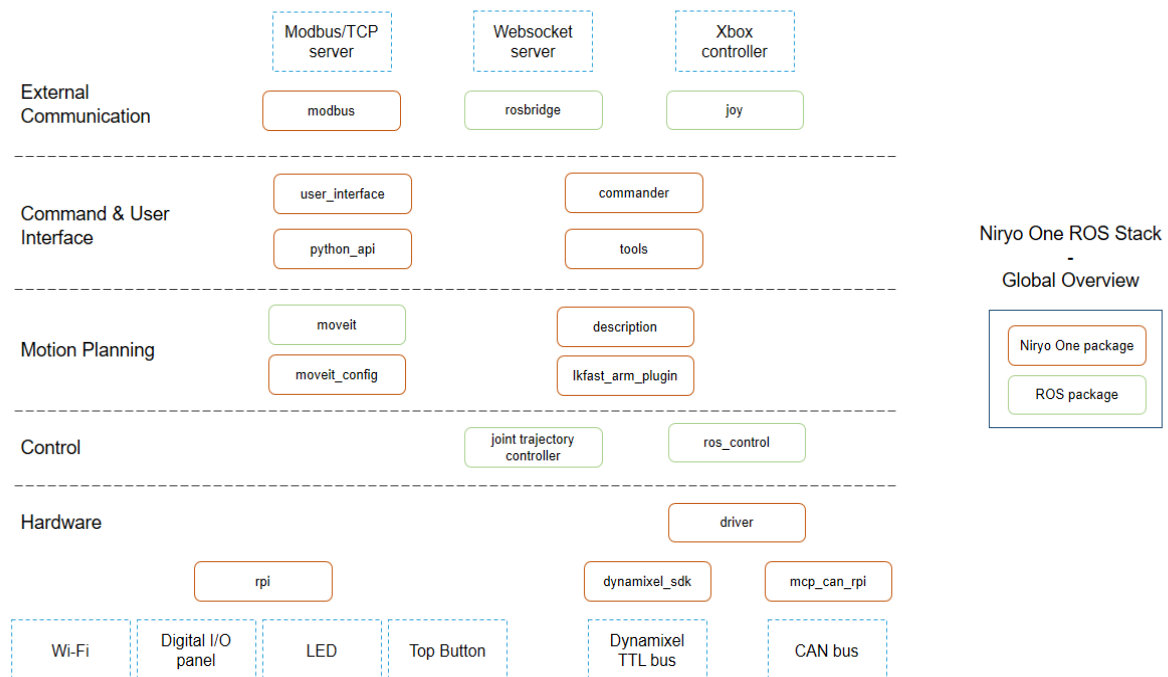


Figure 4-5: Niryo One ROS Stack packages architecture.

A brief explanation of the various layers starting from the bottom:

- **Hardware:** any package that directly deals with hardware belongs to the hardware layer. The "*niryo_one_rpi*" package is responsible mainly for managing the external interfaces, such as the LEDs, the GPIOs from the panel connector, the Wi-Fi connection. The other packages, instead, are responsible for controlling the motors. They consist of a driver that handles both CAN and Dxl bus to allow to the upper level (control layer) to send commands and receive back the position and errors of all the robot motors.
- **Control:** the "*ros_control*" and "*joint_trajectory_controller*" packages together represent the control layer. In particular, the latter implements a position controller which attempts to perform the target trajectory received from the upper level (motion planning). It works by running a control loop that at each cycle: receives the current position from the driver, then interpolates the trajectory to get

the next position command and lastly sends to the driver the newly elaborated command.

- **Motion planning:** this layer takes care of calculating the inverse kinematics in order to create a trajectory for the robotic arm. Here all the work is done by the well-known ROS MoveIt package; while the other packages contain the description of the actual arm's kinematic chain, necessary to configure MoveIt. The resulting trajectory consists of a sequence of points. Furthermore, for each point, the position, speed and acceleration of each arm axis are also defined. Thus, this path will be provided to the controller, which will execute the actual movement of the robot.
- **Command & User Interface:** here we find those packages that act as an interface between the user (a person or another machine) and the core ROS functionalities of the robot. Every command sent to the robot, regardless of how it is controlled, will pass through this layer. Then, the "*commander*" package will take care of validating the command parameters, query the underlying motion planning layer in case movements are required and lastly send the command directly to the control or hardware layer to execute it.
- **External Communication:** this is the top layer of this architecture, and includes those packages that have the task of hiding all the complexity of ROS. It, therefore, provides the ability for users who have no ROS knowledge to control the robot from a non-ROS environment, such as the software Niryo One Studio, other programming languages, web application.

In this thesis work, faithfully for its purposes, the last layer has not been used at all. Instead, the ROS package developed for this test application, "*niryo_one_test_app*", will

be placed just in the command layer where it will directly interact with the package "*commander*".

4.3 Simulation mode

The same Niryo One ROS stack installed in the Raspberry Pi to control the robot was installed on the PC allowing to work in simulation mode during the development phase, i.e. without the physical robot. In fact, another great potentiality of ROS, as explained in the previous chapter, is the possibility to visualise and simulate the robot using the ROS Rviz packages. To do this a PC with Ubuntu 16.04 operating system is needed, where ROS kinetic and Niryo One ROS Stack are already installed. Besides, in the Niryo One project on GitHub, the company Niryo provides a ready-to-use ROS setup of the robotic arm that allows to launch the robot in simulation mode and display it on the PC thanks to the 3D viewer Rviz ROS package. This setup consists of a ROS package named "*niryo_one_description*" which includes the URDF (Unified Robot Description Format) files and 3D meshes (Collada and STL files) needed to Rviz in order to visualise the robot. Figure [XXX] shows a picture of the robotic arm model displayed by Rviz. In order to correctly calculate the 3d pose of the arm and detect potential collisions between its parts, the 3D model of the robot must match with the real one. The kinematic arm model is built starting from the URDF specifications, shown in figure [XXX], which describes the arm using only two elements, called links and joints. The *links* are the rigid parts of the robots while a *joint* is the connections between two different links. For each joint, the type (such as revolute, prismatic), speed and operating range are specified. For each link, instead, the size, shape, colour and some dynamic properties such as inertial matrix can be

defined. Then, by applying the respective 3D mesh to each link, it is possible to visualise both the robotic arm and the motion planning on Rviz.

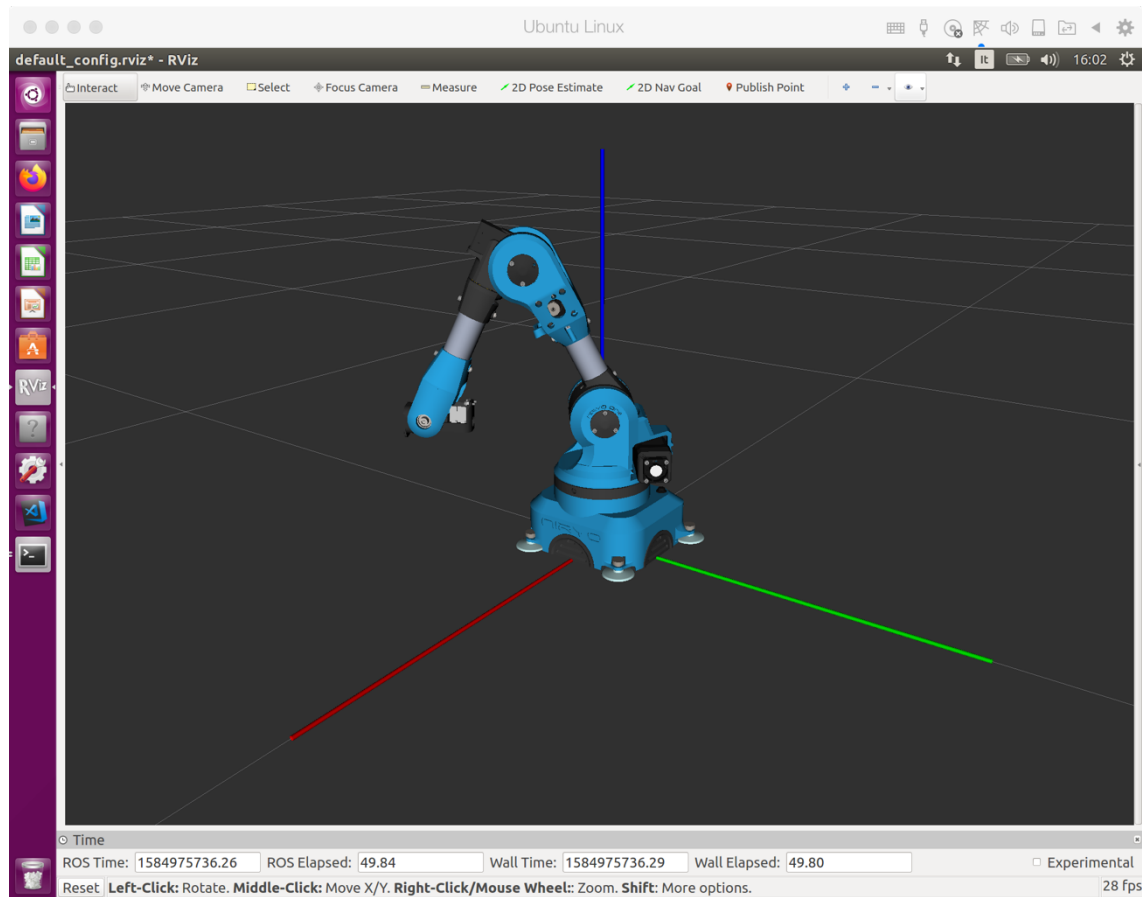


Figure 4-6: the Niryo One robotic arm visualised by Rviz.

Finally, to control the 3D model on Rviz the ROS packages "*joint_state_publisher*" and "*robot_state_publisher*" are used. The former implements a ROS node that reads the robot model description to find all the joints; then allows to set the value of each joint via a GUI slider. The latter implements another ROS node that, instead, reads the current joint states from the "*joint_state*" ROS node and publishes the 3D poses of each robot link as ROS *tf*(transform). These ROS messages contains all the relationships/constraints among the different coordinates frames, one for each link, of the robot.

The main difference between running the code on the PC, i.e. simulation mode, and running it on the Raspberry Pi 3 situated in the arm, i.e. normal mode, is the disablement

of the hardware layer and the launch of the additional Rviz package to visualise the movements of the robotic arm on the PC monitor. Consequently, all commands sent to the hardware layer are redirected to the "*robot_state_publisher*" node and ideally executed by the 3D model on Rviz (Rviz is only a visualiser, i.e. allows even physically impossible arm movements, if not specified otherwise in the robot model description).

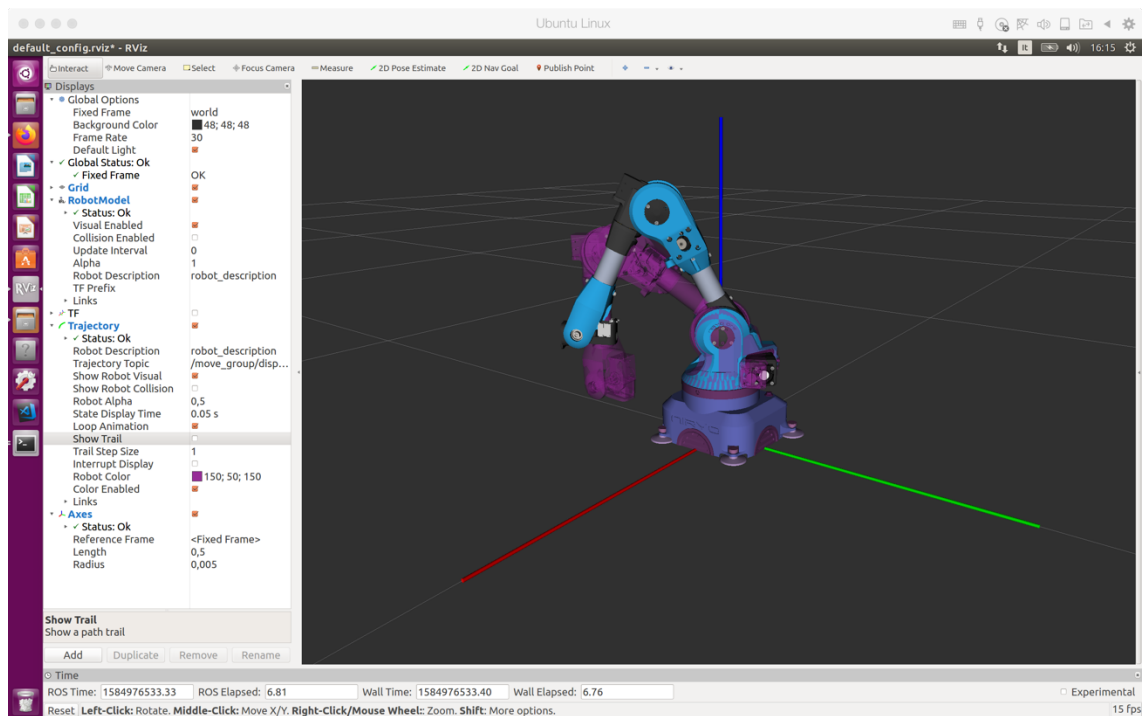


Figure 4-7: a planned trajectory visualised by Rviz

To recap, all the robot features above the hardware layer are the same, allowing developers to work in both simulation and normal mode without noticing any differences. Of course, the features offered by Rviz can also be exploited on the side of the actual robot and not only during the simulation.

4.4 *Niryo One setup*

In the original design of this test bench, there was no camera. The arm, being blind, will have to know in advance the positions of the objects with which it will interact. For this reason, a fixed position layout for the test bench was realised. That means that the robotic arm assumes that the DUT and the artificial hand will always be situated in the same place.

In order to find all positions needed to perform the test, the learning mode of the robot has been exploited. Once the learning mode is activated, the torque is deactivated on all motors, making it possible to move it with the hand like a collaborative robot. Hence all the positions of interest, that are the exact positions of artificial hand and DUT were saved through the Niryo One Studio software on the Raspberry Pi and then exported via ssh connection so that they could be used in the developed application code. On the software, a saved position consists basically of a name and a list of the robot joint positions in radiant. Once exported, as can be seen in the example reported below, it also includes the Cartesian position (*Point*) and the orientation of the TCP (Tool Center Point), expressed both in *RPY* and *Quaternions*.

```
Position_Name: hand_pos
Joints:
0.5062747701481692, -0.6988934985208465, -0.4386456919685022,
0.5479328810228972, 1.1945801121392605, -0.4201297222790069
RPY:
-0.2, 0.0, 0.001
Point:
0.24, 0.125, 0.15
Quaternion:
0.448, 0.547, 0.448, 0.547
```

Unfortunately, due to the serious emergency that we are living in Italy because of the corona virus, it was not possible to show the setup made in Ema company to develop the test application. Instead, figure XX shows the setup made temporarily at home to produce the photos and video of the work done.

FOTO SETUP TEST BENCH

CHAPTER 5

Device Under Test (DUT)

5.1 Overview

The device to be tested, shown in Figure 5-1, is a smart IoT device developed and produced by the company Ema s.r.l. for a client of his own. It was requested to create a led lamp with adjustable intensity and variable colour that would report, in real time, all the parameters collected by its sensors and other smart devices connected to it present in the house. Alarms, lighting management and air quality are just some of the features of this device. The user, in turn, will handle the device without physically touching it and he can access to its features by performing gestures. For example, it is possible to change the lighting intensity of the room, the colour of the light or activate a scenography mode which reproduces all the rainbow colours. It is a part of a complete home system management system that allows to monitor the energy consumption of the home. Thanks to the easy installation of an additional device on the electricity meter of the house, consisting basically of an optical sensor, the colour of the light integrated in the device will vary according to the energy consumption read in real time. Therefore, this will evolve from a green light signal, indicating low consumption, up to a red flashing signal, indicating that you are close to the kilowatt threshold limit.



Figure 5-1: the Device Under Test (DUT)

Its main feature is that it does not have a physical interface, such as buttons or wheels, and it does not even integrate a display: all interactions will be visual and will take place through gestures. This functionality will be explained in detail in 5.2 since the original purpose of this thesis is precisely to automate the testing of this type of interface.

The other functionality that will be used for testing is Bluetooth communication. Being an IoT device, it interacts with other smart devices connected to it through BT, and 5.3 deepens this aspect.

Finally, 5.4 discusses the integrations made to the original code to implement an additional operating mode, named “*Diagnostic Session*”, utilised during the tests.

5.2 Gesture Technology: the controller MGC3130

The device to be tested can receive command inputs with natural hand and finger movements in real-time thanks to the MGC3130, a three-dimensional (3D) gesture recognition and motion tracking controller chip.

This integrated circuit is based on Microchip's GestIC® technology that, applying the principles of electrical near-field sensing, provides all the building blocks needed to develop a robust 3D gesture input sensing system.

All the theory about electrical near-field (E-field) sensing is deeply discussed and explained in [3]. With the support of Figure 5-2 and Figure 5-3, which show the influence of a grounded body on the electric field, a brief explanation of the whole functioning is given.

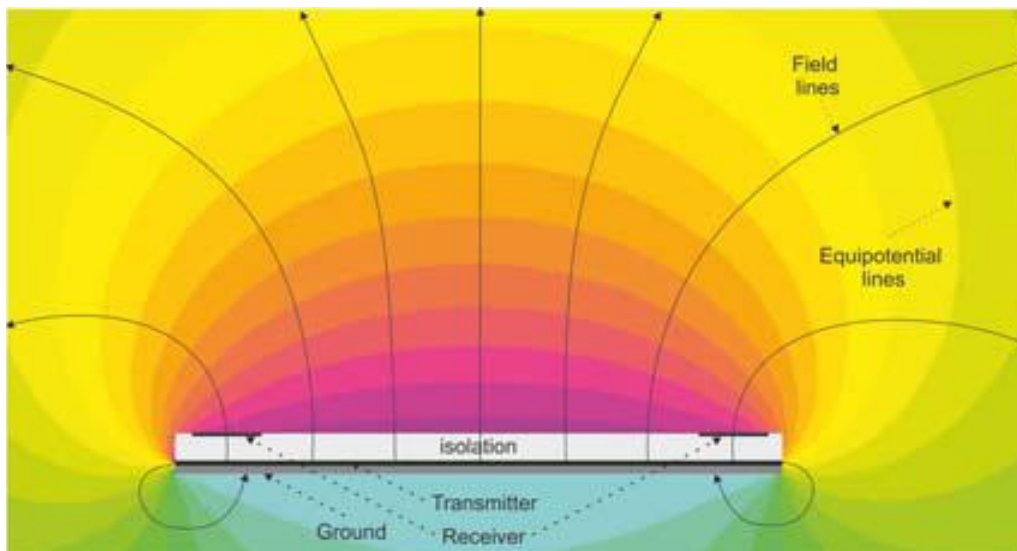


Figure 5-2: equipotential lines of an undistorted E-field.

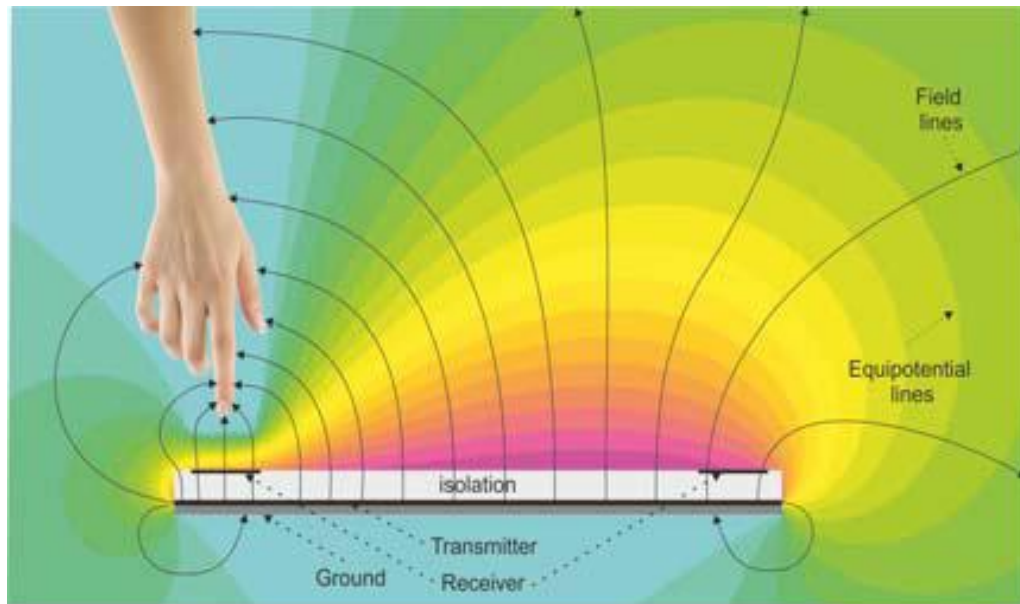


Figure 5-3: equipotential lines of a distorted E-field.

If a person's hand or finger enters the electric field, the field becomes distorted, and the lines of the field will direct toward the hand because of the conductivity of the human body itself, which is grounded. From Figure 5-3, it is possible to observe how the three-dimensional electric field decreases locally due to the proximity of the body and shifts the signal levels of the Rx electrode to a lower potential. Each receiver electrode (Rx) measures the local electric field, and Microchip's GestIC technology requires a minimum of four receivers to detect variations in different positions in order to determine the origin of field distortion from the various signals received. Then the GestIC Colibri Suite, which runs on-chip, processes all this information to calculate position, track movements and detect movement patterns (gestures). The algorithms needed to process information are included in the Colibri Suite that incorporates data acquisition, digital signal processing (DSP) and interpretation. It exposes then two important functional features: the position tracking feature, that provide the absolute three-dimensional position of the hand in real-time, and the gesture recognition model, which detects and classifies hand movement patterns performed within the tracking area.

Moreover, the Colibri Library includes a set of predefined hand gestures, some of which have been exploited by the final application installed on the DUT. Only the latter will be precisely the inputs utilised during the test application developed for the robotic arm. They are described below:

- Flick gestures:

A flick gesture is a one-way gesture in a rapid movement, such as a linear hand movement from south to north inside the detection area. Once determined the position of the cardinal points, according to the orientation of the device the following



flicks were covered during the test:

- Flick East-West
- Flick West-East
- Flick North-South
- Flick South-North.

- Circular gestures:

A circular gesture is a round-shaped hand movement defined only by the direction, i.e. clockwise or counter-clockwise, and not by the starting position of the hand. GestIC technology discriminates between two types of circular movement, of which only one will be used and tested:



- AirWheel: it requires the recognition of circles continuously executed within the sensing area. It provides information about the rotational movement in real-time, such as a counter that increases or decreases according to the direction of rotation applied. The airwheel is very

convenient in various applications such as volume control or, as in this case, light dimming control.

Finally, Microchip also provides a software package that supports the entire design process called the Aurea Graphical User Interface (GUI). In particular, this software has been widely used both during the development of the device and afterwards to display the electrode performance, to fully configure the controller, to record data and to flash the chip. Among the various controller configuration procedures, it is worth mentioning the *Electrode Weighting* and *E-Field Linearization* procedures. These basically consist in repeating some measurements at different heights, using a brick connected to ground, above each electrode and in the centre, as shown below respectively in Figure 5-4 and Figure 5-5. The same brick used in these procedures is the one that has been used in this application that tests the device, the so-called "artificial hand".

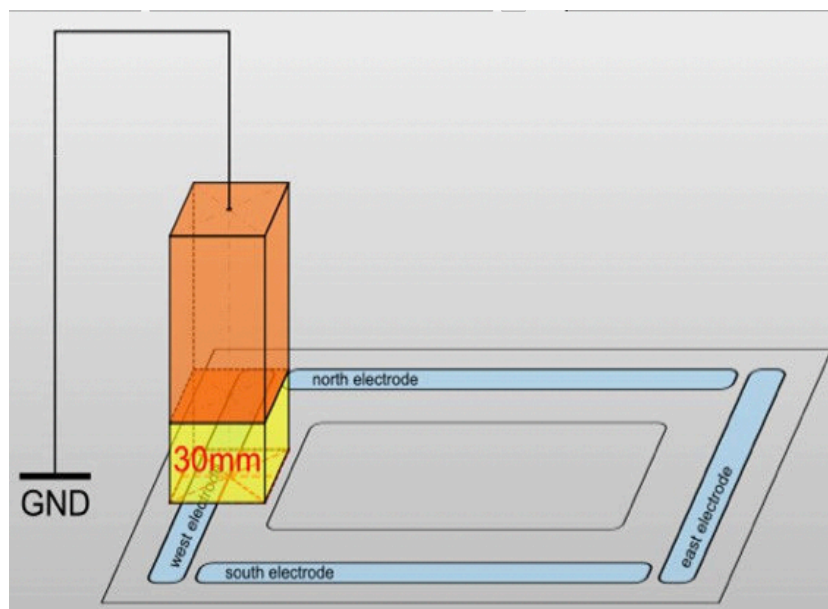


Figure 5-4: Electrode Weighting procedure

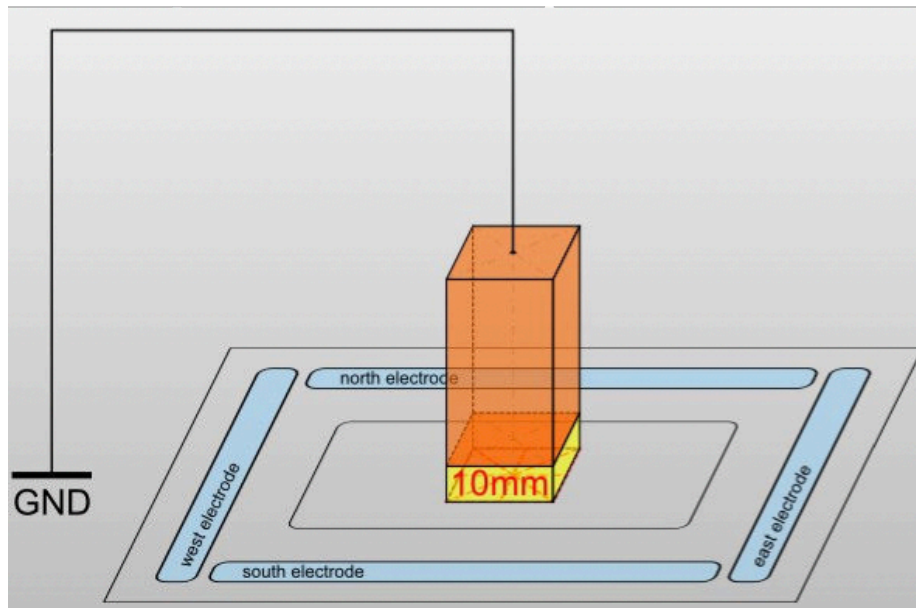


Figure 5-5: E-Field Linearization procedure

Moreover, it has to be said that during the development phase of the device, the company Ema has designed and manufactured several prototypes of boards with different customized arrangements of the four receiving electrodes. It is precisely from there that the need to be able to perform identical and repeatable tests was born, especially to be able to correlate the measurements collected with the different layouts developed

Finally, in the source code developed by Ema s.r.l. for the final product, there is a software module, called "MGC_Driver", which manages and initialize the chip. The module has been modified to implement some functionality needed during the test execution and is analysed in 5.4.

5.3 Bluetooth functionality: the RN4678 module

The device to be tested is also equipped with a Dual-Mode Bluetooth® 4.2 RF module, the Microchip RN4678, that supports both Bluetooth Classic and Bluetooth Low Energy (BLE) communication. It offers a complete wireless solution with built-in Bluetooth stack and integrated antenna, providing the local connectivity for the Internet of Things (IoT). So, the final application on the device can connect it to smartphones and tablets for convenient control or cloud application access and data transfer.



Figure 5-6: RN4678 Bluetooth®4.2 Dual-Mode Module.

For the test, instead, since the DUT is battery powered during testing, only BLE technology has been used due to its lower power consumption.

The RN4678 module has two operating modes [4]: data mode (default) and command mode. In particular, when the RN4678 is in Data mode, it acts as a data pipe. In other words, all data sent to the module's UART is transmitted to the connected BLE device through a custom private service and, conversely, everything received from the connected BLE device via custom private service is sent directly to UART. This BLE data streaming

function that mimics the standard SPP (Serial Port Profile) functionality used under Classic Bluetooth is labelled as UART Transparent. It supports data streaming between two BLE devices and facilitates integration with any microcontroller with a UART interface. When the RN4678 is in Command mode, instead, the module can be easily configured or controlled by sending three types of ASCII commands via UART: Set Commands, Get Commands and Action Commands. In particular, the Set Command allows modifying the configuration by writing the new parameters in the Non-Volatile Memory (NVM). Hence any configuration changes made survive the power cycle but need a reboot of the module to take effect. On the contrary, all the action commands effect immediately, but they will be lost in case of power recycle.

The source code on the device contains a software module, called "CTL_RN4678", which is responsible for initializing and managing the RN4678 integrated circuit. To realize this test bench, therefore, this code has been modified to properly configure the chip and allow the connection via BLE with the robotic arm. It is thoroughly described in the next section.

5.4 Code development: “Diagnostic Session” implementation

As mentioned before, in order to be able to test the different prototype touchless interfaces made, and then the final product ready for the market, it is necessary to collect data. The various prototypal layouts developed by Ema s.r.l. are different only for the geometrical shape/arrangement of the four receiver electrodes on the electronic board. The different samples are shown in Figure 5-7. In this case, therefore, the most crucial information to be able to compare the various layouts are the data obtained by the four sensors and interpreted by the 3D gesture recognition controller. Since the purpose of this thesis is precisely to develop a functional test of the input interface of the final device, such data are exactly the most relevant to our test. In order to easily distinguishing them, each electrode is labelled as to its respective cardinal direction, assuming a determined fixed orientation of the DUT.



Figure 5-7: one of the four-electrode receiver layouts.

The basic idea is to develop an additional operating mode to the device, which can be enabled only during the development and testing phases and offers all the features required by the test. The original device already implemented several operating modes that could be selected with a hand flick. The management of this menu is carried out by the software module "APP_Menu", in which practically the new testing operational mode has been created, called "Diagnostic Session". All the additions/modifications introduced to the C-files that constitute the module "APP_Menu" are available in Annex A-I.

There are two communication channels available to provide feedback during the test: via Bluetooth or the micro-USB port on the device. Since this is an IoT device, and we want to automate the tests minimizing human intervention, a wireless solution was the most suitable. Therefore, communication via BLE was exploited, which offers a considerable reduction in power consumption and ensures better management of the integrated battery. Once this new operational mode has been created, the software module "CTL_RN4678" that manages the Bluetooth module RN4678 was worked. In fact, as soon as we enter the diagnostic session, it is necessary that the device enables Bluetooth and then waits for the robotic arm to be connected. To implement this functionality, a portion of code has been mostly added to the "CTL_RN4678" module. It partly takes advantage of what has already been developed and takes charge of appropriately reconfiguring the hardware module. Recalling the notions about the operating modes of the RN4678 introduced in 5.3; it is possible to configure the chip using simple ASCII commands sent via UART [4]. They are listed below in the same order as they are executed:

- '\$\$\$', is an Action-Command and set the device in Command Mode;
- 'GB', is a Get-Command that returns the Bluetooth MAC address of the device;
- 'SG,1', is a Set-Command to change between Bluetooth modes supported: the number 1 set the chip as Bluetooth Low Energy only;

- ‘**SN,Lamp**’, is another Set-Command that sets the device name to “Lamp” and which will be shown to other BLE devices;
- ‘---’, is the Action-Command complementary to the first one and exits the device from Command mode.

This configuration procedure is performed each time the device enters in Diagnostic Session. Complementarily, it will be seen in the next chapter that the test application, instead, will be responsible for entering the DUT in the diagnostic session and then connecting to it via BLE. Once the connection with the robot was configured correctly, it was possible to take advantage of all the software architecture already developed by Ema s.r.l., in order to easily send the desired data to the Bluetooth module via UART, and finally to the robot via BLE. All the additions/modifications introduced to the C-files that constitute the module “CTL_RN4678” are available in Annex A-II.

Afterwards, in order to retrieve the desired information, work was conducted on the "MGC_Driver" software module, which manages the gesture controller in charge of data acquisition and interpretation. Here the data was obtained directly from the MGC3130 chip, which is actually connected to the four receiver electrodes. The latter is then connected to the micro through the I2C bus and continuously sends a message containing the data processed by GestIC Colibri Suite. Among the data available, the ones of interest for the test are the absolute position in a defined Cartesian reference system (x, y, z), the gesture recognized between those defined in the default set from the library and the counter to track the airwheel movement. Therefore, the driver source code has been modified in order to transmit this feedback to the robotic arm. It is transmitted whenever the device recognizes a gesture among those discussed in 5.2. All the additions/modifications introduced to the C-files that constitute the module “MGC_Driver” are available in Annex A-I.

CHAPTER 6

Tester Application

6.1 Overview

The test application shall have the robotic arm perform a functional test of the device's 3D gesture input system and provide a report on the test performed. The key components of the test are the DUT (Device Under Test), the robotic arm Niryo One that performs the test gestures and the PC that supervises the test. Figure 6.1 shows the improvised test environment due to the state of national emergency:

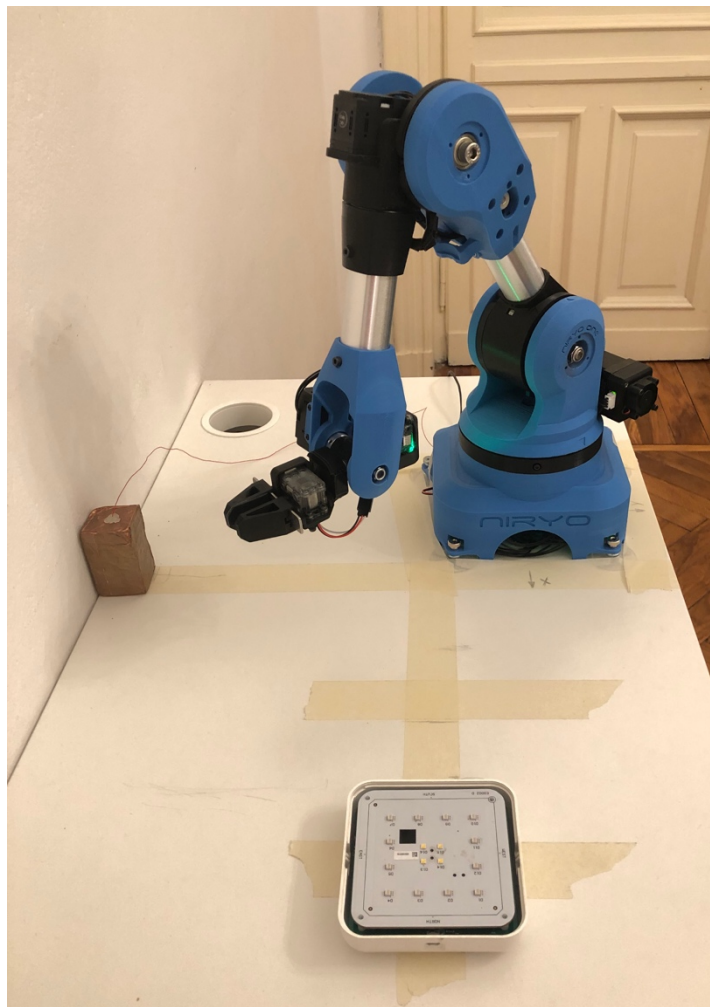


Figure 6-1: the test-bench realised at home

The DUT is examined in chapter 5, where are also described the integrations made to the source code to implement in the device an additional operating mode to be adopted during the test, named "Diagnostic session". Niryo One is presented in chapter 4 as well as the methodology used to develop the application, the Robot Operating System (ROS) introduced in chapter 3. The robot is constantly connected to the PC via Wi-Fi, and it has also to connect to the DUT via Bluetooth Low Energy (BLE) in order to receive feedback during the test. Lastly, the PC must have Ubuntu 16.04 as operating system and installed ROS Kinetic and also Niryo One ROS Stack in order to work in simulation mode or view the robot. For this purpose, a virtual machine with Ubuntu OS was downloaded and installed on a conventional Windows PC that was used to develop the whole project.

ROS is a distributed computing environment, so was chosen to run the test application on the PC, not overloading the Raspberry Pi 3 that controls the robot and taking advantage of the ease-of-development of ROS systems on multiple machines. It essentially consists of a network with a single master that provides bi-directional connectivity between all pairs of machines. To achieve this, all slave nodes must be configured to use the same master and to present themselves with a name that all other machines can resolve. For this work, the robot (i.e. the Raspberry Pi 3 embedded in the robot) is the master, whereas the PC that is connected via Wi-fi has been configured as a slave to 'remote' control the robotic arm.

The test application developed consists of a ROS catkin package, called "*niryo_one_test_app*", which, by recalling the software architecture of the robot shown in Figure Figure 4-5, can be located in the Control layer. Here, it works alongside the "*niryo_one_commander*" package, enriching the robot with specific features used to move the arm during the test. Besides, just like the former, when a motion plan is required

it directly interrogates the "*moveit_commander*" package in the underlying layer that allows the access to actions and services offered by MoveIt, such as trajectory planning.

Once the application is started, it is designed as an FSM (Finite States Machine) to conveniently handle the following sequence of tasks:

1. initialising and preparing the robotic arm;
2. searching and connecting to the DUT via BLE;
3. picking up the artificial hand from its prefixed location;
4. executing of the predefined test gesture and processing the feedback received from the DUT;
5. if the gesture recognised by the DUT is different from the performed one or not detected at all, the robot retries a configurable number of times the same gesture and then moves on to the next one.
6. Repeating steps 4 and 5 for each predefined test gesture, in order:
 - Flick East-West,
 - Flick West-East,
 - Flick North-South,
 - Flick South-North,
 - AirWheel.
7. After testing all the gestures, disconnect from the DUT and place the artificial hand in its spot;
8. providing the test report to the PC and according to it, controlling the robot in order to move the DUT in one between two different direction.

In the next section will be illustrated the main steps involved in the developed of this application as well as the tools, the libraries and the software used.

6.2 Code development and implementation

The decision to make a ROS catkin package was taken because it can be built either as a stand-alone project or within a workspace, where multiple interdependent packages can be built together all at once. A catkin workspace is basically a directory that adopt typical and recommended layout, where packages can be edited, compiled and installed.

6.2.1 Creating the ROS package

After the preparation of the virtual machine with Ubuntu OS and the installation of ROS Kinetic, a catkin workspace was created, and all the Niryo One ROS Stack packages were imported into it. Now, thanks to the catkin tools, it is possible to build all the packages and boot the robot. Before, some ROS environment variables, such as ROS_MASTER_URI and ROS_IP, were configured in order to create a ROS system distributed between the Raspberry Pi 3 and the PC. In fact, the test application has been designed to run on the PC, reducing the workload of the Raspberry Pi. However, it can also run on the robot by delegating only the launch and the report visualisation to the PC.

A package, to be considered a catkin package, must contain: a catkin compliant package.xml file that provides meta-information about the package and a CMakeLists.txt that uses catkin. The package.xml is an XML file, called the package manifest, which must be included in the root directory of the package. It defines all the properties of the package, including mainly the package name and the dependencies on other ROS packages. The CMakeLists.txt file, instead, is the build system input required to build software packages which specifies how to build and where to install the code. The first step in the realization of this application was to create a new ROS package catkin-

compliant provided of a package.xml and a CMakeLists.txt files. At the beginnings, only the package name and some basic dependencies, such as “*rospy*” that allows to program in Python, were defined. Then they have been constantly updated as more packages were used during the application development. Figure 6-2 shows the layout of the ROS catkin package “*niryo_one_test_app*” created for this project, while the final version of each file composing the package is attached in the Annex B.

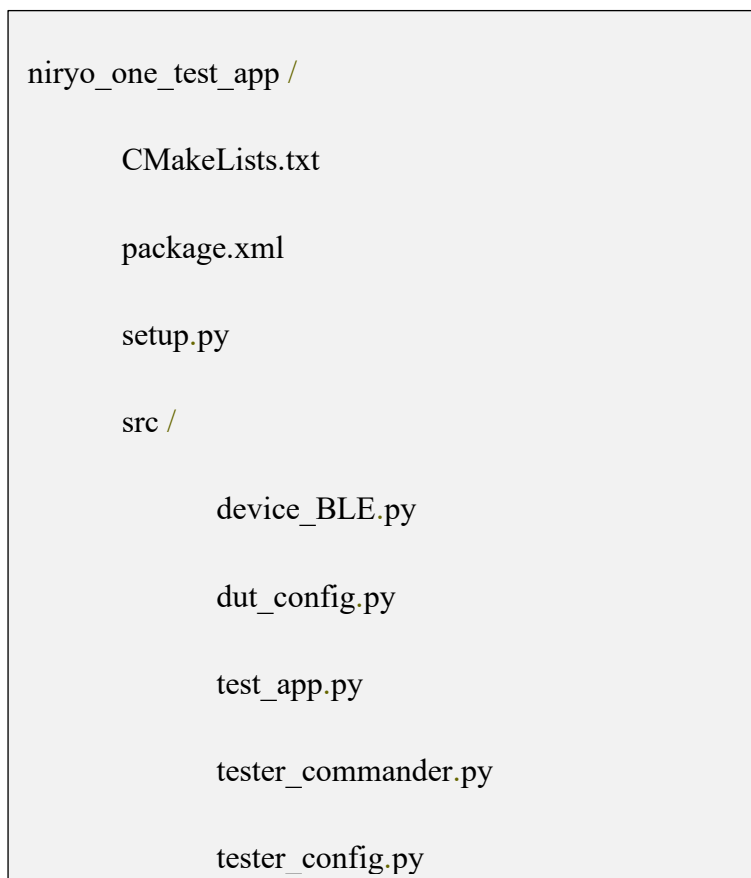


Figure 6-2: the structure of the ROS package "niryo_one_test_app"

6.2.2 *Python scripts*

In the root of the package, according to the catkin workspace layout, then a directory called "src" has been created and will contain all the scripts written in Python that constitute the application. The application code has been organised into several Python files which are listed and briefly outlined below:

- **dut_config.py:** contains customizable parameters about the data transmitted via BLE from the device to be tested and implements the "DUT" Python class which provides some features (through the device_BLE.py script) and information about the device under test;
- **device_BLE.py:** handles everything related to the Bluetooth connection with the DUT, from the creation of a new thread that will be listening to the BLE channel waiting for messages from the device, up to the disconnection of the DUT;
- **tester_config.py:** is a simple configuration file for the robotic arm that contains all the saved fixed 3D poses, discussed in 4.4 and necessary to perform the test;
- **tester_commander.py:** in parallel to using the tools offered by the "niryo_one_commander" package to control the robot, it implements some additional features developed specifically to perform the test gestures;
- **test_app.py:** is the core of the test application since it contains the main program and the finished state machine (FSM) presented before that implements the test procedure. It is the only file to run, with the ROS command *roslaunch* in order to start the test application.

The first two scripts listed are therefore exclusively dedicated to the DUT and are mainly concerned with managing the Bluetooth connection and collecting data from the device. To do this, they include the "Bluepy" and "threading" Python modules that will be further

explored in the next paragraph. They are entirely reported, respectively, "dut_config.py" in Annex B-V and "device_BLE.py" Annex B-IV.

The scripts "tester_config.py" and "tester_commander.py", on the other hand, only concern the manipulation of the robotic arm. The former simply stores the saved positions and orientations and prepares the 3D pose coordinates in the proper format needed to move the robot. It does not provide any functionality but allows quick access in case of adjustments or modification of the test-bench setup. The latter, instead, implements all the new features added for controlling the arm and performing specific test gestures. As mentioned above, it interfaces directly with the "*niryo_one_commander*" and "*moveit_commander*" ROS packages and imports the ROS "*actionlib*" library and Niryo One ROS messages (from "*niryo_one_msgs*" package) to be able to actually manipulate the arm. They both are listed respectively in Annex B-VIII and Annex B-VII.

Finally, the "test_app.py" implements the main program function and the test procedure. It has been represented through the simplified logical block flowchart of Figure 6-3, in which it is possible to distinguish the two parts.

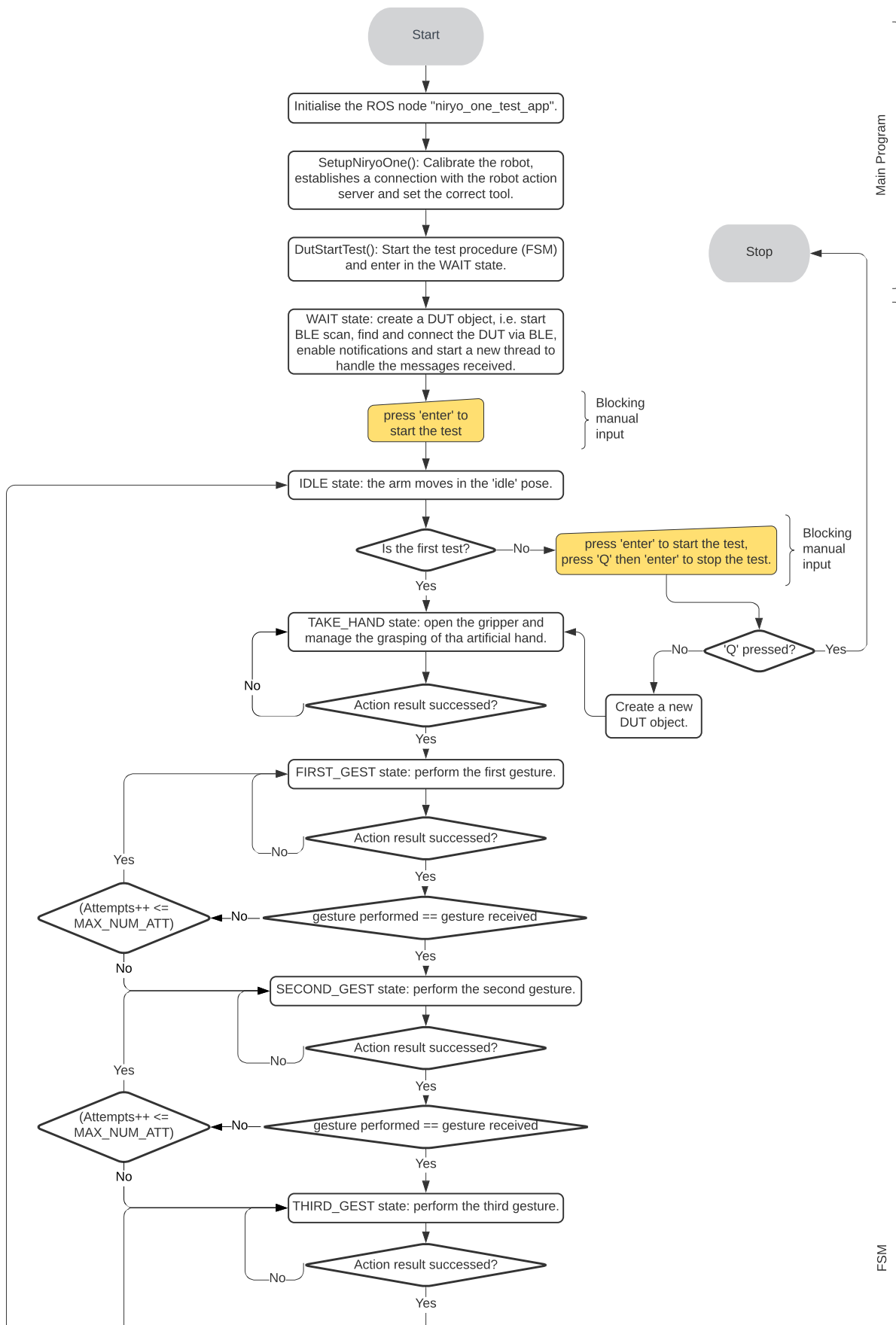




Figure 6-3: concise logical flowchart of "test_app.py" script

The beginning part is the main program that first of all initialises the ROS node named "niryo_one_test_app", and then calls the custom function *SetupNiryoOne()*. This one is defined in the "tester_commander.py", and takes care of launching the autocalibration procedure needed at each reboot of the robot, then establishes a connection with the action server "niryo_one/commander/robot_action" provided by the robot, and lastly set the correct tool through the ROS service "/niryo_one/change_tool". Once the robot is ready, a DUT object is created, which, as detailed in the next paragraph, will connect to the real

device via BLE and collect the messages sent by it. Once the DUT, that should already be in Diagnostic Session, is also connected, the actual test application is started by calling its function *DutStartTest()*. This one implements the FSM machine presented in the overview of the chapter and, after each test sequence execution, block the program offering the choice to run a new test or to close the application. At this point, whether the app is running on PC or Raspberry Pi 3, from the Ubuntu terminal it is necessary to type 'Q' and then 'enter' in case one wants to quit, or just 'enter' if one wants to start another test. The first time we start the test procedure, the DUT is already connected and only the possibility to start a test is offered, allowing the user to prepare and correctly place the DUT in its location as specified in the test-bench configuration file "tester_config.py". In the other cases, a new DUT object is created and then the same sequence of steps is performed. In 6.2.4 it's explained in detail how the arm was manipulated, whereas here follows the description of the logic behind the different operations. Each state of the FSM listed in figure XX, except the WAIT state, consists of a movement of the arm towards a predefined starting position and then a series of actions specific to each state. All starting poses are calculated from the fixed positions of DUT and artificial hand stored in "tester_config.py", where also the parameters used for the calculation can be configured. To reach the first pose and then to perform the following set of specific actions, such as grasping the artificial hand, performing gestures or moving the device at the end of the test, a set of dedicated functions has been developed in "tester_commander.py" to convenient manipulate the arm as required, and they are presented in 6.2.4. Once the robot successfully takes the artificial hand it starts to perform the different test gestures, one at a time, according to the order configured at the beginning of the script. At the end of each gesture execution, the application checks both that all the actions were successfully performed by the arm and that the gesture detected and received via BLE by the DUT matches the one actually performed. If the robot fails to execute a single action, for

example when MoveIt motion planning takes too long to find a valid trajectory, simply retry the same action again. Instead, if the gesture detected by the DUT is different from the expected one, or has not been detected at all, the arm will retest the same gesture only until the number of attempts exceeds the configurable *MAX_NUM_ATT* value and then move on to the next one. Once all gestures are performed, the robotic arm places the artificial hand in its place before moving the DUT according to the test result. This last action has also been thought from the point of view of industrial applications, where the robot can be imagined positioned at the end of a production line, with conveyors, to perform end-of-line tests in order to separate faulty devices from those functioning correctly. Finally, concluding the test, the application disconnects the BLE device and moves the arm into a safe position, the *IDLE* pose, waiting for the user's choice to quit or to run another test. During the entire execution of the script, the application prints continuously on screen, in the terminal window on Ubuntu, the actions that are going to perform, the result of each single action, the data received and those processed, all in real time, using the *loginfo* method of the “rospy” ROS module.

6.2.3 *BlueZ and Bluepy libraries*

When the application runs on the Robot, the integrated Raspberry Pi 3 has built-in support for Bluetooth Low Energy (BLE), the technology that allows even small, low-power devices to transmit and receive information. A simple commercial USB dongle supporting BLE has been used, instead, to run the application on the PC that supports only Classical Bluetooth.

Since the application is developed under ROS frameworks, the source code shall be written in C++ or Python (version 2.7) programming language. Python was chosen from

among the two because of its many advantages. These include, for example, integrated memory management or the no need to compile object files or connect to libraries. The only challenging part to deal with before development starts is making sure of adding Bluetooth support to Python. Therefore, in order to develop all the Bluetooth functionalities requested by the test, the BlueZ and Bluepy libraries has been imported and exploited:

- BlueZ [5] is an open-source project that consists of separate kernel modules, libraries and utilities which together provide everything needed to manage all the Bluetooth core layers and protocols. It implements a Bluetooth stack for Linux-based operating system family and works perfectly with either single or multi-processor and even hyper-threading systems. It has finally become the official Linux Bluetooth protocol stack, and its kernel modules are in the Linux kernel since 2.4.6 release. All the source code is available in the Git repository [6].
- Bluepy, on the other hand, is a Python module that allows controlling BLE devices with Linux-based OS, through a useful API Python interface. The latter, in turn, takes advantage of the API offered by BlueZ module and allows communication with a BLE device via convenient Python commands. This package is an open-source project too, developed mostly on the Raspberry Pi 3, and all the source code together with the API documentation are available on [7].

Hence first BlueZ module and then Bluepy library were installed on both the Raspberry Pi 3 and the PC with Ubuntu 16.04 OS.

Several classes and their instance methods of Bluepy library has been used in the "device_BLE.py" script, in order to define a Class named "*deviceBLE*" which is responsible for:

- scan for the DUT address between the BLE devices available for connection;
- connect to the DUT in order to discover services and characteristics offered by the device;
- create a custom delegated object that processes notifications whenever one is received;
- through UUID retrieve the service and characteristic necessary to enable the reception of notifications from the DUT;
- correctly disconnect the DUT at the end of the test.

Together with Bluepy, the Python *threading* module was used to finally define a class named “*devThread*”, that heirs to the *Thread* class, needed to generate a parallel thread to the main one running the test. This new process is started in the background and is dedicated exclusively to listening to the Bluetooth channel waiting for notifications from the DUT. It creates a *deviceBLE* object instance, and then collects all the data processed by the delegate object in a buffer shared with the main process. In the file "due_config.py" is defined instead the class *DUT* which has several members as name or MAC BLE address and is responsible for generating the new thread that handles the Bluetooth communication. Finally, as seen before, in the file "test_app.py" is effectively created the *DUT* object that will generate the new process responsible for all the BLE features listed above.

6.2.4 The ROS *actionlib* library

In any ROS-based system, it is possible to perform some task by sending requests to a node and then receiving a response, through the ROS service. However, sometimes, if the service takes too long for execution, the user would like to be able to cancel the request during execution or to receive periodic feedback on how the request is proceeding.

For this purpose, the ROS "actionlib" package provides the tools to create servers that accomplish long-run goals with the features just described and also a client interface to send requests to the server. These are named "ActionServer" and "ActionClient" respectively. Therefore, they provide a simple API in order to request goals, on the client side, or execute action, on the server side, by means of callbacks and function calls. Servers and clients communicate using a "ROS Action Protocol", shown in Figure 6-4, that is based on ROS messages and an action specification file.

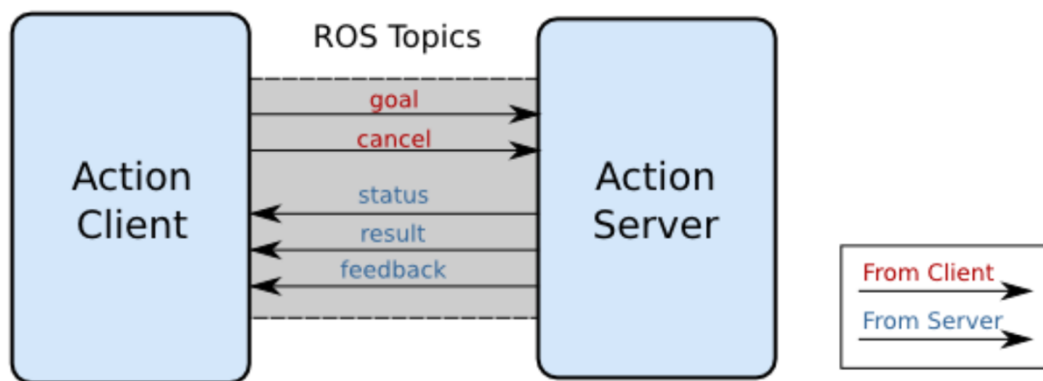


Figure 6-4: the ROS Action Protocol.

The *action specification* consist of an .action file which contains three sections where goals, feedback and result message are defined:

- Goal: a goal is a target that can be sent from an ActionClient to an ActionServer to accomplish a task through actions.

- **Feedback:** the feedback message is used by an ActionServer to communicate to an ActionClient the incremental progress of a goal.
- **Result:** Finally, a result message is sent from the ActionServer to the ActionClient when the task is done, differently from feedback, this is sent only once.

In addition to the messages just defined, as illustrated in Figure 6-4, the action interface protocol includes two other ROS messages:

- **Cancel:** is sent by the client to cancel a request sent to the server.
- **Status:** is sent by the server to notify clients of the current status of each goal in the system.

As example, the action specification “RobotMove.action” implemented by Niryo One is reported below:

```
# goal
niryo_one_msgs/RobotMoveCommand cmd
---
# result
int32 status
string message
---
# feedback
niryo_one_msgs/RobotState state
```

Therefore, in order to manipulate the arm by sending direct commands using ROS, the “niryo_one_commander” package implements and exposes an Action Server named “niryo_one/commander/robot_action”. Alongside the “niryo_one_msgs” package contains the definition of all messages, services and .action files offered by the robot.

Starting from here, the script “tester_commander.py” has been developed in order to implement a SimpleActionClient that will request to the Niryo Action Server all the

movements performed by the robot during the test. Below is the type of goal message, defined in the “RobotMove.action” above, that can be sent to the server to perform a task:

```
int32 cmd_type

float64[] joints
geometry_msgs/Point position
niryo_one_msgs/RPY rpy
niryo_one_msgs/ShiftPose shift
niryo_one_msgs/TrajectoryPlan Trajectory
geometry_msgs/Pose pose_quat
string saved_position_name
int32 saved_trajectory_id

niryo_one_msgs/ToolCommand tool_cmd
```

Then, to send a valid goal message it requires that both the "cmd_type" field, which specifies the type of command to be executed, and the fields relating to the various messages necessary to carry out the requested command are correctly filled. To give a clearer overview of the type of goals that can be requested, the script “command_type.py” that handles the "cmd_type" field and is included inside “niryo_one_commander” package is shown below:

```
1. class CommandType(object):
2.     JOINTS      = 1
3.     POSE        = 2
4.     POSITION     = 3
5.     RPY         = 4
6.     SHIFT_POSE  = 5
7.     TOOL        = 6
8.     EXECUTE_TRAJ = 7
9.     POSE_QUAT   = 8
10.    SAVED_POSITION = 9
11.    SAVED_TRAJECTORY = 10
```


Several of these commands has been used for developing as many functionalities in the “tester_commander.py” file. This Python script will then provide the "test_app.py" program with all the necessary tools to move the arm as desired. So, to execute actions such as perform gestures, take and place the artificial hand or move the DUT at the end of the test, the latter will use the functions exposed by "tester_commander.py", such as *moveArmByPose(pose)*, *moveArmByTrajectory(traj)* or *openGripper()*. All the new features added for controlling the arm that exploited the ROS "*actionlib*" library are reported in Annex B-VII.

Finally, during the development and debugging of the application, some of the logging tools offered by ROS were used and are presented below:

- rqt_console: to display and filter all the messages published in the ROS network;
- rqt_rosbag: for visualizing, inspecting and replaying log files called bags;
- rqt_graph: to display and filter a graph showing the nodes of the ROS network and how they are connected, i.e. they communicate with each other, through topics.

It was no possible to report some picture of the work accomplished, due to the large dimensions of both the ROS Graph picture and the bag files collected.

CHAPTER 7

Conclusion and future developments

7.1 Results

Once the tester application was developed, calibrated and tested, it has been adopted to test the final device produced by Ema. The total time taken by the test to perform the sequence of gesture on a working device has been measured and is approximately equal to 10 seconds. On the contrary, it is not easy to measure the time taken by the human operator to perform the same test. Although at first glance it may seem a quite fast task, only at the practical execution one truly realizes how many times it is necessary to repeat the same gesture to obtain a replicable test. The constant speed of the movement during the gesture, the distance between the device and the hand, or the configuration of the hand itself (the shape that the hand assumes during the gesture) are just a few among the factors on which one must keep focused during the execution of each gesture. In fact, the time taken by the human operator, making no mistakes, to perform the sequence of test gestures is comparable to that used by the robotic arm. However, by analysing the many tests I performed manually before completing this project, succeeding in performing a whole sequence of tests quite similar to the previous one without having to repeat any gestures is something that rarely happens.

For what concerns the accuracy of the tests, the MATLAB software has been used. Since it was used extensively during university courses, the software was already available on the PC as well as the robotic toolbox that implements the ROS framework. The Niryo company, to conclude, also provides a MATLAB interface that connects to Niryo One and allows to analyse the differences between the planned and executed trajectory. It

consists of a Graphical User Interface (GUI) written in MATLAB code, always open-source and available on the GitHub repository at [8], which offers several useful features for debugging and testing the robot's movements and hardware. Figure 7.1 shows the trajectories plotted on the GUI:

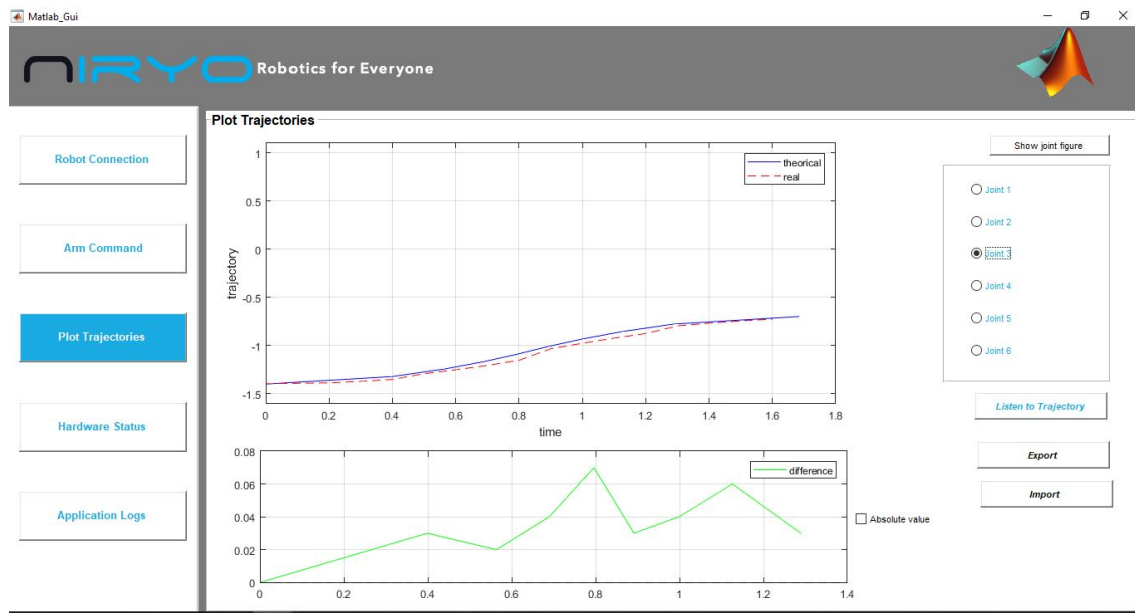


Figure 7-1: planned and executed trajectory plotted on MATLAB.

From the Figure 7.1, it can be noted that the maximum deviation between the objective trajectory and the executed trajectory is about 0.8 mm at most.

7.2 Conclusion

Since the very beginning of modern industrialization, testing and validating products has always been a key process to ensure quality and safety. In Ema's special case study the product line had one operator responsible of programming and testing the whole product. Considering the fact that, under the same conditions, a complete human test execution can take from 10 to 20 seconds, while the robotic arm always takes about 10 seconds, making the test fully automated has led to an effective reduction of work/costs and a saving of production time.

Of course, this project is not suitable for precision industrial applications, but it shows how much can be obtained even from a product targeted to the educational market at a very affordable price.

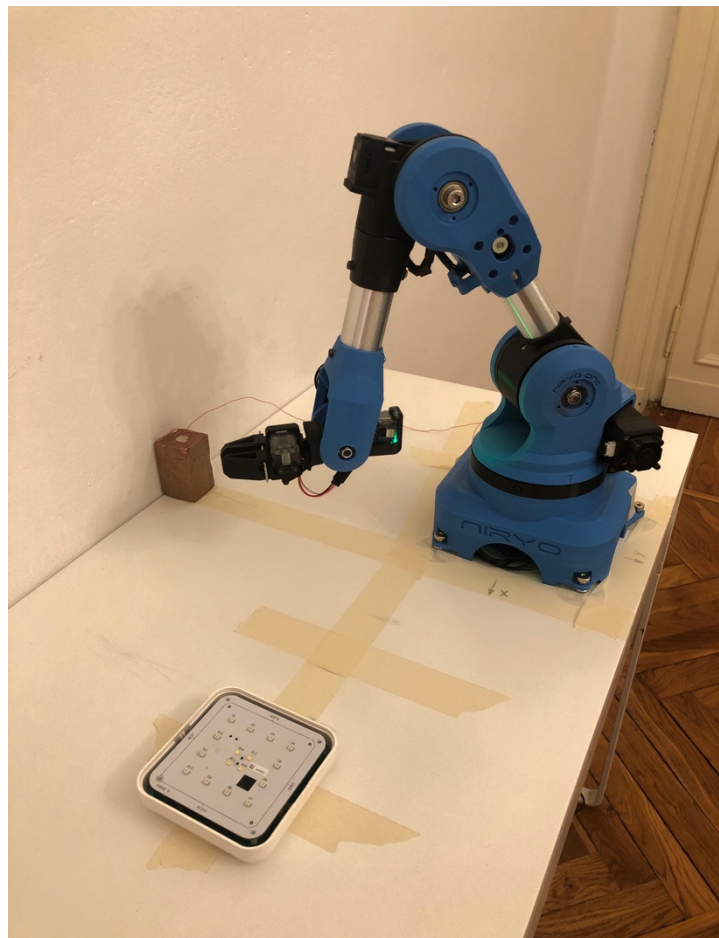


Figure 7-2: the test-bench up and running.

7.3 [9] [10] [11] [12] [13] [14] [15]Future developments

The project realized and presented in this thesis is still work in progress and many improvements and new functionality are just now in the design phase.

Being a lamp that implements a system of interaction based on the use of particular gestures and signals of light and colour, the natural extension of the test proposed so far is precisely to test also the lighting system of the device. The idea is that, while the DUT is in “Diagnostic Session”, in addition to send the Bluetooth message, it will turn on the equipped RGB LEDs with a different colour or pattern, according to the detected gesture. To achieve this, the source code of the device needs some quick modifications to drive the lighting system as desired. Meanwhile, in order to collect the visual feedback from the DUT, the test-bench will be provided of a fixed web camera, whereas the tester application will exploit the popular ROS library OpenCV (Open Source Computer Vision Library) [16]. This package offers several hundred machine vision algorithms that will allows to recognise the colours or the pattern of the light emitted by the DUT.

Another enhancement that will be introduced soon is providing the PC with a graphical interface that will show in a clear and simple way all the information about the test performed relevant for the tester. Several solutions are still under investigation to obtain the best result with the minimum effort.

Finally, new applications for the robotic arm can be developed. For example, as mentioned in 5.2, the calibration needed to optimize the motion tracking controller, that consist of the Electrode Weighting and E-Field Linearization procedures, might be a promising candidate for the next application.

Bibliography

- [1] Niryo, "NiryoRobotics/niryo_one," [Online]. Available: https://github.com/NiryoRobotics/niryo_one. [Accessed 2019].
- [2] Niryo, "NiryoRobotics/niryo_one_ros," [Online]. Available: https://github.com/NiryoRobotics/niryo_one_ros. [Accessed 2019].
- [3] Microchip Technology Inc., *MGC3030/3130 3D Tracking and Gesture Controller Data Sheet*, 2017.
- [4] Microchip Technology Inc., *RN4678 Bluetooth® Dual Mode Module Command Reference User's Guide*, 2016.
- [5] BlueZ Project, "BlueZ," [Online]. Available: <http://www.bluez.org/>.
- [6] BlueZ Project, "Bluetooth protocol stack for Linux," [Online]. Available: <https://git.kernel.org/pub/scm/bluetooth/bluez.git>. [Accessed 2019].
- [7] I. Harvey, "IanHarvey/bluepy," [Online]. Available: <https://github.com/IanHarvey/bluepy>. [Accessed 2019].
- [8] Niryo, "NiryoRobotics/niryo_one_matlab," [Online]. Available: https://github.com/NiryoRobotics/niryo_one_matlab. [Accessed 2019].
- [9] S. Hernandez-Mendez, C. Maldonado-Mendez, A. Marin-Hernandez and H. V. Rios-Figueroa, "Design and Implementation of a Robotic Arm using ROS and MoveIt!," *IEEE*, 2017.
- [10] D. A. Yousuf, M. W. Lehman, r. M. A. Mustafa and D. M. M. Hayder, "Low-Cost Robot Arms for the Robotic Operating System (ROS) and MoveIt," *ASEE*, 2016.

- [11] G. Reinhart, J. Werner and F. Lange, "Robot based system for the automation of flow assembly lines," *WGP*, 2008.
- [12] J. Kruger, T. Lien and A. Verl, "Cooperation of human and machines in assembly lines," *Elsevier*, 2009.
- [13] D.-I. R. Boot, D.-I. J. Richert and D.-I. H. Schutte, "Automated Test of ECUs in a Hardware-in-the-Loop Simulation Environment," *IEEE*, 1999.
- [14] E.-I. Voisan, B. Paulis, R.-E. Precup and F. Dragan, "ROS-Based Robot Navigation and Human Interaction in Indoor Environment," *IEEE*, 2015.
- [15] J. Gmeiner, R. Ramler and J. Haslinger, "Automated Testing in the Continuous Delivery Pipeline: A Case Study of an Online Company," *IEEE*, 2015.
- [16] OpenCV team, "OpenCV," [Online]. Available: <https://opencv.org>. [Accessed 2020].
- [17] M. M. Trivedi, C. Chen and S. B. Marapane, "A Vision System for Robotic Inspection and Manipulation," *IEEE*, 1989.
- [18] Open Robotics, "ROS Wiki," [Online]. Available: <http://wiki.ros.org/>. [Accessed 2019].

Annex A: DUT modified source code

The source code of the device to be tested could not be reported because intellectual property of Ema s.r.l. company. Therefore, only the parts of code written by the author of this thesis with the support of the company tutor are listed below:

A-I. The APP_Main.c file:

```
1.  /*.....
2.  * proprietary code above
3.  *.....*/
4.
5.      /* Macchina a stati applicativo (menu) */
6.      switch (ST_APP)
7.      {
8.          case C_ST_APP_MENU:
9.              /* Modalità d'uso */
10.             switch (APP_LAMP_STR.CURR_MENU)
11.             {
12.                 case C_MODE_CONSUMPTION:
13.                     APP_PowerConsumption();
14.                     break;
15.                 case C_MODE_COURTESY_LIGHT:
16.                     APP_CourtesyLight();
17.                     break;
18.                 case C_MODE_LIGHT_GAME:
19.                     APP_LightGame();
20.                     break;
21.                 case C_MODE_BOREAL:
22.                     APP_BOREAL();
23.                     break;
24.                 case C_MODE_DIAG_SESSION:
25.                     APP_Testing();
26.                     break;
27.
28.                 default:
29.                     break;
30.             }
31.
32.  /*.....
33.  * proprietary code below
34.  *..... */
```


A-II. The CTL_RN4678.c file:

```
1.  /*.....
2.  * proprietary code above
3.  *..... */
4.
5.  case C_RN_ENTER_SETTINGS:
6.      /* comando per entrare in COMMAND session */
7.      if (FLAG_RN.CMD_SENT == 0)
8.      {
9.          EUSART1_Write('$');
10.         EUSART1_Write('$');
11.         EUSART1_Write('$');
12.         FLAG_RN.CMD_SENT = 1;
13.     }
14.     if (eusart1RxCount != 0)
15.     {
16.         command = EUSART1_Read();
17.         if (command == 0x20)
18.         {
19.             FLAG_RN.CMD_SENT = 0;
20.             ST_RN = C_EXEC_COMMAND;
21.             RN_COMMAND = C_RN_COMMAND_GET_MAC;
22.         }
23.     }
24.     break;
25.
26. case C_EXEC_COMMAND:
27.     switch (RN_COMMAND)
28.     {
29.         case C_RN_COMMAND_NOCMD:
30.             /* nessun comando o uscita dalla sessione di comando */
31.             ST_RN = C_RN_ACTIVE;
32.             break;
33.             //
34.         case C_RN_COMMAND_GET_MAC:
35.             /* richiesta MAC */
36.             if (FLAG_RN.CMD_SENT == 0)
37.             {
38.                 rn_bind = 0;
39.                 EUSART1_Write('G');
40.                 EUSART1_Write('B');
41.                 EUSART1_Write('\r');
42.                 FLAG_RN.CMD_SENT = 1;
43.                 //RN_COMMAND = C_RN_COMMAND_NOCMD;
44.             }
45.             if (RN_BUFF[18] == 0x20)
46.             {
47.                 /* retrieve datas */
48.                 for (uint8_t i = 0; i < 12; i++)
49.                 {
50.                     NET_MAC[i] = RN_BUFF[i];
51.                 }
52.
53.                 /* svuoto buffer in ricezione dall'RN */
54.                 for (uint8_t i = 0; i < C_RN_BUFF_DIM; i++)
```

```

55.         {
56.             RN_BUFF[i] = 0;
57.         }
58.         /* uscita dal comando*/
59.         FLAG_RN.MAC_ON = C_TRUE;
60.         FLAG_RN.CMD_SENT = 0;
61.         //RN_COMMAND = C_RN_COMMAND_NOCMD;
62.         RN_COMMAND = C_RN_COMMAND_SET_BT;
63.     }
64.     break;
65.
66. case C_RN_COMMAND_PIN_MODE:
67.     if (FLAG_RN.CMD_SENT == 0)
68.     {
69.         rn_bind = 0;
70.         EUSART1_Write('S');
71.         EUSART1_Write('A');
72.         EUSART1_Write(',');
73.         EUSART1_Write('2'); /* default mode (no pin)*/
74.         EUSART1_Write('\r');
75.         /* blocco comando */
76.         FLAG_RN.CMD_SENT = 1;
77.         //RN_COMMAND = C_RN_COMMAND_NOCMD;
78.     }
79.     if (RN_BUFF[9] == 0x20)
80.     {
81.         /* svuoto buffer in ricezione dall'RN */
82.         for (uint8_t i = 0; i < C_RN_BUFF_DIM; i++)
83.         {
84.             RN_BUFF[i] = 0;
85.         }
86.         /* ricarica comando */
87.         FLAG_RN.CMD_SENT = 0;
88.         /* uscita dal comando*/
89.         RN_COMMAND = C_RN_COMMAND_SET_PIN;
90.     }
91.     break;
92.
93. case C_RN_COMMAND_SET_PIN:
94.     if (FLAG_RN.CMD_SENT == 0)
95.     {
96.         rn_bind = 0;
97.         EUSART1_Write('S');
98.         EUSART1_Write('P');
99.         EUSART1_Write(',');
100.        EUSART1_Write('1');
101.        EUSART1_Write('2');
102.        EUSART1_Write('3');
103.        EUSART1_Write('4');
104.        EUSART1_Write('4');
105.        EUSART1_Write('4');
106.        EUSART1_Write('\r');
107.        /* blocco comando */
108.        FLAG_RN.CMD_SENT = 1;
109.    }
110.    if (RN_BUFF[9] == 0x20)
111.    {
112.        /* svuoto buffer in ricezione dall'RN */
113.        for (uint8_t i = 0; i < C_RN_BUFF_DIM; i++)
114.        {
115.            RN_BUFF[i] = 0;

```

```

116.         }
117.         /* uscita dal comando*/
118.         //RN_COMMAND = C_RN_COMMAND_NOCMD;
119.         /* ricarica comando */
120.         FLAG_RN.CMD_SENT = 0;
121.         /* uscita da impostazioni */
122.         RN_COMMAND = C_RN_COMMAND_SET_DNAME;
123.     }
124.
125.     break;
126.
127. case C_RN_COMMAND_SET_DNAME:
128.     if (FLAG_RN.CMD_SENT == 0)
129.     {
130.         rn_bind = 0;
131.         EUSART1_Write('S');
132.         EUSART1_Write('N');
133.         EUSART1_Write(',');
134.         EUSART1_Write('L');
135.         EUSART1_Write('A');
136.         EUSART1_Write('M');
137.         EUSART1_Write('P');
138.         EUSART1_Write('\r');
139.         /* blocco comando */
140.         FLAG_RN.CMD_SENT = 1;
141.     }
142.     if (RN_BUFF[9] == 0x20)
143.     {
144.         /* svuoto buffer in ricezione dall'RN */
145.         for (uint8_t i = 0; i < C_RN_BUFF_DIM; i++)
146.         {
147.             RN_BUFF[i] = 0;
148.         }
149.         /* uscita dal comando*/
150.         //RN_COMMAND = C_RN_COMMAND_NOCMD;
151.         /* ricarica comando */
152.         FLAG_RN.CMD_SENT = 0;
153.         /* uscita da impostazioni */
154.         RN_COMMAND = C_RN_COMMAND_EXIT_CMD;
155.     }
156.
157.     break;
158.
159. case C_RN_COMMAND_SET_BT:
160.     if (FLAG_RN.CMD_SENT == 0)
161.     {
162.         rn_bind = 0;
163.         EUSART1_Write('S');
164.         EUSART1_Write('G');
165.         EUSART1_Write(',');
166.         EUSART1_Write('2');
167.         EUSART1_Write('\r');
168.         /* blocco comando */
169.         FLAG_RN.CMD_SENT = 1;
170.     }
171.     if (RN_BUFF[9] == 0x20)
172.     {
173.         /* svuoto buffer in ricezione dall'RN */
174.         for (uint8_t i = 0; i < C_RN_BUFF_DIM; i++)
175.         {
176.             RN_BUFF[i] = 0;

```

```

177.         }
178.         /* uscita dal comando*/
179.         //RN_COMMAND = C_RN_COMMAND_NOCMD;
180.         /* ricarica comando */
181.         FLAG_RN.CMD_SENT = 0;
182.         /* uscita da impostazioni */
183.         RN_COMMAND = C_RN_COMMAND_SET_DNAME;
184.     }
185.
186.     break;
187.
188.     //
189. case C_RN_COMMAND_EXIT_CMD:
190.     /* chiusura sessione di comando */
191.     if (FLAG_RN.CMD_SENT == 0)
192.     {
193.         //          EUSART1_Write('-');
194.         //          EUSART1_Write('-');
195.         //          EUSART1_Write('-');
196.         //          EUSART1_Write('\r');
197.
198.         EUSART1_Write('R');
199.         EUSART1_Write(',');
200.         EUSART1_Write('1');
201.         EUSART1_Write('\r');
202.
203.         FLAG_RN.CMD_SENT = 1;
204.         RN_COMMAND = C_RN_COMMAND_NOCMD;
205.     }
206.     break;
207. }
208.
209. /*.....
210. * proprietary code below
211. *..... */

```

A-III. The MGC_Decode.c file:

```
1. /*.....
2. * proprietary code above
3. *..... */
4. //New i2c message from MGC3130 to process
5. if(i2cMsgFlag) {
6.     i2cMsgFlag = 0;
7.     uint8_t cmd = mgcProcMsg(); //process the message
8.
9.     if(cmd != GI_NOGESTURE) {
10.        switch(cmd) {
11. //*****
12. //User code below for acting on GestIC command
13. //*****
14.            case GI_TAP_WEST:
15.            case GI_FLICK_WE:
16.                go_test = 0x00;
17.                EUSART1_Write(go_test);
18.                break;
19.
20.            case GI_TAP_EAST:
21.            case GI_FLICK_EW:
22.                go_test = 0x01;
23.                EUSART1_Write(go_test);
24.                break;
25.
26.            case GI_TAP_SOUTH:
27.            case GI_TAP_CENTER:
28.            case GI_FLICK_SN:
29.                go_test = 0x02;
30.                EUSART1_Write(go_test);
31.                break;
32.
33.            case GI_TAP_NORTH:
34.            case GI_FLICK_NS:
35.                go_test = 0x03;
36.                EUSART1_Write(go_test);
37.                break;
38.
39.            case GI_AIRWHEEL_CCW:
40.                /* azzeramento tempo di swipe */
41.                TIMER_D_TRACK = 0;
42.                Gesture_Track.air_ccw = 1;
43.                if (Gesture_Track.air_counter != 255)
44.                    Gesture_Track.air_counter++;
45.                else
46.                    Gesture_Track.air_counter = 0;
47.                //GestAir(md.sensorData.airWheelCounter);
48.                break;
49.
```

```
50.         case GI_AIRWHEEL_CW:
51.             /* azzeramento tempo di swipe */
52.             TIMER_D_TRACK = 0;
53.             Gesture_Track.air_cw = 1;
54.             if (Gesture_Track.air_counter != 255)
55.                 Gesture_Track.air_counter++;
56.             else
57.                 Gesture_Track.air_counter = 0;
58.             //go_test = md.sensorData.airWheelCounter;
59.             //GestAir(md.sensorData.airWheelCounter);
60.             break;
61.
62.         default: break;
63.     }
64. }
65.
66. /*.....
67.  * proprietary code below
68.  *..... */
```

Annex B: Test Application source code

The final test application, described in 6, is a ROS catkin package named “*niryo_one_test_app*” and is essentially a folder presenting the following layout:

```
niryo_one_test_app /  
  
    CMakeLists.txt  
  
    package.xml  
  
    setup.py  
  
    src /  
  
        device_BLE.py  
  
        dut_config.py  
  
        test_app.py  
  
        tester_commander.py  
  
        tester_config.py
```

All files in the package are individually explained in 6.2 and they are reported, in the same order, below:

B-I. The “CMakeList.txt” file:

```
1. cmake_minimum_required(VERSION 2.8.3)  
2. project(niryo_one_test_app)  
3.  
4.  
5. ## Find catkin macros and libraries
```

```

6.  ## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS
    xyz)
7.  ## is used, also find other catkin packages
8.  find_package(catkin REQUIRED COMPONENTS
9.      rospy
10.     roscpp
11.     std_msgs
12.     geometry_msgs
13.     niro_one_msgs
14.     actionlib
15.     moveit_msgs
16.     sensor_msgs
17.     std_srvs
18.     tf
19.     trajectory_msgs
20. )
21.
22. catkin_python_setup()
23.
24. #####
25. ## catkin specific configuration ##
26. #####
27. ## The catkin_package macro generates cmake config files for your
    package
28. ## Declare things to be passed to dependent projects
29. ## INCLUDE_DIRS: uncomment this if your package contains header
    files
30. ## LIBRARIES: libraries you create in this project that dependent
    projects also need
31. ## CATKIN_DEPENDS: catkin_packages dependent projects also need
32. ## DEPENDS: system dependencies of this project that dependent
    projects also need
33. catkin_package(
34.     INCLUDE_DIRS include
35.     # LIBRARIES robot_tester
36.     CATKIN_DEPENDS roscpp
37.     DEPENDS rospy
38. )
39.
40. #####
41. ## Build ##
42. #####
43.
44. ## Specify additional locations of header files
45. ## Your package locations should be listed before other locations
46. include_directories(
47.     # include
48.     ${catkin_INCLUDE_DIRS}
49. )

```


B-II. The “*package.xml*” file:

```
1.  <?xml version="1.0"?>
2.  <package format="2">
3.    <name>niryo_one_test_app</name>
4.    <version>0.1.0</version>
5.    <description>The niryo_one_test_app package</description>
6.
7.    <!-- One maintainer tag required, multiple allowed, one person
8.    per tag -->
9.    <!-- Example: -->
10.   <!-- <maintainer email="jane.doe@example.com">Jane
11.   Doe</maintainer> -->
12.   <maintainer email="c.poggi@ema-ic.it">Cristiano
13.   Poggi</maintainer>
14.
15.   <license>BSD</license>
16.
17.   <buildtool_depend>catkin</buildtool_depend>
18.   <depend>rospy</depend>
19.   <depend>roscpp</depend>
20.   <depend>actionlib</depend>
21.   <depend>std_msgs</depend>
22.   <depend>niryo_one_msgs</depend>
23.   <depend>geometry_msgs</depend>
24.
25.   <!-- The export tag contains other, unspecified, tags -->
26.   <export>
27.     <!-- Other tools can request additional information be placed
28.     here -->
29.   </export>
30. </package>
```

B-III. The “*setup.py*” script:

```
1.  from distutils.core import setup
2.  from catkin_pkg.python_setup import generate_distutils_setup
3.
4.  d = generate_distutils_setup(
5.    packages=['niryo_one_test_app'],
6.    #scripts=['bin/myscript'],
7.    package_dir={'': 'src'})
8.
9.
10. setup(**d)
11.
```

B-IV. The “dut_config.py” script:

```
1.  #!/usr/bin/env python
2.
3.  import device_BLE
4.
5.
6.  BLE_MAC_ADDR = "D8:80:39:F5:DE:D7"
7.
8.  BLE_NOTIFICATION_VALUE = 1
9.  BLE_NOT_VALID_DATA = 255
10.
11.  directions = {
12.      0 : "E",
13.      1 : "W",
14.      2 : "N",
15.      3 : "S",
16.      255: "INVALID"
17.  }
18.
19.
20.  class DUT():
21.      def __init__(self, name, device_addr = "0"):
22.          self.name = name
23.          self.ble_MAC_addr = device_addr
24.          self.bleThread = device_BLE.devThread(self.ble_MAC_addr)
25.          self.bleThread.daemon = True
26.          self.bleThread.start()
27.
28.      def setDutName(self, name):
29.          self.name = name
30.
31.      def setDutBleAddress(self, device_addr):
32.          self.ble_MAC_addr = device_addr
33.
34.      def getDutName(self):
35.          return self.name
36.
37.      def getDutBleAddress(self):
38.          return self.ble_MAC_addr
```

B-V. The “device_BLE.py” script:

```
1. #!/usr/bin/env python
2.
3. from bluepy import btle
4. import threading
5. import time
6. from dut_config import *
7.
8.
9. class DeviceBLE:
10.
11.     def __init__(self, device_addr = "0"):
12.         self.MAC_addr = device_addr
13.         self.dev = 0
14.         self.buffer_msgs = [BLE_NOT_VALID_DATA]
15.
16.         class devDelegate(btle.DefaultDelegate):
17.             def __init__(self):
18.                 btle.DefaultDelegate.__init__(self)
19.                 self.handle_data = BLE_NOT_VALID_DATA
20.
21.             def handleNotification(self, cHandle, data):
22.                 data = int(data.encode('hex'), 10)
23.                 self.handle_data = data
24.
25.
26.         def connect(self):
27.             # Connection to the BLE device
28.             print("Connecting to %s" %self.MAC_addr)
29.             try:
30.                 self.dev = btle.Peripheral(self.MAC_addr)
31.                 print("Connected to %s" %self.MAC_addr)
32.             except:
33.                 print("Unable to connect to BTLE device %s"
34.                     %self.MAC_addr)
35.                 retval = -1
36.
37.             try:
38.                 self.delegate = self.devDelegate()
39.                 self.dev.setDelegate(self.delegate)
40.                 # Enable notifications from BLE device
41.                 svc = self.dev.getServiceByUUID("49535343-fe7d-4ae5-
42.                     8fa9-9fafd205e455")
43.                 ch = svc.getCharacteristics()[2]
44.                 self.dev.writeCharacteristic(ch.valHandle+1,
45.                     "\x01\x00")
46.                 retval = 0
47.             except:
48.                 print("Unable to setDelegate of %s" %self.MAC_addr)
49.                 retval = -1
```

```

47.
48.         return retval
49.
50.     def waitForNotifications(self, ble_time):
51.         self.dev.waitForNotifications(ble_time)
52.         data = self.delegate.handle_data
53.         self.delegate.handle_data = BLE_NOT_VALID_DATA
54.         return data
55.
56.     class devThread (threading.Thread):
57.         def __init__(self, BLEAddr):
58.             threading.Thread.__init__(self)
59.             self.dev = DeviceBLE(BLEAddr)
60.             self.dev_data = BLE_NOT_VALID_DATA
61.             while (self.dev.connect() != 0):
62.                 time.sleep(1.0)
63.
64.         def run(self):
65.             while(1):
66.                 self.dev_data =
self.dev.waitForNotifications(BLE_NOTIFICATION_VALUE)
67.                 if (self.dev_data != BLE_NOT_VALID_DATA):
68.                     self.dev.buffer_msgs.append(self.dev_data)
69.                     #time.sleep(0.01)
70.
71.         def getLastData(self):
72.             return self.dev.buffer_msgs[-1]

```

B-VI. The “test_app.py” script:

```

1.  #!/usr/bin/env python
2.
3.
4.  from __future__ import print_function
5.  import rospy, sys
6.  from sys import exit as sysExit
7.  import device_BLE
8.  from dut_config import *
9.  from tester_commander import *
10. import time
11.
12. # Lib
13. import actionlib
14.
15. # Action
16. from niryo_one_msgs.msg import RobotMoveActionResult
17. from niryo_one_python_api.niryo_one_api import *
18.

```

```

19. def dutTest():
20.
21.     action_result = RobotMoveActionResult()
22.     start = False
23.     test_passed = False
24.     state = states["WAIT"]
25.     n = NiryoOne()
26.     n.calibrate_auto()
27.     while(1):
28.
29.         if (state == states["WAIT "]) :
30.             rospy.loginfo("Enter WAIT state
=====")
31.             rospy.loginfo("  Press enter to start DUT TEST ...")
32.             raw_input()
33.             start = True
34.             state = states["IDLE"]
35.
36.         elif (state == states["IDLE"]) :
37.             action_result = moveArmByPoseID(poses_id["idle_pose"])
38.             time.sleep(2)
39.             if (action_result.status) :
40.                 if (start) :
41.                     start = False
42.                     test_passed = False
43.                     state = states["TAKE_HAND"]
44.                 else:
45.                     rospy.loginfo("  Press enter to start DUT TEST ...")
46.                     raw_input()
47.                     start = True
48.             else: # action failed retry
49.                 pass
50.
51.         elif (state == states["TAKE_HAND"]) :
52.             openGripper()
53.             shift_up_pose = list(poses[poses_id["hand_pose"]])
54.             shift_up_pose[2] = shift_up_pose[2] + 0.1
55.             action_result = moveArmByPose(shift_up_pose)
56.             time.sleep(2)
57.             if (action_result.status) :
58.                 action_result = moveArmByPoseID(poses_id["hand_pose "])
59.                 time.sleep(2)
60.                 if (action_result.status) :
61.                     action_result = closeGripper()
62.                     time.sleep(2)
63.                     if (action_result.status) :
64.                         state = states["FIRST_GEST"]
65.                     else: # action failed retry
66.                         pass
67.                 elif (state == states["FIRST_GEST"]) :
68.                     action_result =
moveArmByPoseID(poses_id["firstGest_pose"])
69.                     time.sleep(1)

```

```

70.         if (action_result.status) :
71.
72.             try:
73.                 n.shift_pose(Axis_Y, -0.35)
74.             except NiryoOneException as e:
75.                 rospy.loginfo("    Niryo One Exception:")
76.                 print (e)
77.             finally:
78.                 mydata = dut.bleThread.getLastData()
79.                 if (directions[mydata] == "W") :
80.                     rospy.loginfo("CORRECT Gesture direction received
from DUT = %s", directions[mydata])
81.                     state = states["SECOND_GEST"]
82.                 else: # wrong message received retry
83.                     rospy.loginfo("WRONG Gesture direction received
from DUT = %s", directions[mydata])
84.                 #else: # action failed retry swipe
85.                 else: # action failed retry start pose
86.                     pass
87.
88.         elif (state == states["SECOND_GEST"]) :
89.             action_result =
moveArmByPoseID(poses_id["secondGest_pose"])
90.             time.sleep(1)
91.             if (action_result.status) :
92.                 try:
93.                     n.shift_pose(Axis_Y, 0.35)
94.                 except NiryoOneException as e:
95.                     rospy.loginfo("    Niryo One Exception:")
96.                     print (e)
97.                 finally:
98.                     time.sleep(1)
99.                     mydata = dut.bleThread.getLastData()
100.                    if (directions[mydata] == "E") :
101.                        rospy.loginfo("CORRECT Gesture direction received
from DUT = %s", directions[mydata])
102.                        state = states["THIRD_GEST"]
103.                    else: # wrong message received retry
104.                        rospy.loginfo("WRONG Gesture direction received
from DUT = %s", directions[mydata])
105.                    else: # action failed retry
106.                        pass
107.
108.        elif (state == states["THIRD_GEST"]) :
109.            action_result =
moveArmByPoseID(poses_id["thirdGest_pose"])
110.            time.sleep(1)
111.            if (action_result.status) :
112.                try:
113.                    n.shift_pose(Axis_X, -0.24)
114.                except NiryoOneException as e:
115.                    rospy.loginfo("    Niryo One Exception:")
116.                    print (e)

```



```

165.         rospy.loginfo("    TEST FAILED!!")
166.     else: # action failed retry
167.         pass
168.
169.     elif (state == states["TEST_PASSED"]) :
170.
171.         shift_up_pose = list(poses[poses_id["testPassed_pose"]])
172.         shift_up_pose[2] = shift_up_pose[2] + 0.1
173.         action_result = moveArmByPose(shift_up_pose)
174.         time.sleep(2)
175.         if (action_result.status) :
176.             action_result =
moveArmByPoseID(poses_id["TEST_PASSED"])
177.             time.sleep(2)
178.             if (action_result.status) :
179.                 try:
180.                     n.shift_pose(Axis_Y, 0.3)
181.                 except NiryoOneException as e:
182.                     rospy.loginfo("    Niryo One Exception:")
183.                     print (e)
184.                 finally:
185.                     time.sleep(5)
186.                     state = states["IDLE"]
187.         else: # action failed retry
188.             pass
189.
190.     elif (state == states["TEST_FAILED"]) :
191.         shift_up_pose = list(poses[poses_id["testFailed_pose"]])
192.         shift_up_pose[2] = shift_up_pose[2] + 0.1
193.         action_result = moveArmByPose(shift_up_pose)
194.         time.sleep(2)
195.         if (action_result.status) :
196.             action_result =
moveArmByPoseID(poses_id["TEST_FAILED"])
197.             time.sleep(2)
198.             if (action_result.status) :
199.                 try:
200.                     n.shift_pose(Axis_Y, -0.3)
201.                 except NiryoOneException as e:
202.                     rospy.loginfo("    Niryo One Exception:")
203.                     print (e)
204.                 finally:
205.                     time.sleep(5)
206.                     state = states["IDLE"]
207.         else: # action failed retry
208.             pass
209.
210.
211. if __name__ == '__main__':
212.     try:
213.         # Initializes a rospy node so that the SimpleActionClient
can publish and subscribe over ROS.
214.         rospy.init_node('niryo_one_app_test')

```



```

215.         # Initializes BTLE connection and thread
216.         dut = DUT('lamp', BLE_MAC_ADDR)
217.         # Initializes Action Client and tool
218.         setupNiryo()
219.         print ("TEST READY")
220.         dutTest()
221.
222.     except rospy.ROSInterruptException:
223.         print("program interrupted before completion",
            file=sys.stderr)

```

1.

B-VII. The “tester_commander.py” script:

```

1.     #!/usr/bin/env python
2.
3.     from __future__ import print_function
4.     import rospy, sys
5.     #from sys import exit as sysExit
6.     from tester_config import *
7.     import time
8.
9.     # Lib
10.    import moveit_commander
11.    import actionlib
12.
13.    # Action
14.    from niryo_one_msgs.msg import RobotMoveAction
15.    from niryo_one_msgs.msg import RobotMoveActionGoal
16.    from niryo_one_msgs.msg import RobotMoveActionResult
17.    from niryo_one_msgs.msg import RobotMoveCommand
18.    from niryo_one_msgs.msg import ToolCommand
19.
20.    # Messages
21.    from geometry_msgs.msg import Point
22.    from niryo_one_msgs.msg import RPY
23.    from niryo_one_msgs.msg import TrajectoryPlan
24.
25.    # Services
26.    from niryo_one_msgs.srv import SetInt
27.
28.
29.    from niryo_one_commander.move_group_arm import MoveGroupArm
30.    from copy import deepcopy
31.
32.    # for shift_pose function
33.    from niryo_one_python_api.niryo_one_api import *

```

```

34.
35.  AXIS_X    = 0
36.  AXIS_Y    = 1
37.  AXIS_Z    = 2
38.  ROT_ROLL  = 3
39.  ROT_PITCH = 4
40.  ROT_YAW   = 5
41.
42.  poses_id = {
43.      "idle" : 0,
44.      "hand" : 1,
45.      "firstGest" : 2,
46.      "secondGest" : 3,
47.      "thirdGest" : 4,
48.      "fourthGest" : 5,
49.      "testPassed" : 6,
50.      "testFailed" : 7,
51.  }
52.  poses = [idle_pose, hand_pose, EW_start_pose, WE_start_pose,
SN_start_pose, NS_start_pose, pass_start_pose, fail_start_pose]
53.
54.  states = {
55.      "idle" : 0,
56.      "takeHand" : 1,
57.      "firstGest" : 2,
58.      "secondGest" : 3,
59.      "thirdGest" : 4,
60.      "fourthGest" : 5,
61.      "letHand" : 6,
62.      "testPassed" : 7,
63.      "testFailed" : 8,
64.      "wait" : 9,
65.  }
66.
67.  client = 0
68.  action = RobotMoveAction()
69.
70.  def setupNiryo():
71.
72.      global client
73.      # Creates the SimpleActionClient, passing the type of the
74.      # action to the constructor.
75.      client =
76.      actionlib.SimpleActionClient('/niryo_one/commander/robot_action',
77.      RobotMoveAction)
78.
79.      # Waits until the action server has started up and started
80.      # listening for goals.
81.      connection_success = False
82.      while (connection_success == False):
83.          connection_success =
84.          client.wait_for_server(rospy.Duration(3))
85.          if (connection_success):

```

```

82.         rospy.loginfo(" Robot Connection established")
83.     else:
84.         rospy.logwarn(" Error connecting to Robot. Trying
again")
85.
86.     # Setting the gripper
87.     rospy.loginfo("Setting the gripper 1")
88.     rospy.wait_for_service('/niryo_one/change_tool/')
89.     try:
90.         changeToolSrv =
rospy.ServiceProxy('/niryo_one/change_tool/', SetInt)
91.         resp = changeToolSrv(toolID)
92.         rospy.loginfo(" Success: " + str(resp))
93.     except rospy.ServiceException, e:
94.         print ("Service call failed: "+str(e))
95.         rospy.logwarn(" Could not set the tool type. Trying again
in one second")
96.
97.
98. def moveArmByPose(pose):
99.
100.     global client, action
101.     action_goal = RobotMoveActionGoal()
102.     cmd = RobotMoveCommand()
103.     p = Point()
104.     rot = RPY()
105.     p.x = pose[0]
106.     p.y = pose[1]
107.     p.z = pose[2]
108.     rot.roll = pose[3]
109.     rot.pitch = pose[4]
110.     rot.yaw = pose[5]
111.     cmd.cmd_type = 2
112.     cmd.position = p
113.     cmd.rpy = rot
114.     rospy.loginfo(" Sending Move to pose [ %3f, %3f, %3f, %3f,
%3f, %3f] command =====", p.x, p.y, p.z,
rot.roll, rot.pitch, rot.yaw )
115.     # Creates a goal to send to the action server.
116.     action_goal.goal.cmd = cmd
117.     client.send_goal(action_goal.goal)
118.     # Waits for the server to finish performing the action.
119.     success = client.wait_for_result(rospy.Duration(30))
120.     if (success) :
121.         rospy.loginfo(" Robot Moved, SUCCESS!")
122.     else :
123.         rospy.logwarn(" Robot Not Moved, UNSECCCESS!")
124.     # Return the result of executing the action
125.     return client.get_result()
126.
127. def moveArmByPoseID(id):
128.
129.     global client, action, poses

```

```

130.     action_goal = RobotMoveActionGoal()
131.     cmd = RobotMoveCommand()
132.     p = Point()
133.     rot = RPY()
134.     p.x = poses[id][0]
135.     p.y = poses[id][1]
136.     p.z = poses[id][2]
137.     rot.roll = poses[id][3]
138.     rot.pitch = poses[id][4]
139.     rot.yaw = poses[id][5]
140.     cmd.cmd_type = 2
141.     cmd.position = p
142.     cmd.rpy = rot
143.     rospy.loginfo("  Sending command poses_id: %d
===== ", id )
144.     # Creates a goal to send to the action server.
145.     action_goal.goal.cmd = cmd
146.     client.send_goal(action_goal.goal)
147.     # Waits for the server to finish performing the action.
148.     success = client.wait_for_result(rospy.Duration(30))
149.     if (success) :
150.         rospy.loginfo("  Robot Moved, SUCCESS!")
151.     else :
152.         rospy.logwarn("  Robot Not Moved, UNSECCCESS!")
153.     # Return the result of executing the action
154.     return client.get_result()
155.
156.
157. def moveArmByTrajectory(traj):
158.
159.     global client, action
160.     action_goal = RobotMoveActionGoal()
161.     cmd = RobotMoveCommand()
162.     cmd.cmd_type = 7
163.     cmd.Trajectory = traj
164.     #rospy.loginfo("  Sending Move to trajectory [ %3f, %3f, %3f,
%3f, %3f, %3f] command =====", p.x, p.y, p.z,
rot.roll, rot.pitch, rot.yaw )
165.     # Creates a goal to send to the action server.
166.     action_goal.goal.cmd = cmd
167.     client.send_goal(action_goal.goal)
168.     # Waits for the server to finish performing the action.
169.     success = client.wait_for_result(rospy.Duration(30))
170.     if (success) :
171.         rospy.loginfo("  Robot Moved, SUCCESS!")
172.     else :
173.         rospy.logwarn("  Robot Not Moved, UNSECCCESS!")
174.     # Return the result of executing the action
175.     return client.get_result()
176.
177. def openGripper():
178.
179.     global client, action

```

```

180.     action_goal = RobotMoveActionGoal()
181.     cmd = RobotMoveCommand()
182.     tcmd = ToolCommand()
183.     tcmd.tool_id = 11
184.     tcmd.cmd_type = 1
185.     tcmd.gripper_open_speed = 300
186.     cmd.cmd_type = 6
187.     cmd.tool_cmd = tcmd
188.     rospy.loginfo("    Sending open gripper command
=====")
189.     # Creates a goal to send to the action server.
190.     action_goal.goal.cmd = cmd
191.     client.send_goal(action_goal.goal)
192.     # Waits for the server to finish performing the action.
193.     success = client.wait_for_result(rospy.Duration(30))
194.     if (success) :
195.         rospy.loginfo("    Gripper Opened, SUCCESS!")
196.     else :
197.         rospy.logwarn("    Gripper Not Opened, UNSECEESS!")
198.     # Return the result of executing the action
199.     return client.get_result()
200.
201. def closeGripper():
202.
203.     global client, action
204.     action_goal = RobotMoveActionGoal()
205.     cmd = RobotMoveCommand()
206.     tcmd = ToolCommand()
207.     tcmd.tool_id = 11
208.     tcmd.cmd_type = 2
209.     tcmd.gripper_close_speed = 300
210.     cmd.cmd_type = 6
211.     cmd.tool_cmd = tcmd
212.     rospy.loginfo("    Sending close gripper command
=====")
213.     # Creates a goal to send to the action server.
214.     action_goal.goal.cmd = cmd
215.     client.send_goal(action_goal.goal)
216.     # Waits for the server to finish performing the action.
217.     success = client.wait_for_result(rospy.Duration(30))
218.     if (success) :
219.         rospy.loginfo("    Gripper Closed, SUCCESS!")
220.     else :
221.         rospy.logwarn("    Gripper Not Closed, UNSECEESS!")
222.     # Return the result of executing the action
223.     return client.get_result()

```

B-VIII. The “*tester_config.py*” script:

```
1.  #!/usr/bin/env python
2.
3.  # Messages
4.  from geometry_msgs.msg import Point
5.  from niryo_one_msgs.msg import RPY
6.
7.
8.  toolID = 11    # Gripper 1
9.
10. # Artificial Hand pose
11. x_hand = 0.0
12. y_hand = 0.253
13. z_hand = 0.07
14. roll_hand = 0
15. pitch_hand = 0
16. yaw_hand = 1.572
17.
18. # Idle position pose
19. x_idle_pos = 0.193
20. y_idle_pos = 0.253
21. z_idle_pos = 0.15
22. roll_idle_pos = 0.0
23. pitch_idle_pos = -0.001
24. yaw_idle_pos = 0.001
25.
26. # Center position pose
27. x_center_pos = 0.24
28. y_center_pos = 0.0
29. z_center_pos = 0.162
30. roll_center_pos = 0.0
31. pitch_center_pos = 0.0
32. yaw_center_pos = 0.0
33.
34. # EW start position pose
35. x_EW_start_pos = 0.24
36. y_EW_start_pos = 0.125
37. z_EW_start_pos = 0.13
38. roll_EW_start_pos = -0.2
39. pitch_EW_start_pos = 0.0
40. yaw_EW_start_pos = 0.001
41.
42. # WE start position pose
43. x_WE_start_pos = 0.24
44. y_WE_start_pos = -0.125
45. z_WE_start_pos = 0.13
46. roll_WE_start_pos = 0.199
47. pitch_WE_start_pos = 0.003
48. yaw_WE_start_pos = -0.002
49.
```

```

50. # NS start position pose
51. x_NS_start_pos = 0.143
52. y_NS_start_pos = 0.0
53. z_NS_start_pos = 0.13
54. roll_NS_start_pos = 0.001
55. pitch_NS_start_pos = 0
56. yaw_NS_start_pos = -0.001
57.
58. # SN start position pose
59. x_SN_start_pos = 0.373
60. y_SN_start_pos = 0.0
61. z_SN_start_pos = 0.13
62. roll_SN_start_pos = 0.001
63. pitch_SN_start_pos = 0
64. yaw_SN_start_pos = 0.002
65.
66. # Fail start position pose
67. x_fail_start_pos = 0.24
68. y_fail_start_pos = 0.172
69. z_fail_start_pos = 0.035
70. roll_fail_start_pos = 0.0
71. pitch_fail_start_pos = 0.121
72. yaw_fail_start_pos = -1.57
73.
74. # Pass start position pose
75. x_pass_start_pos = 0.24
76. y_pass_start_pos = -0.172
77. z_pass_start_pos = 0.035
78. roll_pass_start_pos = -0.001
79. pitch_pass_start_pos = 0.121
80. yaw_pass_start_pos = 1.57
81.
82. #####
83.
84. # Poses definition [ x, y, z, r, p, y ]
85. hand_pose = [x_hand, y_hand, z_hand, roll_hand, pitch_hand,
yaw_hand]
86. idle_pose = [x_idle_pos, y_idle_pos, z_idle_pos, roll_idle_pos,
pitch_idle_pos, yaw_idle_pos]
87. center_pose = [x_center_pos, y_center_pos, z_center_pos,
roll_center_pos, pitch_center_pos, yaw_center_pos]
88. EW_start_pose = [x_EW_start_pos, y_EW_start_pos, z_EW_start_pos,
roll_EW_start_pos, pitch_EW_start_pos, yaw_EW_start_pos]
89. WE_start_pose = [x_WE_start_pos, y_WE_start_pos, z_WE_start_pos,
roll_WE_start_pos, pitch_WE_start_pos, yaw_WE_start_pos]
90. NS_start_pose = [x_NS_start_pos, y_NS_start_pos, z_NS_start_pos,
roll_NS_start_pos, pitch_NS_start_pos, yaw_NS_start_pos]
91. SN_start_pose = [x_SN_start_pos, y_SN_start_pos, z_SN_start_pos,
roll_SN_start_pos, pitch_SN_start_pos, yaw_SN_start_pos]
92. pass_start_pose = [x_pass_start_pos, y_pass_start_pos,
z_pass_start_pos, roll_pass_start_pos, pitch_pass_start_pos,
yaw_pass_start_pos]

```

```
93. fail_start_pose = [x_fail_start_pos, y_fail_start_pos,  
    z_fail_start_pos, roll_fail_start_pos, pitch_fail_start_pos,  
    yaw_fail_start_pos]
```