POLITECNICO DI TORINO

Master's Degree

in

Mechatronic Engineering

Master's Thesis

Testing of a Docking Mechanism with a 6 Degree of Freedom Manipulator and a Force-Torque Sensor

Thesis Supervisor
Marcello Chiaberge

Candidate
Alberto Combina

Academic Year 2019/2020

Dedication

For mom, dad, Ludovica, Gabriele, Vanessa and Zack.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my thesis supervisor Prof. Marcello Chiaberge and to all the PIC4SeR collaborators, in particular Gianluca Dara, Dario Gandini and Luigi Mazzara. Their guidance, patience and motivation deeply helped me during all the time of research and writing of this thesis.

I want to thank my supervisors at Thales Alenia Space, Andrea Merlo, Ciro Napolitano and Genny Scalise for encouraging me in the most stressful moments and not letting me give up, becoming friends as well as colleagues.

My appreciation also extends to my laboratory colleagues, Paolo Bocchiardi, Gabriele Gennaro and Giuseppe Terranova, to whom I wish every success for the future.

Contents

List of Figures

Chapter 1

Introduction

A docking maneuver is defined as the joining of two separate free-flying space vehicles in space, usually called chaser or servicer and target or customer. This delicate operation is crucial in on-orbit servicing missions and has proved its value in the last decades. Thales Alenia Space and "Politecnico di Torino", after an intensive research on the already existing solutions, developed an active soft docking system for the SAPERE-STRONG mission, a program co-funded by MIUR (Italian Ministry of Education, Universities, and Research). The aim of this work is to simulate a docking maneuver with the Schunk Powerball LWA 4P, a 6 degree of freedom robotic arm, equipped with the Schunk FTM115 force torque sensor, in order to test the mechanism. To accomplish this task, the control strategies were implemented to work with a pre-existing framework for the manipulator, customly designed by "Università di Genova", in particular by "Genoa Robotics And Automation Laboratory" (G.R.A.A.L), and with the PCAN-PCI adapter, essential to communicate with the sensor.

This dissertation contains, besides this introductory part, five chapters. The first one defines what are rendezvous and docking operations and their applications, and it also describes the main characteristics of the docking mechanism.

Chapter 3 explains in detail the most relevant aspects of the framework developed by G.R.A.A.L to drive the Schunk LWA 4P manipulator, giving a general description of the entire structure, but with particular attention on the higher levels of abstraction and the elements with which the user can interact, such as the terminal commands, the finite state machine of the controller and the graphical user interface.

Chapter 4 gives an overview of the *Controller Area Network* (CAN) protocol, starting from how it was born to the actual state of the ISO 11898 and the main features of the protocol today. It continues with a detailed description of how it is implemented for the Schunk FTM115, how to interact with the sensor and how to properly manage collected data and errors.

Chapter 5 describes the phases to test the mechanism and the possible control strategies suitable for the job, explaining why the classical impedance control approach was not implementable. Because of its lacks, two solutions are reported.

Chapter 6 presents the conclusions of this work.

After the conclusions there are two Appendices, the first one aims to be a small guide for any user to properly start up, shut down and use the the manipulator and the framework, avoiding critical situations that can compromise the correct functioning of the arm. In the second appendix it is possible to find the code that implements the control algorithms described in Chapter 5.

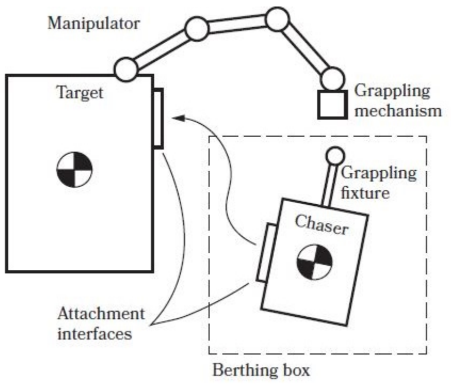
Chapter 2

The Docking Mechanism

2.1 Mating operations and On-Orbit Servicing

On-orbit servicing consists on the execution of refueling, repair, assembly and upgrade satellites after they are launched. Most satellites consist of expensive components, so, using a servicing spacecraft to repair or replace critical units or to move the satellite into another orbit, will save additional utility from what would have been a loss. A clear example are the service missions of the Hubble Space Telescope (HST) [5]. Moreover, if the goal is to travel to distant destinations, depots are a potential enabling infrastructure, so satellite servicing is needed in order to constitute such depots. The servicing capabilities can also be applied for removing debris in the low Earth Orbit (LEO) and in the Geostationary Earth Orbit (GEO) [5].

In all the applications of on-orbit servicing there is always the necessity to precisely couple the two crafts involved, so during the past fifty years two techniques were developed: berthing and docking [1], showed respectively in figures 2.1 and 2.2. Berthing deals with joining two spacecraft, in which the inactive one is placed into the mating interface of the other using a robotic arm.



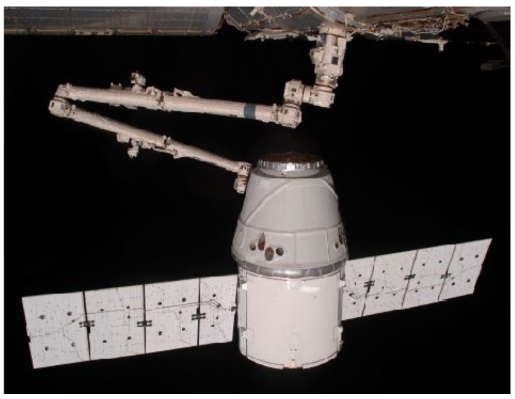


Figure 2.1: Berthing operation schematic and the Canadarm2 manipulator of the ISS grappling a Dragon spacecraft (credit: NASA)

According to [1], it is divided into different phases. For example the following ones are necessary to complete the mating between the ISS and a visiting satellite [16]:

1. Acquisition of the berthing box.
2. Positioning of the arm in the ”ready position”, done after the acquisition of the berthing box for safety reasons.
3. Switch-off the thrusters of the chaser and initiation of capture.
4. Grappling of the capture interface by the manipulator.
5. Transfer to the berthing port.
6. Insertion into the reception interfaces.
7. Structural connection.

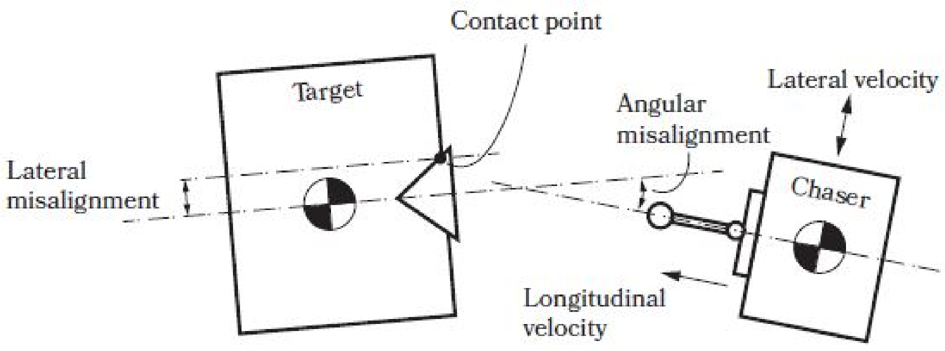


Figure 2.2: Docking operation

Instead, in the docking operation, the Guidance, Navigation, and Control (GNC) system of the chaser (or servicer) satellite controls the relative state with the target (or customer) so as to ensure suitable contact conditions (relative misalignments, relative velocities, etc) [16]. It is generally composed by four main steps: approach, alignment, soft docking and hard docking. Initially the chaser approaches the target and the two interfaces are prepared for the impact, and by geometrical features or by other means, the poses are aligned. Then, with a first connection that still allows relative displacements and rotations, usually called soft docking, a great amount of the energy associated to the relative velocity of the two satellites is safely dissipated. Finally a firm connection between the two is ensured by the hard docking, where a secondary fastening device creates a stiff and final connection [16].

The aim of the STRONG mission is the development of a reusable space tug with electric propulsion. This tug should be able to deploy satellite platforms from low injection orbits into their final destination orbits with considerable savings in weight and a strong optimization of the payload/platform ratio. The tug has also the aim of allowing the re-entry of vehicles for the retrieval of payload samples. After each orbit raising, the space tug is expected to perform an on-orbit refueling with an orbital tank [16]. Figure 2.3 shows a graphical representation of the mission.

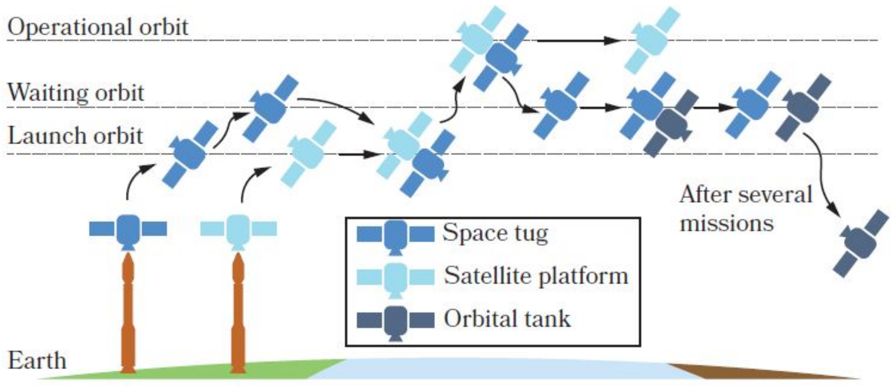


Figure 2.3: Representation of the Strong mission

Due to the performance of the GNC of the tug, the docking mechanism has to recover some position and angular misalignments as well as to dissipate the energy associated with the relative velocities between the spacecraft [16].

Figure 2.4 shows four different types of central docking mechanisms suitable for the STRONG mission. The first one, in the top left corner is a 6SPS parallel manipulator with impedance control, composed by an active half based on a Stewart-Gough platform and a drogue as passive half. In the top right corner a compliant articulated arm mechanism can be seen. The bottom left corner shows the central docking mechanism developed by Thales Alenia Space Italia (TAS-I) during the STEPS project. Finally, in the bottom right corner a central active mechanism, where the active part is equipped with a linear actuator for controlling the longitudinal approaching between the chaser and the target. A trade study was performed to choose the most balanced solution, considering the following criteria [16]:

- Mass of the mechanism, to be taken into consideration to reduce the cost of the mission.
- Mechatronic complexity, composed by mechanical complexity, sensors and actuators.
- Control complexity, that evaluates a system in terms of the control architecture needed to use it properly during the docking maneuver.
- Compatibility with different spacecraft masses.
- Energy consumption to drive sensors and actuators.
- Expected reliability and functional confidence.

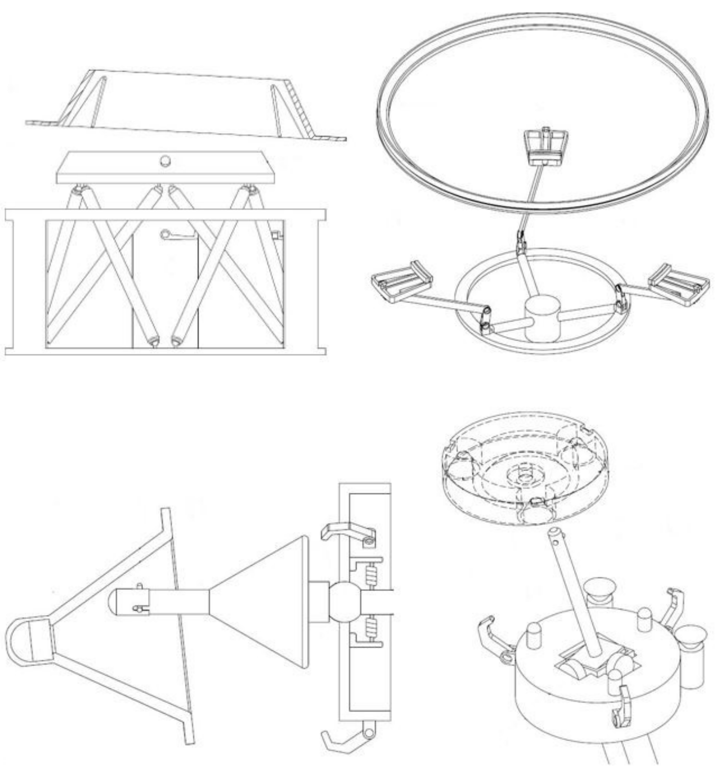


Figure 2.4: Four possible docking mechanisms for the STRONG mission

After evaluating all these parameters for each candidate, the STEPS' probe and drogue was further investigated [16].

2.2 Active central docking mechanism

In the first design iteration, a modified version of the STEPS' system, passively controlled with two sets of springs, was developed. The 2D and 3D simulations showed that the passiveness and the lack of dumping devices were the two main problems, mostly in the soft docking phase. On the other hand, the second design iteration produced the final active docking mechanism, which required to be tested with the Schunk LWA 4P described in this section. Figure 2.5 shows a schematic view of the active part and its main components [16].

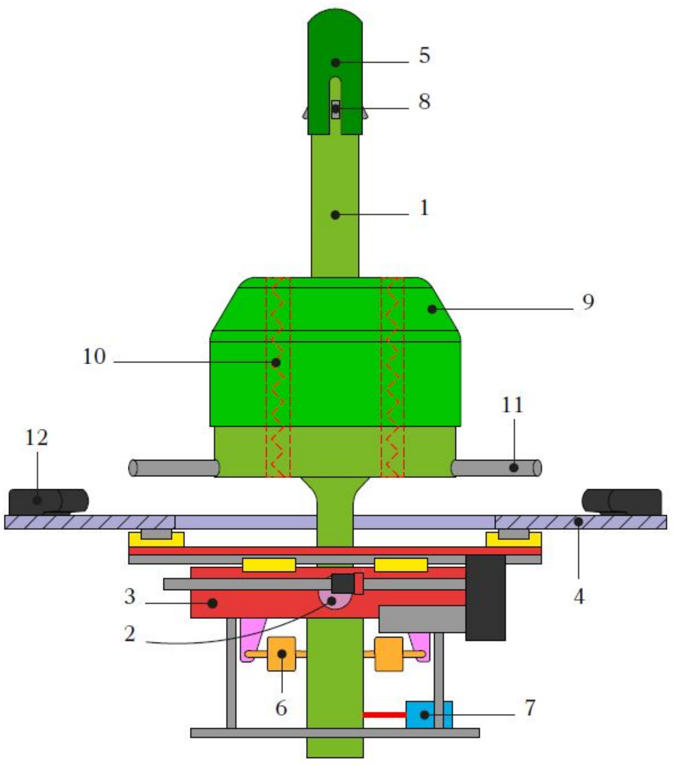


Figure 2.5: Schematic view of the active part

The probe (1) uses the same retraction system developed by Thales Alenia Space during the project STEPS (figure 2.6). To achieve the translational movement needed after the soft docking phase, an electric motor and a ball screw, rigidly connected to the internal part of the probe, are used [16]. The motor rotates the ball screw through a planetary gearhead and since the internal probe must only translate relative to the external part, three dowel pins are connected to it. In addition to these pins, the external probe has three built-in tracks that combined with three low friction Torlon 4301 keys prevent the rotation of both the external and internal parts of the probe. The cap of the probe (5) contains a set of springs to avoid large contact forces with the passive half during the first phase of the maneuver [16].

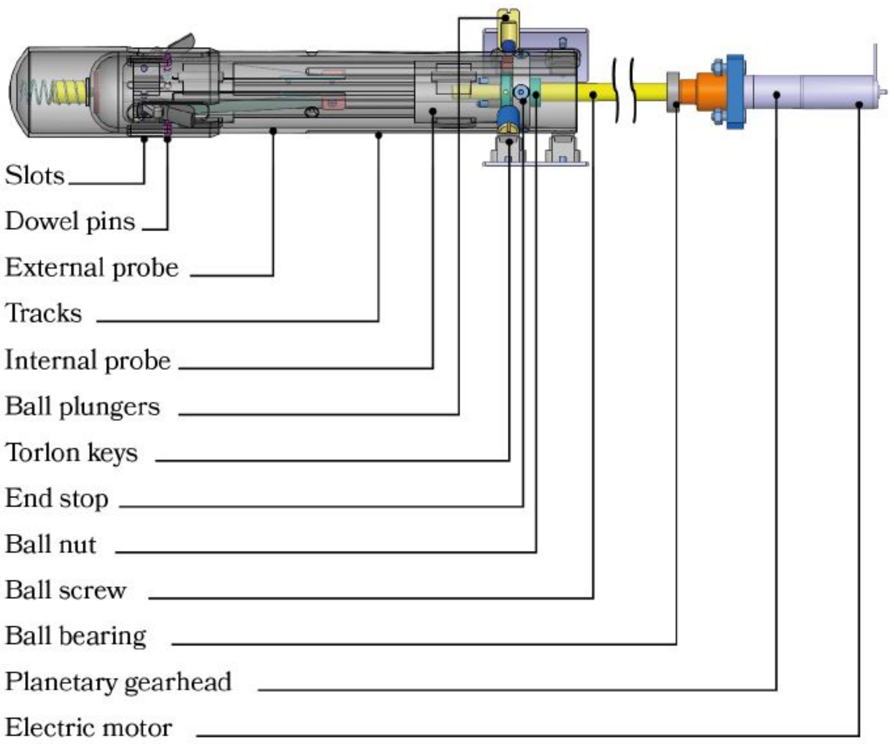


Figure 2.6: View of the internal elements of the retraction system of the STEPS' docking mechanism

In order to constrain the roll of the probe, a universal joint (2) is used. It is mounted on a slide (3) that translates, driven by an actuator based on a ball screw on linear rails. The slide plus the rails can translate orthogonally since the whole is mounted on an analogous slide equipped with a linear actuator. The rails of this second slide are bolted to the mechanism support plate (4), that is the interface between chaser and target [16]. In order to mechanically re-align the probe with the longitudinal axis of the servicer spacecraft traction springs (6) are used, while mutually orthogonal laser triangulation displacement sensors (7) measure the displacements of the rear end of the probe [16]. Linear actuators use these measurements as reference to move the slides on which the probe is mounted, in order to reduce the contact force.Once the spring loaded petals (8) overpass the female socket of the passive element (figure 2.7), the soft docking is completed, and a damper mounted at the end of the socket dissipates the energy associated with the relative velocities.

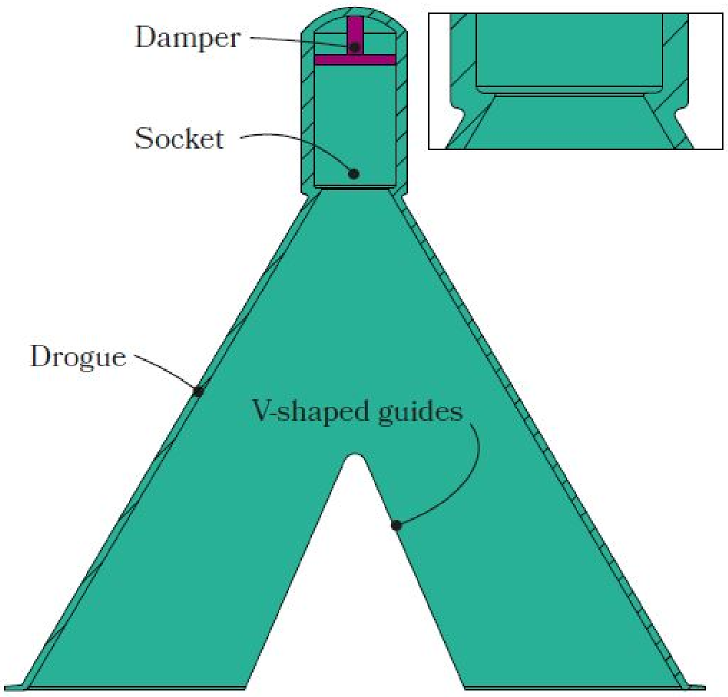


Figure 2.7: Schematic view of the passive part

Then the probe retracts and the contact between the male cone (9) and the female cone eliminates the pitch and yaw misalignments [16]. During the retraction, elastic energy is stored with the compression springs (10) mounted inside the male cone, which will be used during the undocking operation. Three rods (11), in combination with the v-shaped guides of the passive half, are used to eliminate roll misalignments. After that the linear actuators have returned to their nominal position on the base of the female cone is in contact with the support plate of the chaser, three radial hooks (12) secure the two parts, completing the hard docking.

2.3 Docking maneuver phases

A general introduction to what is intended for docking in space was already given in section 2.1, meanwhile this section the four phases are explained considering to use the active central docking mechanism described in the previous section. Referring to figure 2.8, it is possible to see how the mechanism works during the entire maneuver. During phase 1, **Approach and Deployment**, the axis of the passive half and the active one are misaligned. At the end of this phase the tip of the probe and the drogue are in contact. To compensate the contact forces between the two parts, the linear actuators move the slides accordingly to the inputs coming from the laser displacement sensors, in order to complete the **Alignment** phase.

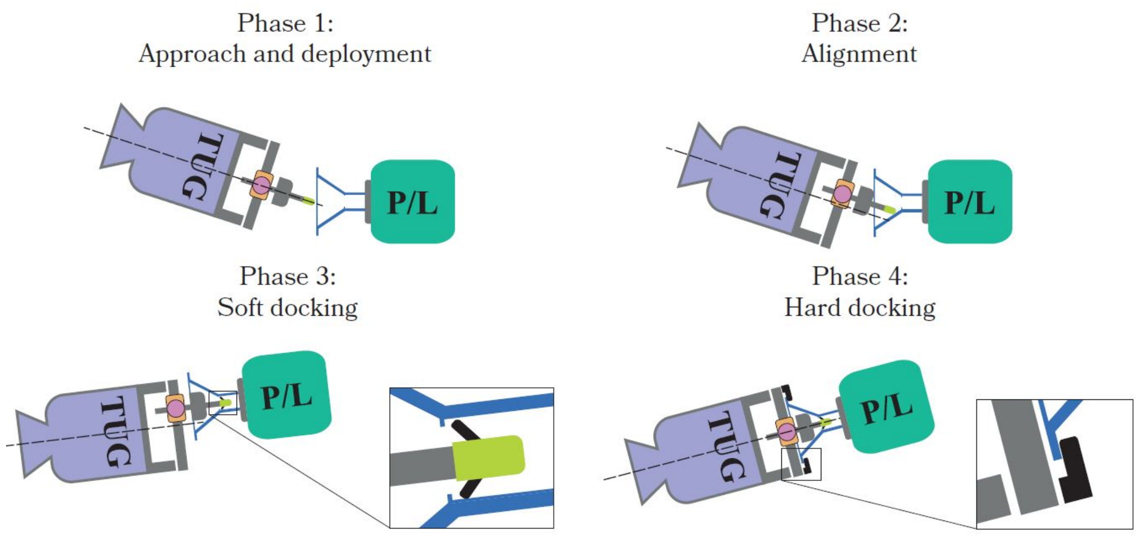


Figure 2.8: Phases of the docking maneuver

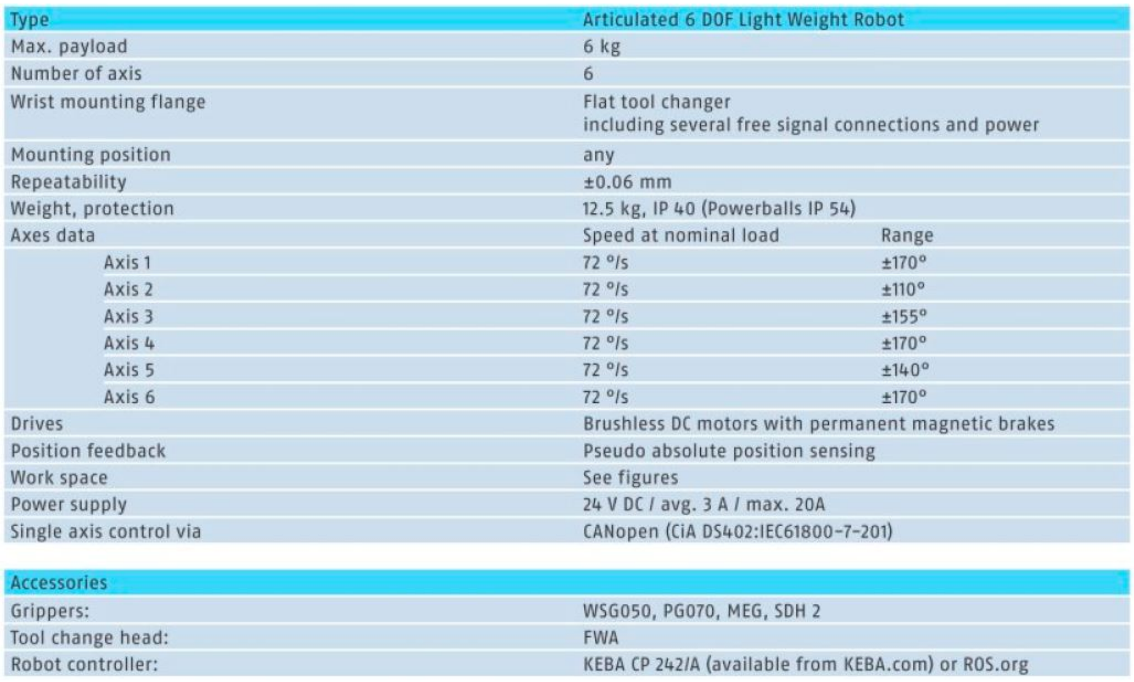
An active alignment is necessary for the success of the maneuver. The STEPS mechanism, that was passively controlled by two groups of springs, was re-designed because the simulations showed that any misalignment could produce too high contact forces and, in the worst case, a contact between the base of the passive half and the male cone. This contact in reality would make the two spacecrafts bounce away from each other before the soft docking is completed [16].

When the spring loaded petals overpass the female socket, the **Soft docking** phase is completed. At this point small relative movements are still allowed due to tolerances, but the two axis are now almost coincident. To complete the last phase, the **Hard docking**, the active element has to be moved back to its initial position by the linear actuators. Then the probe is retracted until contact between the base of the drogue and the support plate. Finally the three radial hooks secure the two parts, completing the hard docking [16].

Chapter 3

Schunk Powerball LWA 4P

The Schunk Powerball Light Weight Arm (LWA) 4p is a 6 Degree-of-Freedom (DOF) arm produced by Schunk, leader in manipulator design [17]. It has high gear ratio for each axes, in order to stand gravitational loadings, but it can still be driven for impedance control and friction identification [13]. One of the most interesting features of the LWA 4P is that it is modular, so it's compatible with a number of accessories, such as the Force/Torque sensor FTM115. In figure 3.1 are reported the most relevant technical aspects of the robotic arm.



Type	Articulated 6 DOF Light Weight Robot	
Max. payload	6 kg	
Number of axis	6	
Wrist mounting flange	Flat tool changer including several free signal connections and power	
Mounting position	any	
Repeatability	±0.06 mm	
Weight, protection	12.5 kg, IP 40 (Powerballs IP 54)	
Axes data	Speed at nominal load	Range
Axis 1	72 °/s	±170°
Axis 2	72 °/s	±110°
Axis 3	72 °/s	±155°
Axis 4	72 °/s	±170°
Axis 5	72 °/s	±140°
Axis 6	72 °/s	±170°
Drives	Brushless DC motors with permanent magnetic brakes	
Position feedback	Pseudo absolute position sensing	
Work space	See figures	
Power supply	24 V DC / avg. 3 A / max. 20A	
Single axis control via	CANopen (CiA DS402:IEC61800-7-201)	

Accessories	
Grippers:	WSG050, PG070, MEG, SDH 2
Tool change head:	FWA
Robot controller:	KEBA CP 242/A (available from KEBA.com) or ROS.org

Figure 3.1: Powerball main features

The manipulator is essentially composed by [17]:

- A base for mechanical mounting and for housing the electrical connections.
- Three double-axis rotary modules, ERB145 for joints 1-2 and 3-4, ERB 115 for

joints 5-6.

- A coupling flange for an end-effector/gripper or sensor.
- Connecting elements between the rotary modules.

It is equipped with a CAN bus interface with the CANopen protocol, dedicated to the rotary modules, and a second CAN bus interface, in this case used for the FTM 115, as well as an RS232 one. Since the arm has to be fed with a 24 V power supply and requests low current inputs, it can be alimented either via a power supply or battery.

3.1 The Framework

Università di Genova and, in particular, *Genoa Robotics and Automation Laboratory (GRAAL)* developed, starting from the real time operative system ORTOS (based on CentOS 6.5) [14], a modular framework adaptable to different manipulators, in order to have full control of these items and let the robotic control system developer concentrate only on the algorithmic part [3].

The **Framework** was developed using the Object Oriented Paradigm (OOP) in C++, and is divided in growing levels of abstraction. Each one has its own tasks, that are connected and managed in real time. The lowest level, the *Kernel Abstraction Layer (KAL)*, has been developed to obtain independence of the control algorithms' code from the underlying software platform and to increase portability, since all the specific calls of the used operative system are concentrated here [3]. The second layer is the *WorkFrame (WF)*, which takes care of centralizing the control of the system resources and helping the user to administer the whole system, for minimizing the code lines not directly related to the control system algorithm implementation [3]. In particular, it implements the tasks and functions related to the controller, the console and the finite state machine. The calls provided by the *KAL* are exploited by the *Network (NET)* layer, whose job is to abstract the communication mechanisms from the underlying physical data channels [3]. To allow a standard communication mechanism between control tasks that has a minimum impact on the algorithmic part, the *Black Board System (BBS)* layer was implemented [3]. It's also important to underline that at this point the code is not strictly related to the LWA 4P, but is adaptable to every manipulator once its mathematical model is correctly linked to the framework.

3.1.1 Finite State Machine, Controller and Console

To obtain the compliant control of the arm and since the Framework is not modifiable by the developer, it is not convenient to dive into every aspect of this compounded structure. Is better to stay at an higher level and to concentrate on the three most important elements of the control structure: the **Finite State Machine (FSM)**, which describe every operating condition of the **Controller**, and that can be driven by the user with the **Console**. In figure 3.2 is reported a schematic of the FSM, whose core states are [14]:

- **INIT**: once the workframe starts its preliminary operations it has also to check if the arm is ready to be used and, if everything goes well, the state is automatically switched to disarm.
- **DISARM**: the controller is correctly initialized but the manipulator is still disarmed, so the brakes are still active.
- **HOLDING**: the arm is ready to work and current position is maintained, so brakes are enabled until the user selects the next command.
- **JOINT POSITION CONTROL**: the controller makes the arm reach a position reference, specified by the user, in the joint space.
- **JOINT VELOCITY CONTROL**: the controller makes the arm follow a velocity reference, specified by the user, in the joint space, for a predefined amount of time.
- **Cartesian POSITION CONTROL**: the controller makes the arm reach a position reference, specified by the user, in the Cartesian space.
- **Cartesian VELOCITY CONTROL**: the controller makes the arm follow a velocity reference, specified by the user, in the Cartesian space, for a predefined amount of time.
- **APPROACHING and PARKING**: if the user wants to disarm the manipulator, the controller makes it reach automatically an exit position first (approaching), and then a parking one (parking).
- **FAILURE**: if the controller or another part of the framework detects an error, the robot is automatically disarmed.

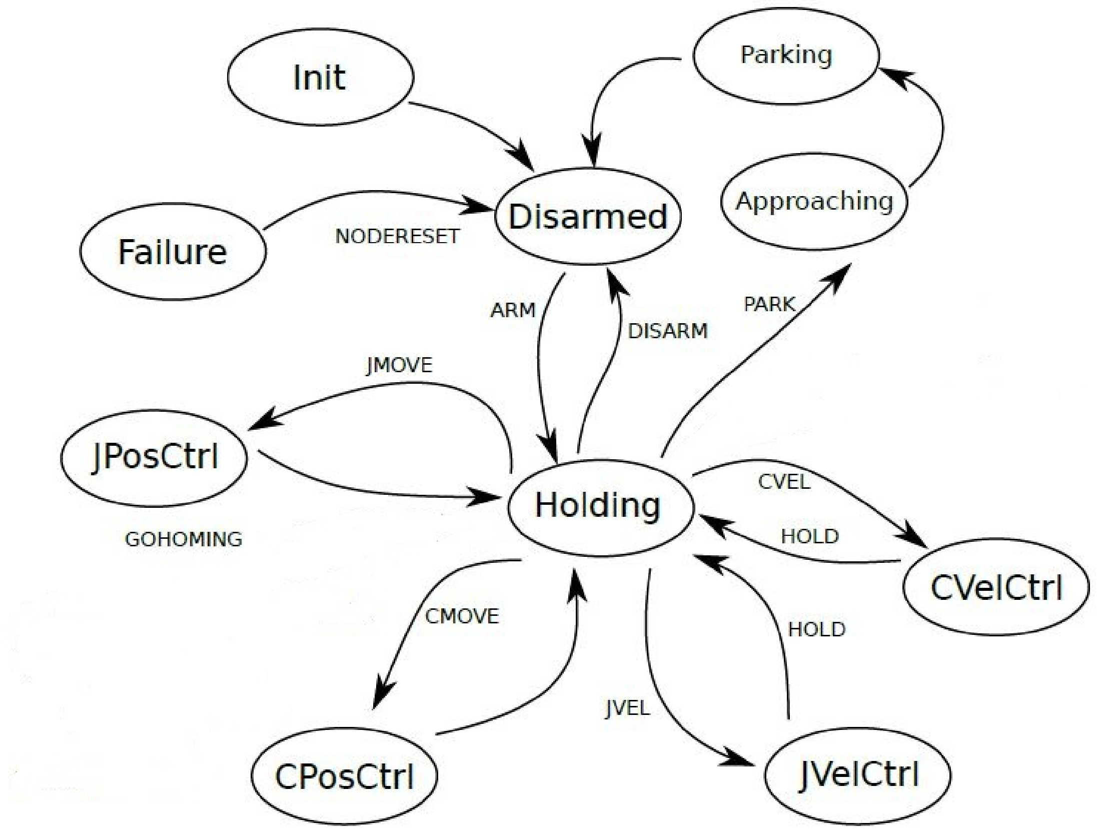


Figure 3.2: Finite State Machine

The controller changes its state for two reasons: a **Command** or an **Event**. Events are automatically managed by the framework and are generated if an error occurred or if the controller has executed the requested operation. For example, if the user wants the arm to reach a specific pose in the Cartesian space, once this assignment is correctly concluded, the framework will make the state of controller switch back from *CPosCtrl* to *Holding*. On the other side the user can act actively on the controller using the console, that in practice consists of a set of commands provided to the controller through the command line or with a specific GUI for the LWA 4P (section 3.2.2).

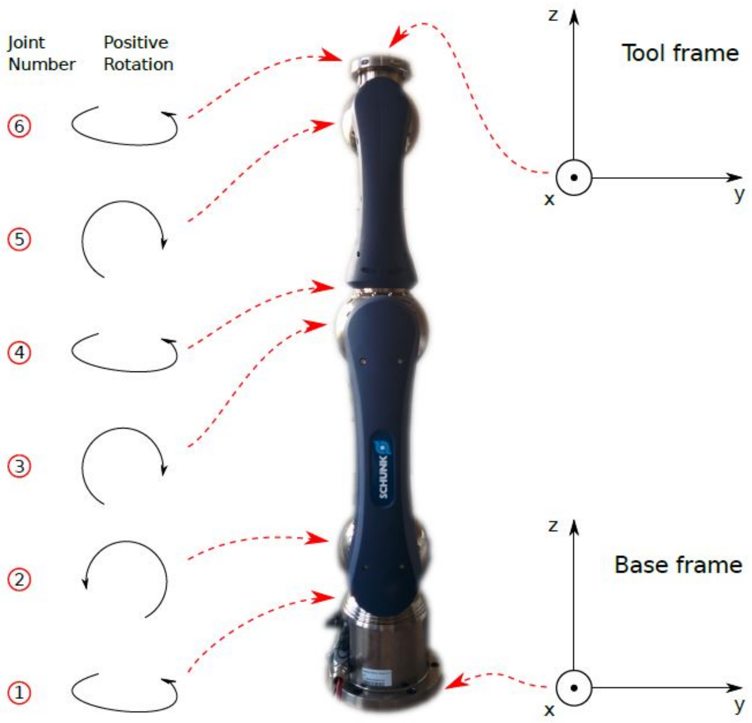


Figure 3.3: Powerball joints' direction and frames' poses

The commands that are implemented at the framework level [14] refer to figure 3.3 for the joint number, direction and base and tool frames pose:

- **arm**: arm the robot, enabling the actuators and switching the state of the controller from disarmed to holding.
- **cgain <*value*>** and **jgain <*value*>**: change, respectively, the Cartesian or the Joint position loop gains to the specified *value*.
- **cmove <*number*><*value*>** and **chmove <*number*><*value*>**: move the arm in Cartesian space (Cartesian position control) and need two variables *number* and *value*, that represent respectively the Catesian axis and the wanted pose, measured in meters. Table 3.1 summarises all the possible values for the *number*:

Number	Axis
1	yaw
2	pitch
3	roll
4	x
5	y
6	z

Table 3.1: *Number* possible values for Cartesian position control

The only difference between cmove and chmove is that the first always moves the tool frame with respect to the base frame, the second one uses the *Cartesian Helper* (see 3.1.2). For example, if a movement of 10 cm in the positive x direction is wanted to be performed, the user can user the command as follows:

```
cmove 4 0.10
```

- **cmoveall<*v1*><*v2*><*v3*><*v4*><*v5*><*v6*>** and **chmoveall<*v1*><*v2*><*v3*><*v4*><*v5*><*v6*>**: move the arm in the Cartesian space (Cartesian position control), with the possibility to specify a *value* for each DOF (rotation are in radianss, translations in meters). So six input are expected, following the order reported in table 3.1. Even for these command is possible to choose between the standard or the one with the Cartesian Helper (see 3.1.2), here follows an example, a rotation around the x axis of 0.1 rad and a translation of 30 cm along z, in the negative direction:

```
cmoveall 0 0 0.1 0 0 -0.3
```

- **cvel<*number*><*value*>** and **chvel<*number*><*value*>**: similarly to cmove and chmove, move the robot with a specified Cartesian velocity, again using as input a *number*, that this time follows the order described in table 3.2 and a *value*.

Number	Axis
1	roll
2	pitch
3	yaw
4	x
5	y
6	z

Table 3.2: *Number* possible values for Cartesian velocity control

The requested velocity value will be maintained for a predefined amount of time (default = 3000 ms), that can be changed with the GUI or with the command **parvtime**. Rotation as to be specified in rad/s, translation in m/s. Example of a rotation around y (pitch), using the Cartesian Helper, with a velocity of 0.03 rad/s:

```
chvel 2 0.03
```

- **cvelall<*v1*><*v2*><*v3*><*v4*><*v5*><*v6*>** and **chvelall<*v1*><*v2*><*v3*><*v4*><*v5*><*v6*>**: the structure is the same of cmoveall and chmoveall, but these use the axis' order of table 3.2. Rotation as to be specified in rad/s, translation in m/s. Example of a rotation around x (roll), using the Cartesian Helper, with a velocity of 0.01 rad/s and two translation, along x and y at 10 cm/s:

  ```
  chvelall 0.01 0 0 0.1 0.1 0
  ```

- **cpos** and **jpos**: print the current arm Cartesian or Joint position respectively.
- **disarm**: disarm the robot, disabling the actuators and enabling the brakes.
- **exitpark**: exit from park position, moving to a predefined exit park position.
- **help**: print all the available commands with a short description.
- **hold**: hold the robot in the current position.
- **jmove<*number*><*value*>**: referring to figure 3.3 for the joint *number* and positive direction, move a joint to a specified *value* (rad). For example, a movement of the last joint of -0.5 rad:

  ```
  jmove 6 -0.5
  ```

- **jmoveall<*v1*><*v2*><*v3*><*v4*><*v5*><*v6*>**: move all the joints of the arm simultaneously, specifying six rotations in radianst. For example, to move joints 1, 4 and 6 of respectively 0.5, -0.3 and 1 rad, the user can use the following command from the console:

  ```
  jmoveall 0.5 0 0 -0.3 0 1
  ```

- **jvel<*number*><*value*>**: move a joint at a specified velocity (rad/s). This command needs two input: a *number*, that represent which joint the user want to move, and a *value*, that is the angular velocity. The rotation will last for a default time (3000 ms), that can be modified with the GUI or with the command **parvtime**. For example, a movement of joint 4 at 0.3 rad/s can be requested with:

  ```
  jvel 4 0.3
  ```

- **jvelall<*v1*><*v2*><*v3*><*v4*><*v5*><*v6*>**: move all the joints of the arm simultaneously, specifying six *values*, that are angular velocities (rad/s). For example, to move all the joints at 0.5 rad/s, the user can use the following command from the console:

```
jmoveall 0.5 0.5 0.5 0.5 0.5 0.5
```

- **parcalg<*alg*>**: set Cartesian algorithm (J = jacobian, I = iterative, T = task-based).
- **parcsat <*lin*><*ang*>** : set Cartesian linear and angular velocity saturation.
- **parcth <*lin*><*ang*>**: set Cartesian linear and angular position error threshold.
- **paroer** : enable or disable the end of races.
- **parexited**: set joint positions for exited preset.
- **parfen**: enable or disable joint velocity filter and Cartesian filter.
- **parfvel**: display joint velocity filter parameters.
- **pargfvel**: display gripper velocity filter parameters.
- **paritern**: iterative algorithm.
- **parjsat<*sat*>**: set joint velocity saturation.
- **parjth <*err*>**: set joint position error threshold.
- **park**: park the robot, moving it to the predefined parking position, passing through the exit position. Once the robot is parked the controller will be disarmed.
- **parpark**: set joint position for parked reset.
- **parseteor <*number*><*value*>**: set joint *number* new end of race to *value* (type: H = race high limit, L = race low limit).
- **partaskprec**: change precision-rate task gain and saturation parameters (task: e = end effector).
- **partaskset**: change set-rate task gain, saturation, s1 and s2 parameters (task: m = manipulability, j = joint limits).

- **partasksvd**: change the Task threshold and lambda parameters (task: e = end effector m = manipulability, j = joint limits) (parameter: t = tikhonov, p = pseudoinverse, w = w-matrix).
- **parvfgain**: Cartesian virtual frame maximum gain.
- **parvtime <*jvalue*><*cvalue*>**: set two *values*, both in milliseconds, respectively the Joint and the Cartesian velocity timeouts. With the following command the joint timeout is set to 1 second and the Cartesian one to 20 ms:

  ```
  parvtime 1000 20
  ```

- **showpar**: show the current arm parameters.

3.1.2 Cartesian Helper

With the console commands **cmove**, **cmoveall**, **cvel** and **cvelall** the controller will always move the end-effector frame with respect to the base frame in the Cartesian space. There is an alternative set of commands (**chmove**, **chmoveall**, **chvel**, **chvelall**) that can act in a different way, depending on *Cartesian Helper* configuration [14]. To modify it the commands **chtrasl** and **chrot** have to be used, stating three parameters: the first one sets the frame where the movement is applied (b: base, g:goal), the second one sets the frame with respect to which references are projected (b: base, g:goal), the last one sets references as absolute or relative (a: absolute, r: relative) [14]. The default settings used by the set of commands that don't use the Cartesian Helper correspond to:

```
chtrasl g b a
chrot g b a
```

If, for example, there is the need to move the end-effector (goal) frame with respect to the actual end-effector pose, using the Cartesian Helper set of commands after the following once will produce the desired effect (these settings will be maintained until the manipulator is switched on):

```
chtrasl g g r
chrot g g r
```

3.2 The PowerBall layer

The *PowerBall (PB)* layer, which is composed by a set of classes and functions based on or inherited by the lower levels of abstraction, is the only part of the framework modifiable by the developer. This layer expands the finite state machine (figure 3.4) and adds some new commands.

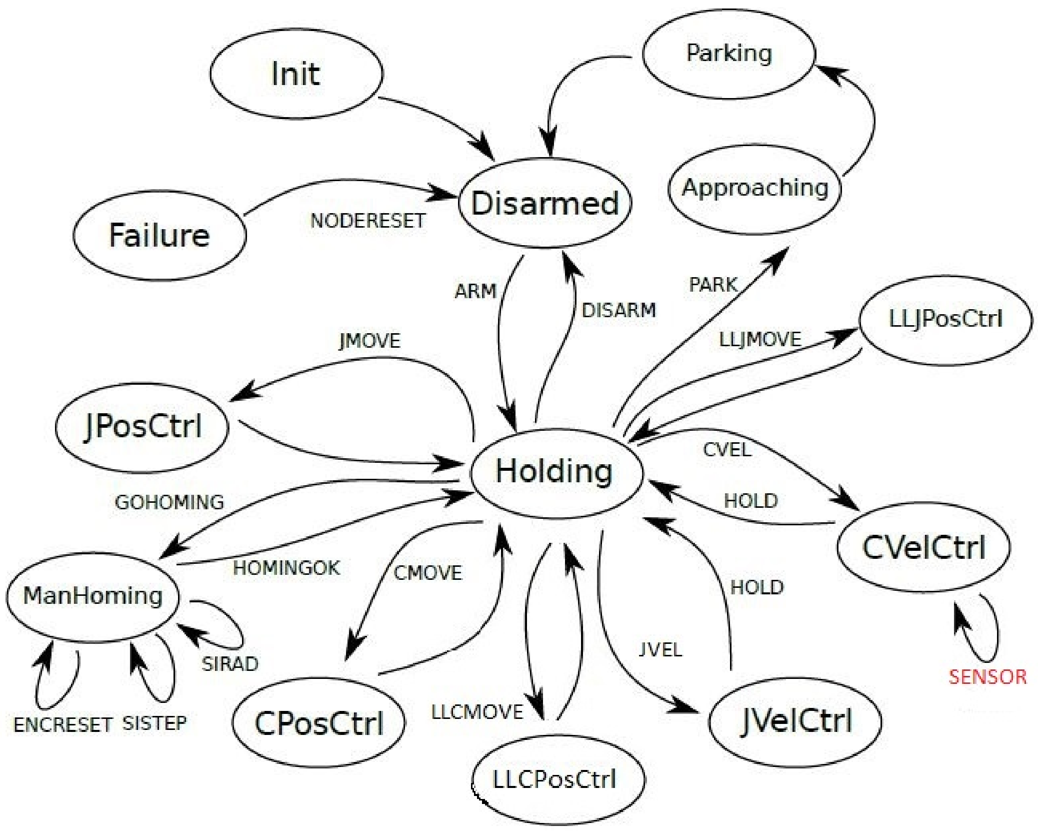


Figure 3.4: Powerball Finite State Machines

The **Low Level Joint Position Control** and **Low Level Cartesian Position Control** states are essentially equal to Joint and Cartesian Position control, with the only difference that here internal controllers are used. These states are associated with the console commands **llcmove**, **llchmove**, **llcmoveall**, **llchmoveall**, **lljmove** and **lljmoveall**, that work exactly the same as the corresponding counterparts of the framework.

Another really useful state is **Manual Homing**, whose purpose is to reset the encoders if an error regarding them occurs, that can happen for example if the arm is manually forced to change position, or sometimes even for a CAN error. The operation needed to correctly recalibrate the joints is the following:

1. Enter the Manual Homing state in with the command **gohoming**. In some cases it's the framework itself that forces the controller into this state.

2. Center all the joints in the zero position (figure 3.3) using the commands:

   - **sirad <*number*> <*inc*> <*speed*>**: moves the robot in the joint space and needs three arguments: *number*, that represents the joint that is wanted to be moved, *inc* is the increment in radians, that can be positive or negative, and also the *speed*, in rad/s, has to be specified (must be positive). Here follows an example to move joint 1 of -0.5 rad at a speed of 0.1 rad:

     ```
     sirad 1 -0.5 0.1
     ```

   - **sistep <*number*> <*inc*> <*speed*>**: moves the robot in the joint space and needs three arguments: *number*, that represents the joint that is wanted to be moved, *inc* is the increment in steps (can be positive or negative, it's a natural number) and also the *speed*, in mdeg/s, has to be specified (must be positive). Here follows an example to move joint 1 of -20 steps at a speed of 1 deg/s:

     ```
     sistep 1 -20 1000
     ```

3. Reset each encoder with the command **encreset <*number*>**:

   ```
   encreset 1
   encreset 2
   encreset 3
   encreset 4
   encreset 5
   encreset 6
   ```

4. End the procedure with the command **homingdone**

The last command added to the existing ones is **sensor**, which activates the compliant control algorithm, that will be deeply described in section 5.3.

3.2.1 Configuration File and Initialization Matrix

The configuration file *powerball.conf* contains a number of parameters that can be set permanently, and will become the default values once the workframe will be switched on. If the developer wants to change these parameters, not only the configuration file has to be modified, but the framework needs to be re-built [14].

Joints

- **velocity_timeout**: indicates how much time (nanoseconds), in Joint Velocity Control, the controller will keep the specified reference, before stopping the manipulator.
- **velocity_saturation**: joint speed upper limit (rad/s).
- **gain**: gain of the control loop. **enable_filter**: enable for first order joint reference filtering.
- **park_position** and **exit_position**: joint positions for parking and exiting from parking (rad).
- **joints_limits_high** and **joints_limits_low**: upper and lower limit positions for the joints (rad). It's essential to set these values once an external module/gripper is mounter on the arm to avoid collisions.

Cartesian

- **velocity_timeout**: indicates how much time (nanoseconds), in Cartesian Velocity Control, the controller will keep the specified reference, before stopping the manipulator.
- **linear_velocity_saturation** and **angular_velocity_saturation**: Cartesian linear and angular speed upper limit, respectively in m/s and rad/s.
- **linear_position_error_threshold**: Cartesian linear position error threshold (in meters) under which the controller will automatically switch from the Cartesian Position Control state to the Holding one.
- **angular_position_error_threshold**: Cartesian angular position error threshold (in radians) under which the controller will automatically switch from the Cartesian Position Control state to the Holding one.
- **algorithm**: active Cartesian algorithm: Jacobian, Iterative or Task-Based.
- **gain**: gain of the control loop. **enable_filter**: enable for first order joint reference filtering.
- **virtual_frame_gain**: gain of the filtering Cartesian virtual frame loop.
- **enable_filter**: enable for the filtering Cartesian virtual frame loop.

Tasks and powerball

- **disable_manipulability**: flag for disabling the manipulability task.
- **qdotmax**: maximum velocity requested for the arm for a single task.
- **pr_ee_approach**: task that controls the end effector goal tracking.
- **sr_joint_limits**: task in charge of the avoidance of joint limits.
- **sr_manipulability**: task in charge of avoiding arm configurations where the arm agility is compromised.
- **showStatusWordChanges**: enable flag for joints status words printing on change.

The **Initialization Matrix** (*InitMatrix()*) can be set in the *PBModel.cc* file, and it is composed by the six roto-translation matrices, that for each joint use the DH parameters, in the standard form [4], reported in table 3.3:

$$\begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Joint Number	θ	d	a	α
1	0	0.205m	0	-$\pi/2$
2	-$\pi/2$	0	0.350m	π
3	-$\pi/2$	0	0	-$\pi/2$
4	0	0.305m	0	$\pi/2$
5	0	0	0	-$\pi/2$
6	0	0.075m	0	0

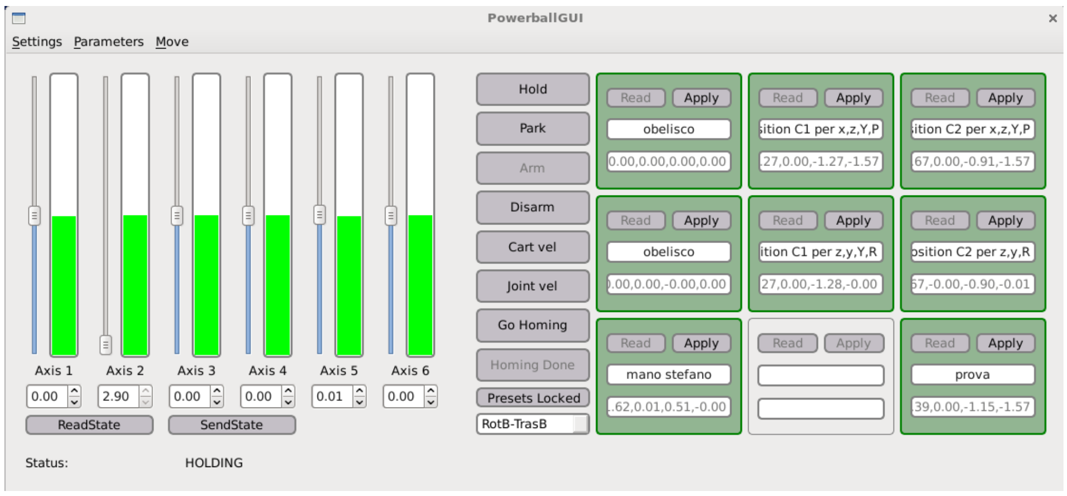
Table 3.3: Nominal DH parameters

In addition to these transformations, there are other two constant ones, ***wTb0_*** and ***eTt_***, whose purpose is to change the pose of the base and end-effector frames respectively. By default these matrices are I_{4x4}, but since the sensor is 8.25 cm tall, it is possible to use the following *eTt_* to translate the end-effector frame up along the z axis:

```
eTt_(1,1) = 1; eTt_(1,2) = 0; eTt_(1,3) = 0; eTt_(1,4) = 0;
eTt_(2,1) = 0; eTt_(2,2) = 1; eTt_(2,3) = 0; eTt_(2,4) = 0;
eTt_(3,1) = 0; eTt_(3,2) = 0; eTt_(3,3) = 1; eTt_(3,4) = 0.0825;
eTt_(4,1) = 0; eTt_(4,2) = 0; eTt_(4,3) = 0; eTt_(4,4) = 1;
```

3.2.2 Powerball Graphical User Interface

In addition to the command that the user can apply through the console terminal, it is possible to control the Schunk Powerball LWA 4P with a dedicated graphical user interface. It is divided in two parts: ***PowerballGUI*** and ***Tragectory Manager***. Starting with the *PowerballGUI* (figure 3.5) on the left there are six sliders, that can be used to move singularly each joint.



0.00 2.90 0.00 0.00 0.01 0.00

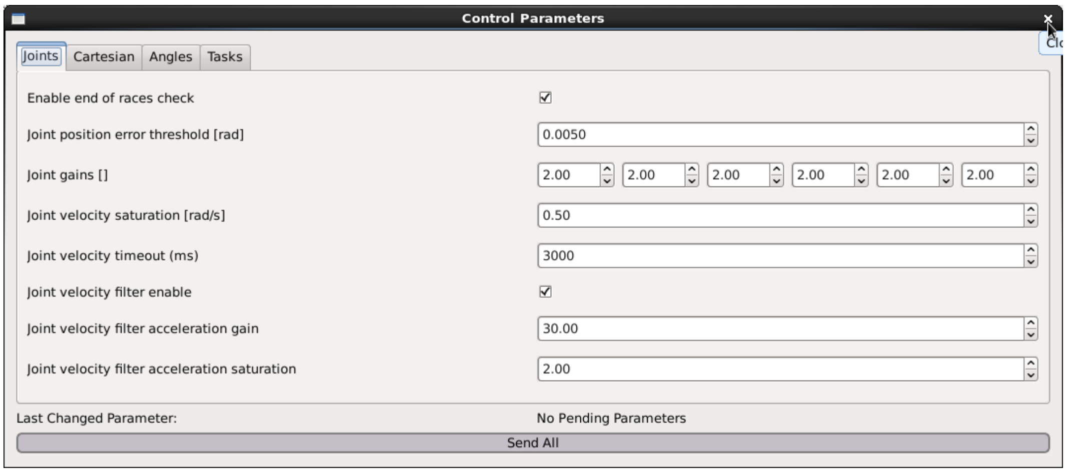
Status: HOLDING

obelisco 0.00,0.00,0.00,0.00
ition C1 per x,z,Y,P 27,0.00,-1.27,-1.57
ition C2 per x,z,Y,P 67,0.00,-0.91,-1.57
obelisco .00,0.00,-0.00,0.00
ition C1 per z,y,Y,R 27,0.00,-1.28,-0.00
osition C2 per z,y,R 57,-0.00,-0.90,-0.01
mano stefano .62,0.01,0.51,-0.00
prova 39,0.00,-1.15,-1.57

Figure 3.5: PowerballGUI

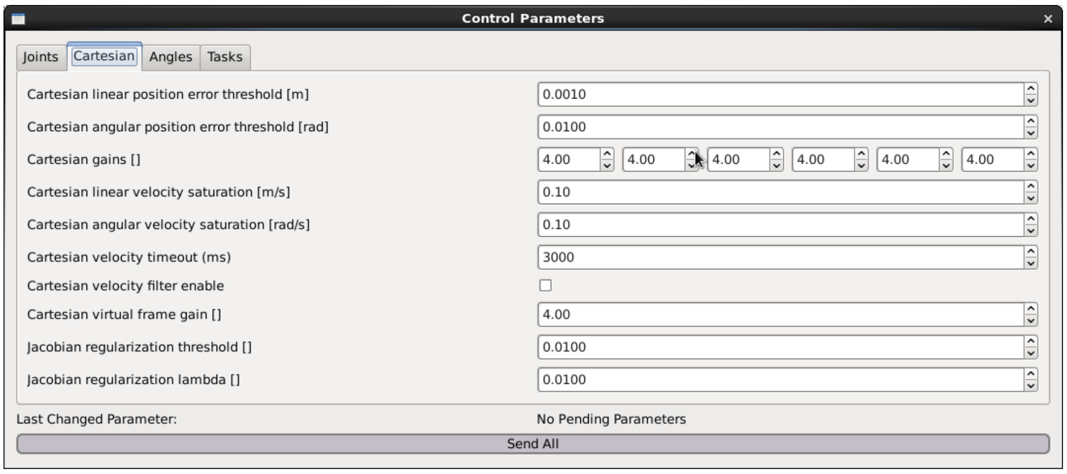
On the right of each slider is placed a colored line to represent at which point of its range the joint is, that is also reported numerically under the sliders. The active state can be red in the left bottom corner under the sliders. To change manually the state of the controller eight buttons are placed in the center of the GUI. Since it is not possible to jump from a particular state to every other one, these buttons will activate and deactivate accordingly to the mapping between states defined in the finite state machine (see figure 3.4). Same thing happens for the functionalities that can't be called in a particular state. For example in figure 3.5 the controller is in the holding state, so this means that the only two operations not available are arming and gohoming. The ninth, smaller button, named `Presets_Locked/Preset_Unlocked` has the ability to unlock the nine presets placed on the right, that can be modified, added or deleted only if this button is manually switched off. The button `Read` saves the current joint position and it is also possible to add or change the name of the presets directly in the tab under the two buttons. Once one `Apply` button is pressed, the end-effector frame will be automatically moved in the saved pose.

Important note: using the `Read` (see section 3.2.2) button near the zero position of the joints, some of the saved joint positions could be corrupted, so, before using the new preset issue the `jpos` command from the console terminal to print all the current joint values to double check that the reading was successful.



Enable end of races check	✓
Joint position error threshold [rad]	0.0050
Joint gains []	2.00 2.00 2.00 2.00 2.00 2.00
Joint velocity saturation [rad/s]	0.50
Joint velocity timeout (ms)	3000
Joint velocity filter enable	✓
Joint velocity filter acceleration gain	30.00
Joint velocity filter acceleration saturation	2.00
Last Changed Parameter:	No Pending Parameters

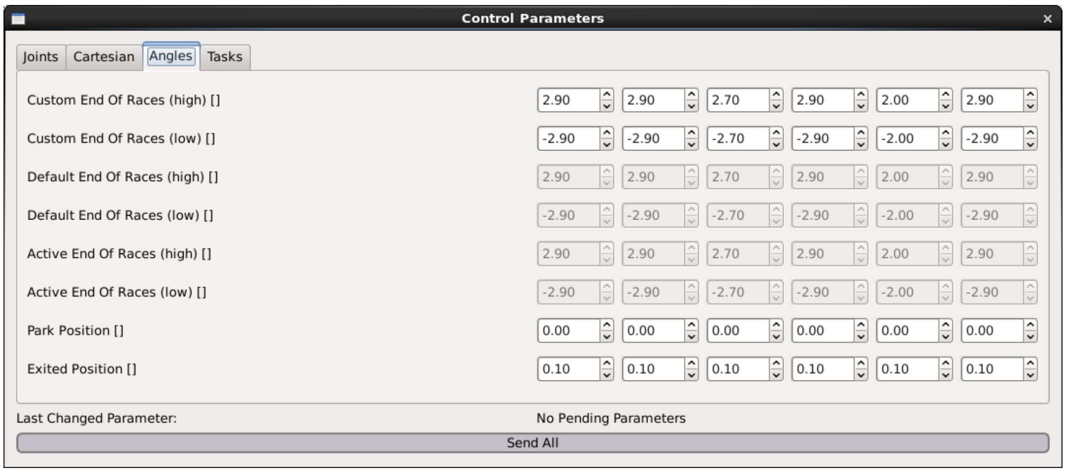
Figure 3.6: Joints control parameters



Cartesian linear position error threshold [m]	0.0010
Cartesian angular position error threshold [rad]	0.0100
Cartesian gains []	4.00 4.00 4.00 4.00 4.00 4.00
Cartesian linear velocity saturation [m/s]	0.10
Cartesian angular velocity saturation [rad/s]	0.10
Cartesian velocity timeout (ms)	3000
Cartesian velocity filter enable	☐
Cartesian virtual frame gain []	4.00
Jacobian regularization threshold []	0.0100
Jacobian regularization lambda []	0.0100
Last Changed Parameter:	No Pending Parameters

Figure 3.7: Cartesian control parameters

On top right corner of the *PowerballGUI* it is possible to set all the constant parameters relative to the arm. By pressing ”p” on the keyboard, the *Control Parameters* page will be opened. With the first two sub-pages, *Joints* and *Cartesian*, it is possible to set error thresholds, control gains, velocity saturation, velocity timeouts and other parameters. Instead the *Angles* page is dedicated to positive and negative end of races, park position and exit position. It is advisable, specially when the sensor or other gadgets are mounted on the arm, to reduce the end of races in order to avoid contacts with the protection case of the arm itself, that will produce problems during the next arming phase (see Appendix A).



Custom End Of Races (high) []	2.90	2.90	2.70	2.90	2.00	2.90
Custom End Of Races (low) []	-2.90	-2.90	-2.70	-2.90	-2.00	-2.90
Default End Of Races (high) []	2.90	2.90	2.70	2.90	2.00	2.90
Default End Of Races (low) []	-2.90	-2.90	-2.70	-2.90	-2.00	-2.90
Active End Of Races (high) []	2.90	2.90	2.70	2.90	2.00	2.90
Active End Of Races (low) []	-2.90	-2.90	-2.70	-2.90	-2.00	-2.90
Park Position []	0.00	0.00	0.00	0.00	0.00	0.00
Exited Position []	0.10	0.10	0.10	0.10	0.10	0.10

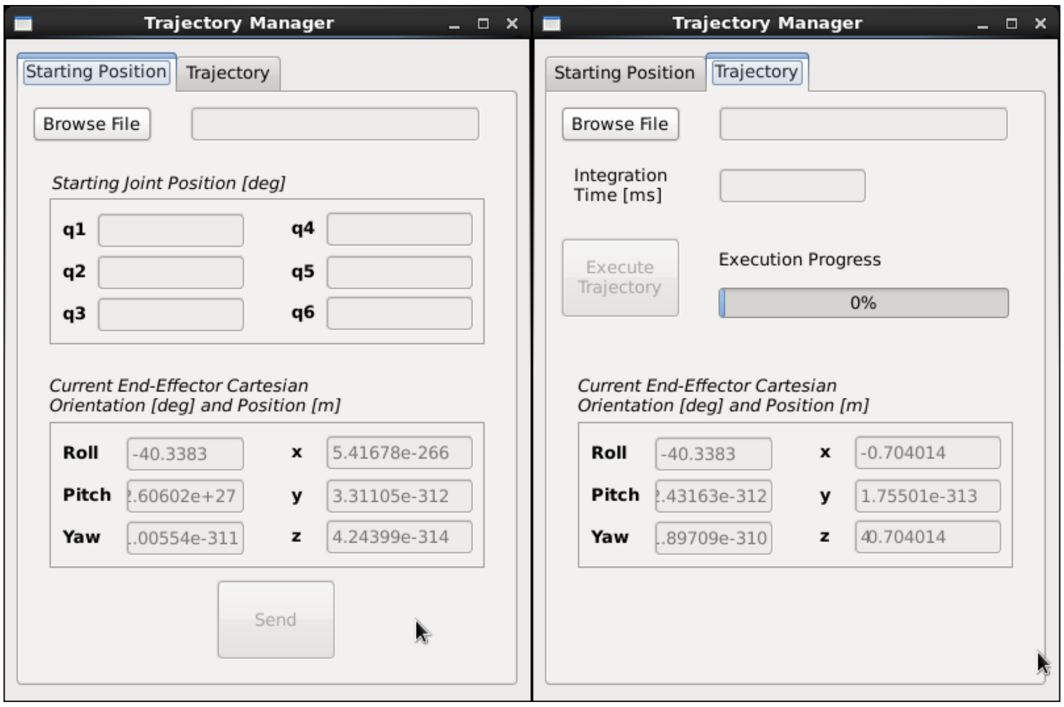
Last Changed Parameter: No Pending Parameters

Figure 3.8: Angles control parameters

The *Trajectory Manager* (figure 3.9) shows the current end-effector pose and can be used for uploading a desired trajectory. It must be contained in formatted `.dat` file, where in the left column are reported the time instants and in the right one the corresponding velocities, where `Vx`, `Vy` and `Vz` are the linear velocities, and `Rot_Petal1`, `Rot_Petal2` and `Rot_Petal3` correspond to ω_x, ω_y and ω_z. To properly separate each velocity the last data sample must be at 0.01 seconds from the previous one and its value must be zero, then `ENDATA` closes the sequence:

```
XYDATA, REQ/70000158 Vx
0.0000000000e+000     3.0000000000e-002
0.1000000000e+000     3.0000000000e-002
0.2000000000e+000     3.0000000000e-002
0.2100000000e+000     0.0000000000e+000
ENDATA

XYDATA, REQ/70000159 Vy
0.0000000000e+000     5.0000000000e-002
0.1000000000e+000     2.0000000000e-002
0.2000000000e+000     4.0000000000e-002
0.2100000000e+000     0.0000000000e+000
ENDATA

XYDATA, REQ/70000160 Vz
0.0000000000e+000     9.0000000000e-002
0.1000000000e+000     8.0000000000e-002
0.2000000000e+000    -3.0000000000e-002
0.2100000000e+000     0.0000000000e+000
ENDATA
```

```
XYDATA, REQ/70000161 Rot_Petal1
0.0000000000e+000     1.0000000000e-002
0.1000000000e+000     0.0000000000e-002
0.2000000000e+000     1.0000000000e-002
0.2100000000e+000     0.0000000000e+000
ENDATA

XYDATA, REQ/70000162 Rot_Petal2
0.0000000000e+000     4.0000000000e-002
0.1000000000e+000     4.0000000000e-002
0.2000000000e+000     4.0000000000e-002
0.2100000000e+000     0.0000000000e+000
ENDATA

XYDATA, REQ/70000163 Rot_Petal3
0.0000000000e+000     5.0000000000e-002
0.1000000000e+000    -5.0000000000e-002
0.2000000000e+000     5.0000000000e-002
0.2100000000e+000     0.0000000000e+000
ENDATA
```



Current End-Effector Cartesian Orientation [deg] and Position [m]

Roll	-40.3383	x	5.41678e-266
Pitch	.60602e+27	y	3.31105e-312
Yaw	.00554e-311	z	4.24399e-314

Current End-Effector Cartesian Orientation [deg] and Position [m]

Roll	-40.3383	x	-0.704014
Pitch	.43163e-312	y	1.75501e-313
Yaw	.89709e-310	z	40.704014

Figure 3.9: Trajectory Manager

Chapter 4

The CAN protocol

Controller Area Network - CAN - is a communication protocol developed by R. Bosch GmbH at the beginning of the 1980s as a working method for enabling robust serial communication, aimed for the automotive area. In 1990 Mercedes-Benz was the first manufacturer to use the protocol in one of their flagship model, the S-class. In 1993, it became an international standard with ISO 11898, CAN 2.0A and 2.0B were also released, and in 1994 other CAN-related higher level protocols have been standardized, such as CANopen and DeviceNet [6]. In 1997, 24 million CAN interfaces were produced in 1 year; 2 years later there were already more than three times as many, and today almost every road vehicle uses this protocol [12]. The reason why CAN became so popular is that it provides an inexpensive (because it can reduce a lot the amount of cables needed) and a durable way to make the numerous micro-controllers present in a car to communicate with one another. Today the CAN networks are also used in space and aerospace applications, robotics, railed transportation, hospitals and even coffee machines [6].

4.1 ISO 11898

Since CAN has evolved and has become more and more complicated in the past 50 years, ISO 11898 had to expand and include all the details of the protocol. Today ISO 11898 is divided in five parts:

- **ISO 11898-1** Road vehicles—Controller area network (CAN) —Part 1: Data link layer and physical signalling
- **ISO 11898-2** Road vehicles—Controller area network (CAN) —Part 2: High-speed medium access unit
- **ISO 11898-3** Road vehicles—Controller area network (CAN)—Part 3: Lowspeed, fault-tolerant, medium-dependent interface

- **ISO 11898-4** Road vehicles—Controller area network (CAN)—Part 4: Timetriggered communication
- **ISO 11898-5** Road vehicles—Controller area network (CAN)—Part 5: High-speed medium access unit with low-power mode

Without diving too much into the standard, a simple representation of the layers that form this communication structure are the ones shown in figure 4.1.

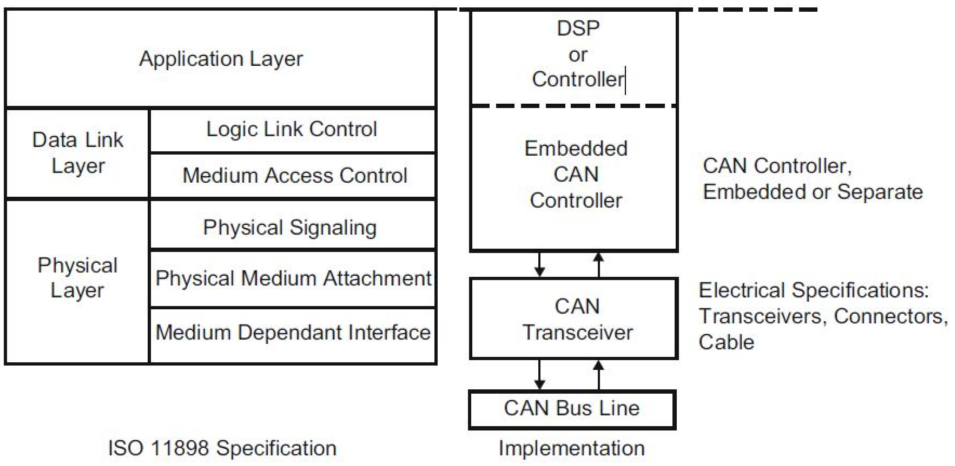


Figure 4.1: ISO 11898 standard architecture for CAN networks

The **Application layer** provides high level communication functions that can be implemented by a software developer or handled by a higher level protocol such as:

- **CAL** (**C**AN **A**pplication **L**ayer): originally developed by Philips Medical Systems, is an application-independent layer that is now maintained by the *CAN in automation* (CiA) user group.
- **CANopen**: built on top of CAL, using some of its services and communication protocols. With this protocol every node in the network is associated to an Object Dictionary (OD) that contains all the parameters that describe the device and its behavior. Also CANopen is now maintained by the *CAN in automation* (CiA) user group.
- **DeviceNet**: developed by the American company Allen-Bradley (now owned by Rockwell Automation), adapts the technology from the Common Industrial Protocol and takes advantage of CAN, making it low-cost and robust compared to the traditional RS-485 based protocols.

Then, the **Datalink layer** is responsible to transfer the messages from a node to all the other ones. It's divided into Logic Link Control layer (LLC) and Medium Access Control layer (MAC) and it handles bit stuffing and error control, waiting for acknowledgement from the receivers after a message is sent [6].

Figure 4.2 shows how nodes are connected in a typical CAN network. The **Physical layer** implements physical signaling, bit encoding and decoding, bit transmitting and synchronization [6].

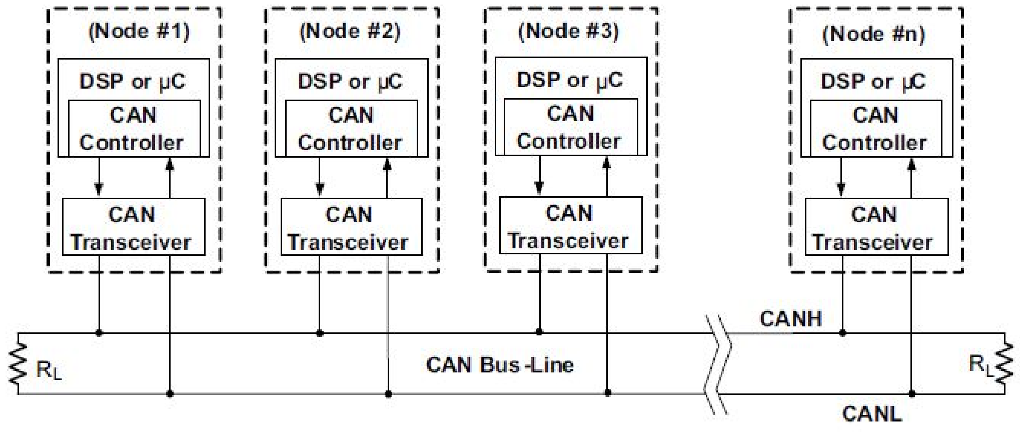


Figure 4.2: Example of the physical connection in a high-speed CAN network

Different kinds of physical layers are used to satisfy the system specification both from a cost or performance point of view. The most used are:

- High-speed CAN hardware: it's the most common physical layer, it uses two wires to allow communication at 1 Mbit/s rate. It is mostly used for anti-lock brake systems, engine control modules and emission systems, and it's also known as CAN-C (ISO 11898-2).

- Low-speed/fault-tolerant CAN hardware: also known as CAN-B (ISO 11898-3), it uses again two wires for the communication, but up to 1 Mbit/s transfer rate. In the automotive field is primarily used for comfort devices.

- Single-wire CAN hardware: using just a single wire for the communication, the transfer rate is limited to 33.3 kbit/s (88.3 kbit/s in high-speed mode). Also known as CAN-A, within an automobile is used just for those devices that do not have any performance requirement, such as mirror adjusters.

4.2 Main features

By being such a robust, fast, safe and widely used protocol, it is inevitably complex. The communication is asynchronous, half duplex (usually with differential signaling), up to 1 Mbit/s. The number of nodes is virtually unlimited, they don't have an address, but filter the data on the bus to determine if it's useful or not, and a master-slave designation is not used. Logic values are "dominant" (low - 0) or "recessive" (high - 1), and dominant overrides recessive, so any node can start a message, so an arbitration process is applied, without any loss of time or data. It's also important to mention that high level of data security since a node that, after error checking, recognise that it's faulty, can disconnect itself from the network [10].

Since all the nodes are equal, it's not possible to simply send "only" the data on the bus, so the communication is based on **messages** or **frames** that allows to carry much more information. Figure 4.3 shows how typically a CAN-message is organized for CAN 2.0A (standard) and can 2.0B (extended).

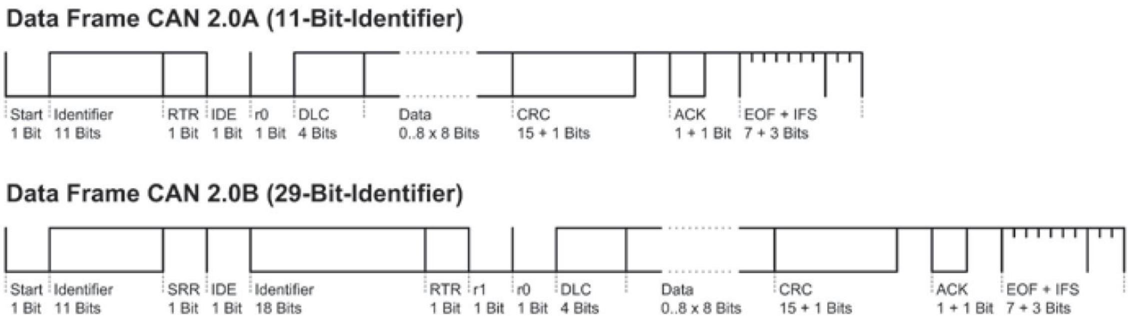


Figure 4.3: Standard structure of a CAN-message

Considering just the 11-Bit-Identifier frame, which is the one used for the Schunk FTM115 force-torque sensor, the message is divided in the following parts [12]:

- **Start of Frame (SOF) = 1 bit (low, dominant)**: always used to start a frame, the falling edge synchronizes all network nodes.
- **Arbitration Field = 12 bit**:ID and RTR together compose the Arbitration Field. Each message has its own **ID = 11 bit** that basically is the "name" of the message (CAN 2.0B has a 29 bit ID), the lower is the identifier the higher is the priority. **Remote Transmission Request (RTR) = 1 bit** is the last bit of the arbitration field. If RTR is high=recessive it means that the message is asking for data (*data request frame*), otherwise if RTR is low the message contains the data itself (*data frame*) or doesn't have to contain data because it only triggers some operations that don't need it. It is important to note that these two messages have the same ID but different RTR so aren't the same.

- **Control Field = 6 bit** : is composed by the **Identifier Extension Flag (IDE)** bit that indicates that the ID is completed, the r0 bit that is reserved and the **Data Length Code (DLC)** 4 bit that indicates how many bytes of data the message is carrying.
- **Data Field = 0–8 bytes of data**: contains the actual data of the message.
- **Cyclic Redundancy Check (CRC) Field = 16 bits**: the first 15 bits (CRC sequence) are used only for fault detection, adding redundant check bits at the transmission end, then this bits are recomputed at the receiver end and tested again the received bits, if there is a misalignment a CRC error occurred [9]. The CRC field is closed by that last bit, called CRC delimiter (high).
- **Acknowledge Field = 2 bits**: All the nodes that have recognized the message as correct will send a dominant level in the ACK slot, if no node send the low level an ACK error occurred. The acknowledge field is closed by the ACK delimiter (high).
- **End of Frame (EOF) = 7 bit**: indicates the end of the data frame (usually all recessive).
- **Inter Frame Space (IFS) = 3 bit (high)**: separates the frame from the following one. The time for this operation is used inside the node to transfer the message from the controller to a receive buffer or to transfer a message from a transmit buffer to the controller.

It's clear that inside the message structure a good part is dedicated to **error detection**, another important feature of the CAN protocol, with the CRC field and the ACK field. Another way in which an error active node or the transmit one can report an error is by transmitting an error frame (usually six consecutive low level bits), a particular message that violates the rules of bit stuffing and frame format, causing all the other nodes to send an error frame as well [2], then bus activity returns to normal. Instead also an error passive node can indicates that have detected an error by sending an error-passive flag (usually 14 recessive bits), so if the fault is not detected by an error active node or the transmit one the message will continue transmission [9].

Generally hundreds if not thousands of nodes are connected and try to transmit messages in a single CAN network, and since there is no master-slave designation, a **bit-wise arbitration** is used to give priority to e frame instead of another one. Figure 4.4 shows how it works.

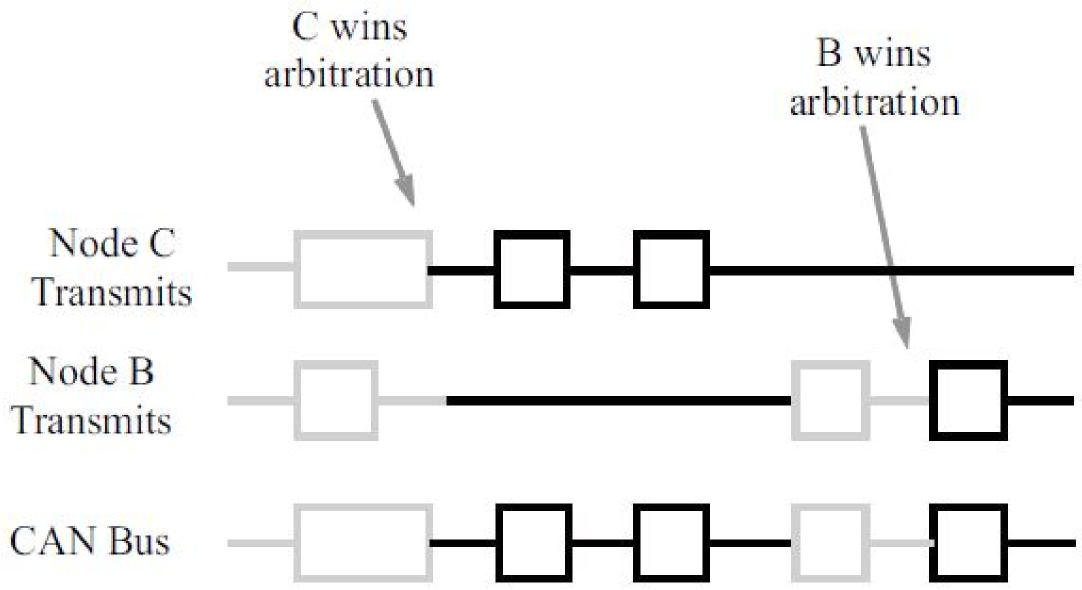


Figure 4.4: Arbitration example scheme on a CAN bus

Node B and Node C start transmitting on the CAN bus at the same moment but after some identical bits, B tries to put on the bus a recessive (high) bit while C transmits a dominant (low) one. So B loses arbitration and stops transmitting, while C finishes its message and after that B wins arbitration again, completing its frame. This functionality is part of the ISO 11898 physical layer, which means that it is contained entirely within the CAN controller and is completely transparent to a CAN user [2].

4.3 Schunk FTM 115 CAN Bus Interface

The robotic arm Schunk LWA 4P implements two separate CAN networks, one dedicated only for the six motors, that uses an higher level protocol (CANopen CiA DS402:IEC61800-7-201) [17], and the other one for all the modules compatible with the robot, as the Schunk FTM 115 force torque sensor (figure 4.5), which instead has to be managed at a lower level.

Table 4.1: Characteristics of the FTM 115

Range of measurement F_x, F_y [N]	$\pm$ 580
Range of measurement F_z [N]	$\pm$ 1160
Range of measurement M_x, M_y, M_z [Nm]	$\pm$ 20
Resolution F_x, F_y, F_z [N]	$\pm$ 1/4
Resolution M_x, M_y [Nm]	$\pm$ 1/188
Resolution M_z [Nm]	$\pm$ 1/376
Weight [Kg]	1.0

Continued on next page

Table 4.1 – *Continued from previous page*

Power Supply [V]	24
Bus Interface	CAN

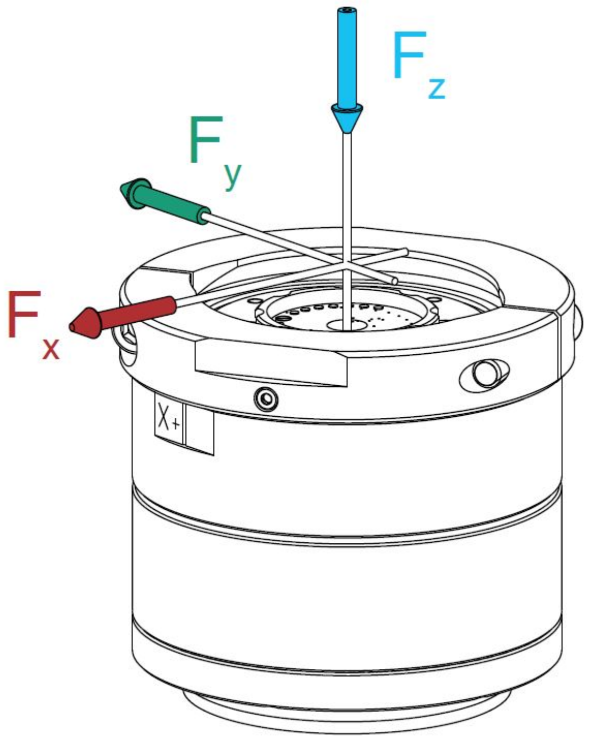


Figure 4.5: Schunk's force torque sensor FTM 115

The F/T sensor integrates the custom designed 9105-TW-MINI45 transducer with through-hole and a F/T-to-CAN interface board (NETCANOEM) [7], produced by ATI Industrial Automation.

As mentioned earlier, the frame format can be quite complex, but with this interface the user has to pay attention only to the identifier and the data (always in big-endian format). The identifier (11 bits) is split into two parts: the first 7 bits are called ***Base Identifier*** (default = 0x20) that the developer can change. Every CAN message for the F/T sensor will start with it. The last 4 bits are called ***OPcode*** and each different function implemented by the sensor has its own. The message can contain data or not, depending on the function.

Other significant elements are ***Active Calibration***, ***Counts per Unit***, ***Calibration Matrix***, ***Strain Gauge Data*** and ***Status***. The first three have to be computed just once, then they will be used to transform the strain gauge data into forces and torques. The status has to be periodically checked in order to guarantee a safe use of the hardware and verify that the communication is not corrupted.

The CAN bus interface allows the user to [7]:

- Determine which calibration is active and select the one to be active.
- Read the calibration matrix.
- Select the baud rate (up to 2 Mbit/s, default = 250 Kbit/s).
- Select the base identifier.
- Reset the NETCANOEM interface.
- Read the firmware version of the NETCANOEM.
- Read FT calibration serial number
- Request strain gage data and status information.

4.3.1 Functions and Status

As previously said the ID for each function is composed by the base identifier and the OPcode, moreover some functions don't need any data or even a response. To be more clear an example will come in handy (supposing to use the default base ID = 0x20): if the SG (strain gauge) data to be red, function that correspond to the OPcode = 0x0 and doesn't need any data, the user must compose and send a message with ID = 0x200 and will receive a response. If he wants to reset the NETCANOEM, OPcode = 0xC and again doesn't need any data, the message will contain just the ID = 0x20C but in this case no response from the sensor will be generated. Here follows a description of all the functions from the user point of view, or rather what the programmer has to send to the sensor to activate a certain functionality and what he will eventually receive back.

Read SG data

OPcode: 0x0
Data: None
Two response packets are sent: One with the opcode set to 0x0, which contains the two byte status, followed by the two byte values (signed integers) for sg0, sg2, and sg4 (total of eight bytes), and another packet with the opcode 0x1, which contains the three two byte values sg1, sg3, and sg5 (total of six bytes). It will be necessary to reorder the strain gauges (sg0, sg1, sg2, sg3, sg4, sg5) before performing the matrix multiplication. Alternatively, you can rearrange the order of the columns in the matrix to match the ordering in this response [7] (sg0, sg2, sg4, sg1, sg3, sg5).

Read Matrix

OPcode: 0x2

Data: 1 byte indicating axis row to read (0=Fx, 1=Fy, 2=Fz, 3=Tx, 4=Ty, 5=Tz)

Three response packets (8 bytes each) are sent. All matrix coefficients are in 4 byte floating point format:

Opcode 0x2 containing the SG0 and SG1 coefficients for the requested axis.

Opcode 0x3 containing the SG2 and SG3 coefficients.

Opcode 0x4 containing the SG4 and SG5 coefficients [7].

Read F/T Serial Number

OPcode: 0x5

Data: None

One response packet with the opcode 0x5, with 8 data bytes which contain an ASCII string representing the F/T serial number [7].

Set Active Calibration

OPcode: 0x6

Data: 1 byte indicating index of calibration to use, possible index values are 0 to 15

One response packet with opcode 0x6 with one data byte echoing the selected calibration index [7].

Read Counts Per Unit

OPcode: 0x7

Data: None

One response packet with opcode 0x7, with 8 bytes of data. The first 4 bytes are the counts per force, followed by the 4 byte counts per torque. Both count values are integers [7].

Read Unit Codes

OPcode: 0x8

Data: None

Table 4.2: Force unit and unit codes

Force Unit	Force Unit Code
lbf	1
N	2
klbf	3

Continued on next page

Table 4.2 – *Continued from previous page*

kN	4
kgf	5
gf	6

Table 4.3: Torque unit and unit codes

Torque Unit	Torque Unit Code
lbf-in	1
lbf-ft	2
N-m	3
N-mm	4
kgf-cm	5
kN-m	6

One response packet with the opcode 0x8, with two bytes of data. The first byte is the force unit code (table 4.2), the second one is the torque unit code (table 4.3) [7].

Reset

OPcode: 0xC

Data: None

The interface will be reset but no response is generated by the sensor [7].

Set base identifier (first 7 bits)

OPcode: 0xD

Data: 1 byte indicating the first 7 bits to use. The 7 bits are right-justified in the data byte.

One response packet with opcode 0xD with no data. The command will take effect at the next power-up [7].

Set baud rate

OPcode: 0xE

Data: 1 byte indicating the divisor code used to set the baud rate. A base rate of 2 mbps is divided by this value + 1 to generate the effective baud rate. For example: value=3 $\rightarrow$ divisor=4 $\rightarrow$ baud rate = 500kbps.

One response packet with opcode 0xE with no data. The command will take effect at the next power-up [7].

Read Firmware version

OPcode: 0xE
Data: None
One response packet with opcode 0xF with 4 bytes of data. The first byte is the major version, the second byte is the minor version, and the next two bytes are the build number [7].

The response message with OPcode 0x0 not only contains data relative to forces and torques, but also 16 bits representing the **Status**. If there isn't any error, every bit will be set to 0, in the opposite case it's possible to translate the error considering the position of the bit that lighted up. In case of a critical error, the NETCANOEM will not stop transmitting strain gauge data (see table 4.4 [7]). The sensor can't solve the problems reported with the two status byte, so is responsibility of the user's application to handle that properly.

Table 4.4: NETCANOEM Status Register

Bit	Name	Type	Remark / Recommended error handling
0	Watchdog Reset		Can occur after firmware-upgrade; replace NETCANOEM if this happens during normal operation
1	DAC/ADC check result too high	Critical	Analog Acquisition system gets checked once after reset Stop operation - replace NETCANOEM
2	DAC/ADC check result too low	Critical	Analog Acquisition system gets checked once after reset Stop operation - replace NETCANOEM
3	Artificial analog ground out of range	Critical	Stop operation - replace NETCANOEM
4	Power supply too high	Critical	Stop operation - check power supply to NETCANOEM
5	Power supply too low	Critical	Stop operation - check power supply to NETCANOEM
6	Bad active calibration	Critical	Select a valid calibration slot. Checksum of the selected calibration is wrong.
7	EEPROM failure	Critical	Stop operation - No or invalid EEPROM response; Checksum error in EEPROM memory, replace NETCANOEM if this happens during normal operation
8	Configuration Invalid		Checksum error in stored configuration data; NETCANOEM will use default settings

Continued on next page

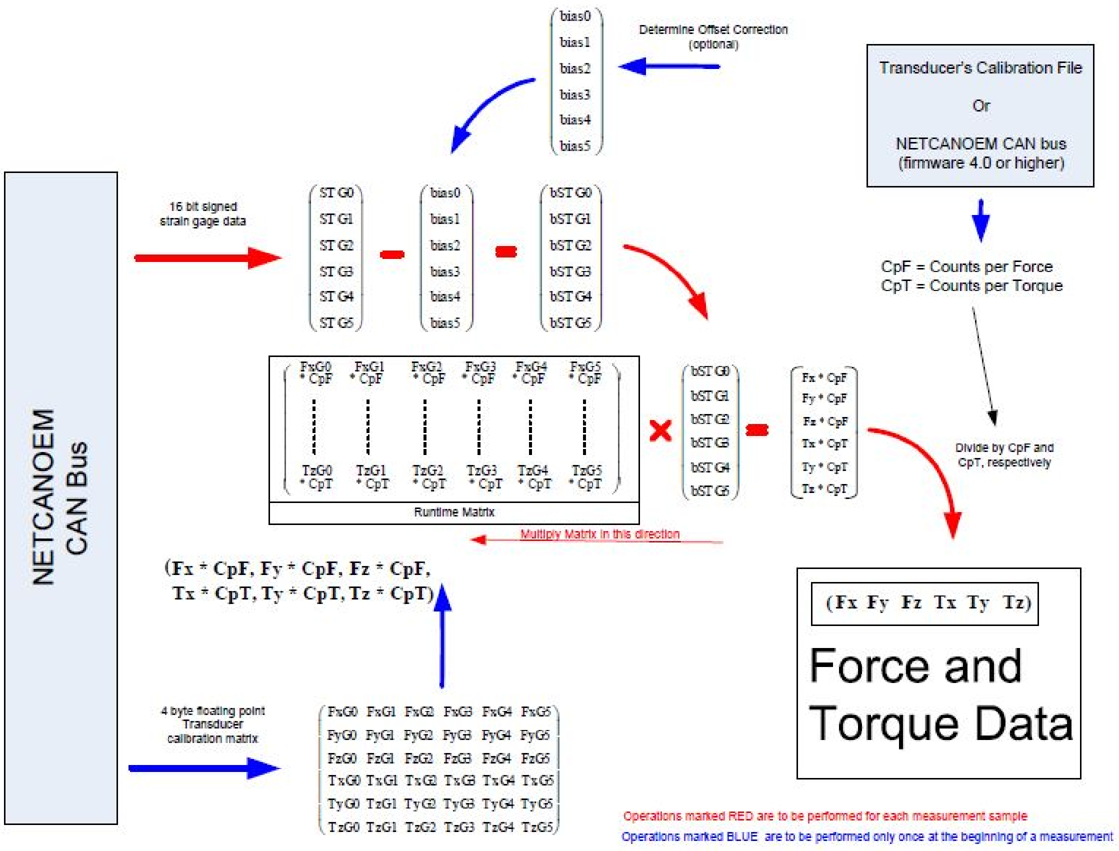
Table 4.4 – *Continued from previous page*

9	Reserved		
10	Reserved		
11	Sensor temperature too high	Critical	Stop operation - Make sure that ambient temperature of the NETCANOEM stays within the specified range
12	Sensor temperature too low	Critical	Stop operation - Make sure that ambient temperature of the NETCANOEM stays within the specified range
13	Reserved		
14	CAN bus error		CAN bus error detected; turns status LED to red
15	Any error causes this bit to turn on		

4.3.2 Data Acquisition and Matrix Calculation

The SG data don't correspond to the real forces and torques so, to correctly compute them, some preliminary operation are needed [7]:

- Select the correct active calibration and verify it. If there is no valid calibration in the selected slot, then bit 6 "Bad active calibration" in the status register will be on. After a reset the calibration slot 0 will be always selected.
- Read the active calibration matrix.
- Read the Counts per Force and Counts per Torques constants.



Determine Offset Correction (optional)
Transducer's Calibration File
Or
NETCANOEM CAN bus (firmware 4.0 or higher)
16 bit signed strain gage data
CpF = Counts per Force
CpT = Counts per Torque
NETCANOEM CAN Bus
Runtime Matrix
Multiply Matrix in this direction
Divide by CpF and CpT, respectively
(Fx * CpF, Fy * CpF, Fz * CpF, Tx * CpT, Ty * CpT, Tz * CpT)
4 byte floating point Transducer calibration matrix
(Fx Fy Fz Tx Ty Tz)
Force and Torque Data
Operations marked RED are to be performed for each measurement sample
Operations marked BLUE are to be performed only once at the beginning of a measurement

Figure 4.6: F/T Matrix Calculations

At this point it is possible to request iteratively the SG data and transform them into real F/T data using the scheme shown in figure 4.6 [7]. A critical operation is the offset correction, since it could be quite significant, which means that is important to perform it at each data acquisition, in order to have a precise reading. Another possible case to consider is when one of the strain gauge data is saturated (nominally -32768 or +32767). For example, this can be caused by a broken transducer cable. In this case, the sensor data is not usable anymore. It is the responsibility of the receiving application to handle this error situation appropriately (e.g., request maintenance) [7].

Chapter 5

Control Strategy

This chapter is dedicated to the description of the control strategy adopted to successfully test the docking mechanism described in Chapter 2, starting with a brief overview of the four phases to be performed in the laboratory to simulate the docking maneuver. Then it follows the description of classical impedance control algorithm, and the one of the approach chosen for this particular application. The code relative to the control strategy finally implemented is reported in appendix B.

5.1 Description of the experiment

In order to demonstrate the validity of the docking mechanism, the Powerball LWA 4P, in conjunction with the FTM115 force/torque sensor, can be used in order to simulate the rendez-vous and docking maneuvers, following a precise procedure divided in 4 main steps.

The chaser (servicer) is mounted on the sensor with a 3D printed connection, fundamental to have a break-point between the arm and the mechanism, so both are preserved in case of failure or unwanted behaviours. As shown in figure 5.1, the axis of the target (customer) and the one of the chaser are parallel, but not coincident, to simulate a real approach and deployment phase. At this point the arm moves in the Cartesian space along the positive z-axis and, once the probe and the drogue are in contact, the control system of the chaser will shift the active half in order to reduce the contact forces, until the tip of the probe overpasses the female socket of the passive half.

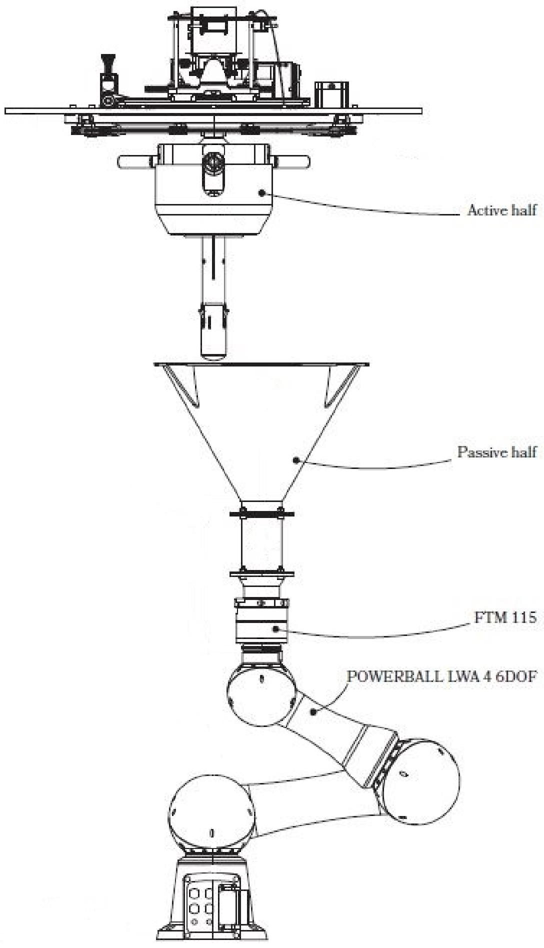


Figure 5.1: Initial position

The customer and the moving part of the servicer are now mechanically connected, but still able to move on a horizontal plane with respect to the support plate of the chaser, so the soft docking procedure is now completed.

Now the goal is to re-align the moving part with the fixed one to proceed with the hard docking. A set of linear actuators pulls the coupled elements towards the center until the initial position of the active half is restored. During this process, and in the following operations, the arm simply has to follow the movement imposed by the mechanism, or, in other words, has to be compliant.

Finally the system is ready to complete the docking: the probe is retracted until the base of the drogue touches the support plate and , with the aim of securing the two parts, three radial hooks makes the two elements rigidly coupled.

5.2 Impedance control: state of the art

Since the manipulator must interact with the environment for this application, a suitable control strategy is **Impedance Control**, where the robot end-effector is usually asked to render particular forced mass, spring, and damper system properties [15].

$$m\ddot{x} + b\dot{x} + kx = f_{ext} \tag{5.1}$$

Referring to the equation (5.1), impedance is defined by the transfer function from position perturbation to forces, $Z(s) = F(s)/X(s)$ [15].

The starting point is the dynamic model of an n-joint manipulator, derived with the *Lagrange formulation* equation (5.2), where q are the generalized coordinates of the joints of the manipulator:

$$B(q)\ddot{q} + C(q,\dot{q})\dot{q} + F_v\dot{q} + F_s sgn(\dot{q}) + g(q) = \tau - J^T(q)h_e \tag{5.2}$$

- $B(q)$ is a configuration dependent matrix whose elements on the main diagonal b_{ii} represent the moment of inertia at joint i axis,in the current arm configuration, when the other joints are blocked. The coefficients b_{ij} account for the effect of acceleration of joint j on joint i.

- $C(q,\dot{q})$ is a configuration and velocity dependent matrix whose coefficients represent the centrifugal effect and the Coriolis effect.

- F_v is the nxn diagonal matrix of viscous friction coefficients. F_s is the nxn diagonal matrix of static friction coefficients. Since $F_s sgn(\dot{q})$ corresponds to a simplified model of static friction torques, usually hard to model and not so significant in this study case, later on the non conservative forces will be grouped in a single element, $F_v\dot{q} + F_s sgn(\dot{q}) = F\dot{q}$.

- $g(q)$ is the $nx1$ configuration dependent vector, whose terms g_i represent the moment generated at joint i axis by the presence of gravity.

- τ are the actuation torques and $-J^T(q)h_e$ are the torques at each joint, where h_e is the $6x1$ vector of forces and moments exerted by the end-effector on the environment if there is contact between the two.

Considering a simplified model, in which all the non-linear terms are grouped in a single one (equation (5.3) and (5.4)), the idea is to find a control vector u capable of realizing an input/output relationship of linear type. This is guaranteed by the form of the simplified dynamical model in equation (5.3), linear in the control u and has a full-rank matrix $B(q)$, invertible for any manipulator configuration.

$$B(q)\ddot{q} + n(q,\dot{q}) = u \tag{5.3}$$

$$n(q,\dot{q}) = C(q,\dot{q})\dot{q} + F\dot{q} + g(q) \tag{5.4}$$

If now the control u is taken as a function of the manipulator state, leads to:

$$u = B(q)y + n(q, \dot{q}) \tag{5.5}$$

$$\ddot{q} = y \tag{5.6}$$

This non-linear control law u is also called **Inverse Dynamics Control**, since is based on the computation of manipulator inverse dynamics . The system described by equation (5.5) is, with respect to the new input y (which has to be determined yet), **linear** and **decoupled** [4]. It's now time to define the new stabilizing input y for the linear system and the reference r. so, by choosing:

$$y = -K_P q - K_D \dot{q} + r \tag{5.7}$$

leads, using equation (5.6) and (5.7) to:

$$\ddot{q} + K_D \dot{q} + K_P q = r \tag{5.8}$$

Under the assumption that K_P and K_D are positive definite matrices, is possible to demonstrate that the system is asymptotically stable [4]. To track a desired trajectory $q_d(t)$ the following reference must be chosen, realizing the block scheme in figure 5.2:

$$r = \ddot{q}_d + K_D \dot{q}_d + K_P q_d \tag{5.9}$$

The dynamics of the position error $\tilde{q} = q_d - q$ is so obtained substituting equation (5.9) in (5.8):

$$\ddot{\tilde{q}}_d + K_D \dot{\tilde{q}}_d + K_P \tilde{q}_d = 0 \tag{5.10}$$

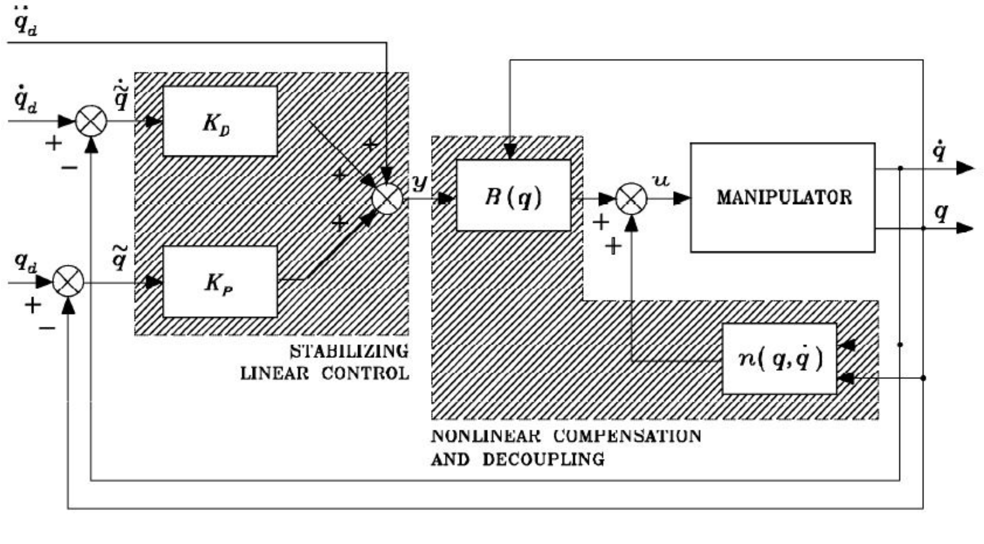


Figure 5.2: Block scheme of the inverse dynamics control strategy [4]

Inverse dynamics control is often chosen when the task is to make the manipulator follow a joint space trajectory when there isn't interaction with the environment, but it is also an essential block of impedance control.

In presence of end-effector forces a nonlinear coupling term due to contact forces arises in equation (5.6) , that becomes:

$$\ddot{q} = y - B^{-1}(q)J^T(q)h_e \tag{5.11}$$

The first operation to obtain again linearity and decoupling is to modify equation (5.5) as follows:

$$u = B(q)y + n(q, \dot{q}) + J^T(q)h_e \tag{5.12}$$

Defining the end-effector pose x_e and using time differentiation is possible to demonstrate that:

$$\ddot{x_e} = J_A(q)\ddot{q} + \dot{J}_A(q, \dot{q})\dot{q} \tag{5.13}$$

where J_A is the analytical Jacobian. Under the assumption of error-free force measurements, to obtain a dynamic behaviour of the position error in the operational space:

$$\tilde{x} = x_d - x_e \tag{5.14}$$

formally equivalent to a mass, spring, dumper system, the following control action y must be chosen:

$$y = J_A^{-1}(q)M_d^{-1}(M_d\ddot{x}_d + K_D\dot{\tilde{x}} + K_P\tilde{x} - M_d\dot{J}_A(q, \dot{q})\dot{q} - h_A) \tag{5.15}$$

Finally substituting equations (5.13), (5.14) and (5.15) in (5.11) yields

$$M_d\ddot{\tilde{x}} + K_D\dot{\tilde{x}} + K_P\tilde{x} = h_A \tag{5.16}$$

where M_d is a positive definite matrix that represent the mass, K_D the dumper and K_P the spring of the system to render, while h_A is the vector of equivalent forces. Figure 5.3 shows the block scheme of this control strategy.

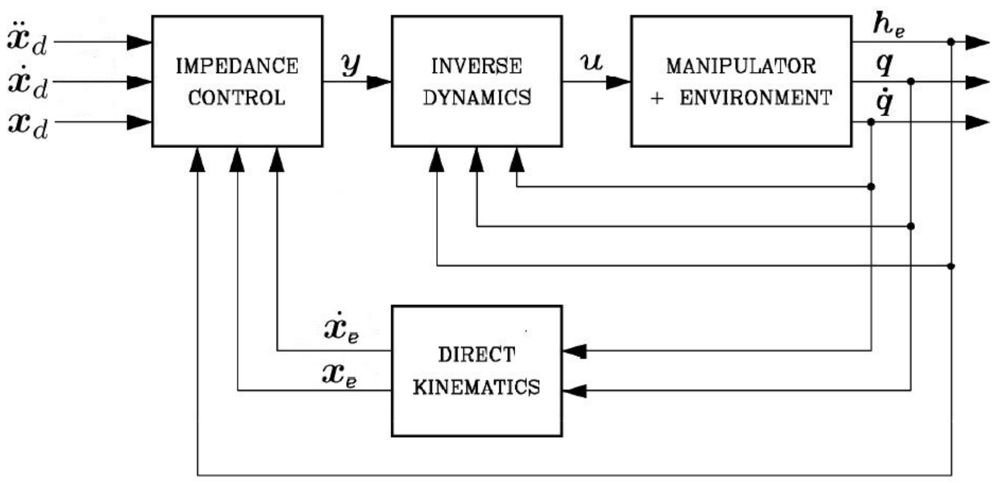


Figure 5.3: Block scheme of the impedance control strategy [4]

5.3 Black-box approach

The impedance control algorithm would be a suitable solution to complete successfully all the operations needed to test the docking prototype. Unfortunately, a series of impediments of practical nature made the implementation of the control strategy unachievable, as described in section 5.2:

- A direct communication with the arm is not realizable, due to the fact that the PCAN-PCI adapter has only two channels. Channel 1 is used by the framework, and has to be kept connected, otherwise the finite state machine will enter in an error state. Channel 2 must be used for the FTM 115 sensor.
- The dynamic model (equation (5.2)) is not implemented for the LWA 4P, and is not realizable externally due to the lack of documentation regarding the arm parameters, and the impossibility to read directly from the arm the generalized coordinates q and their derivatives $\dot{q}$.
- The framework hasn't a function to read the end-effector velocity $\dot{x_e}$, needed to close the loop (figure 5.3).
- Only the code related to the LWA 4P is directly accessible and modifiable, while the underlying functionalities are already compiled and, in many cases, also the related source code is missing.

Moreover, the goal of these tests is to prove that the docking prototype can compensate for small offsets between the axis of the target and the chaser, so rendering a specific dynamic behavior at the end-effector is not mandatory.

In the first phase of the test, while the target approaches the chaser, the control system of the active part will move the slides on which the probe is mounted accordingly to the measurements coming from the displacement sensor, so using a small vertical velocity, starting from the initial position shown in figure 5.1 is sufficient.

In the following phases the only few conditions must be guaranteed:

- The axis of the target has to be vertical, so any unwanted rotation around any axis of the end-effector frame must be corrected. In other words only linear movements are allowed.
- For safety reasons the linear velocities must be limited to low values, in the order of some cm/s.
- The end-effector frame must move in the same direction of the forces sensed by the FTM115.
- Initial bias and noise coming from the sensor, that could produce false readings, must be properly managed.

To guarantee the precedent conditions and to work around all the practical impediments, a **Black-box approach** was chosen. In practice this means that, using the data from the sensor, a velocity reference is iteratively computed and given as an input to the arm, using the user command *cvelall*, described in section 3.1.1. In sections 5.3.1 and 5.4 two different control strategies suitable for testing the docking mechanism are described.

5.3.1 Proportional Control

Since it is not mandatory to reproduce a particular dynamic behaviour, a possibility is to implement a control algorithm that tries to nullify the forces, measured by the sensor, by moving in the Cartesian space the end-effector in the same direction of the forces. So the set point to use as reference in the negative feedback loop is a null vector:

$$F_{ref} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The algorithm must eliminate the noise n_s produced by the sensor, at least for small forces, in order to guarantee that the movement of the end-effector will always be in the same direction of the force. The simplest solution to this problem, since $|n_s| < 1N$, is to impose a null velocity for small forces (red line in figure 5.4). The relationship between the force and the velocity to use as input can be chosen arbitrarily, for this application a proportional one, that transform N in mm/s is sufficient (blue line in figure 5.4). It is also mandatory, for safety reasons, to limit the maximum velocity, so the force readings are saturated via software at $|F_{max}| = 100N$, producing a maximum velocity of $|\dot{x}_{max}| = 0.1\frac{m}{s}$ (green line in figure 5.4), so even if the forces grow, over a certain threshold the velocity will be kept constant. These three condition finally produce the characteristic in figure 5.4

$$\dot{x}_{in} = \begin{cases} 0, & \text{if } |F| < 2 \\ -0.001 \cdot (F_{ref} - F), & \text{if } |F| < F_{max} \\ \dot{x}_{max}, & \text{if } |F| \geq F_{max} \end{cases} \tag{5.17}$$

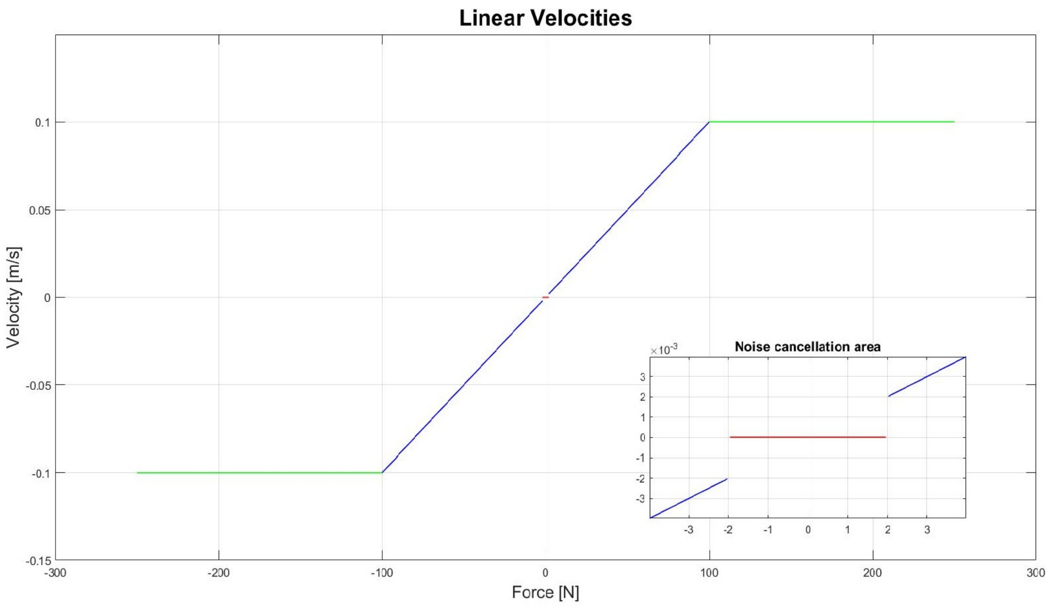


Figure 5.4: Set point control force/velocity characteristic

This strategy produces a passive compliant behaviour whose characteristic can be easily modified and that is quite safe for the prototype, since once the sensor doesn't read any linear force, no movement will be imposed to the end-effector frame.

5.3.2 Practical implementation

In order to communicate with both the robotic arm and the force/torque sensor the PCAN-PCI dual channel adapter (part number IPEH-002065) is used. Since the manipulator uses the CANopen (CiA DS402:IEC61800-7-201) protocol for the motors, while the communication with the sensor is based on a lower level one, described in section 4.3, it's necessary to use both the available channels.

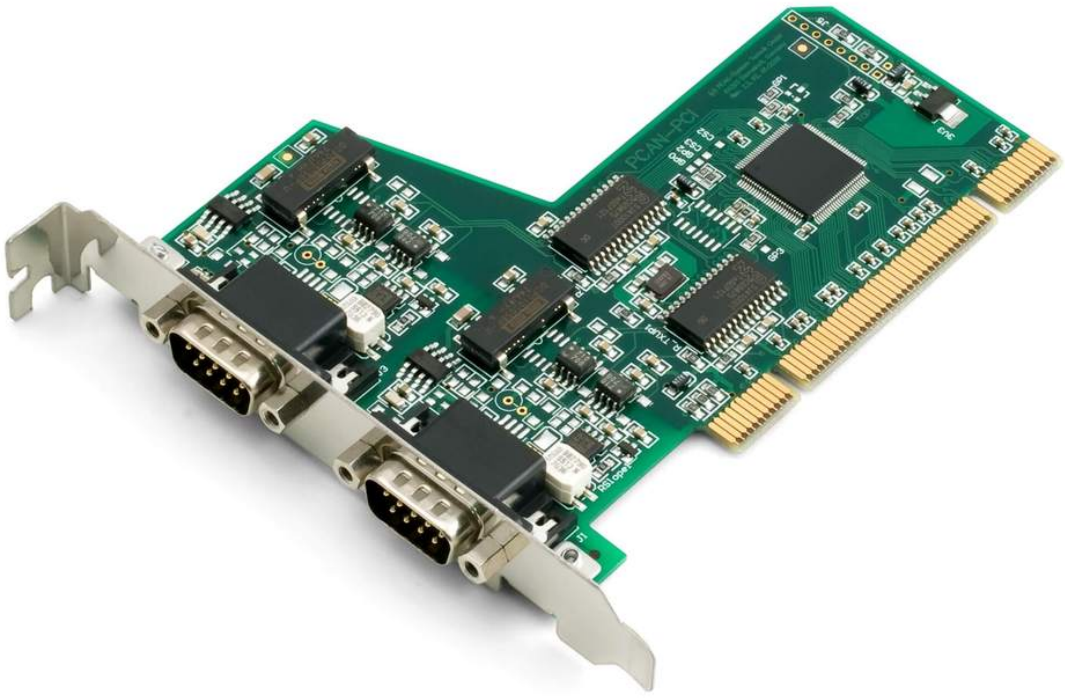


Figure 5.5: PCI to CAN adapter

The main features of this device (figure 5.5) are [11]:

- PC plug-in card for PCI slot.
- 2 High-speed CAN channels (ISO 11898-2).
- Bit rates up to 1 Mbit/s.
- Compliant with CAN specifications 2.0A (11-bit ID) and 2.0B (29-bit ID).
- CAN bus connection via D-Sub, 9-pin (in accordance with CiA® 102).
- NXP SJA1000 CAN controller, 16 MHz clock frequency.
- NXP PCA82C251 CAN transceiver.
- 5-Volts supply to the CAN connection can be connected through a solder jumper, e.g. for external bus converter.
- Extended operating temperature range from -40 to 85 °C (-40 to 185 °F).

PEAK-System Technik GmbH provides also a free driver, whose functionalities are contained in the `pcan.h` and `libpcan.c` files, compatible with this (and other) adapter, with a *ioctl()* interface. In this way user programs are able to receive and to transmit CAN-telegrams, to get information about the channel status and to initialize the CAN-channel [8].

Important note: since the framework runs on a Linux-based operative system (CentOS), external devices are seen by the operative system as *device nodes*. The framework will automatically create the *pcan0* and *pcan1* nodes, correspondent to channel 1 and channel 2, once the workframe is started for the first time after a power cycle, using the `pcan\_make\_devices` function provided by the driver.

To implement the algorithms described in sections 5.3.1 and 5.4, a new command called `sensor` is added to the set available to the user, automatically integrated in the workframe by the constructor of the class `PBConsole`.

Once the `sensor` command is called by the user in the console terminal, four groups of variables are created. The `PCAN PARAMETERS` are used by all the functions related with the initialization or communication with the sensor (based on the driver library). Each of these functions return only error code (`DWORD`) relative to the PCAN driver, to be compared with the ones contained in `pcan.h`. If some error occurred, the execution of the command will be killed. The second group of variables, the `FTM115 PARAMETERS`, is related to the ones needed for the computation of forces and torques, described in section 4.3. Active calibration, unit codes, counts per unit and matrix of coefficients will be saved in this variables one time, at the beginning, since they will remain constant, while the SG data and the bias are iteratively requested. The last two group, `CARTESIAN VELOCITY PARAMETERS` and `CARTESIAN POSITION PARAMETERS`, need respectively to produce a velocity reference for the Cartesian Velocity controller, and to measure possible angular misalignment that must be compensated. In particular the structures `CartesianVelocityReference` and `CartesianPositionFeedback` are defined in the file `ctrl_defines.h`.

At this point the `INITIALIZATION PHASE` stars, giving the following output:

```
>>>>>>>>>>>>>>>>>>>>>>>>>INITIALIZATION PHASE<<<<<<<<<<<<<<<<<<<<<<<<<
Driver Version: Release_20130131_n

Tx/Rx     ID          DLC       D0   D1   D2   D3   D4   D5   D6   D7

[1] Read FT Serial Number -----> Serial Number=  FT18373
Tx        0x205       0
Rx        0x205       8         46   54   31   38   33   37   33   0

[2] Set Active Calibration -----> activeCalibration= 0
Tx        0x206       1         0
Rx        0x206       1         0
```

```
[3] Read Counts per Units -----> CpF= 1000000, CpT= 1000000
Tx        0x207      0
Rx        0x207      8         0    F    42   40   0    F    42   40

[4] Read Unit Codes -----> ucF = 2 , ucT = 3
Tx        0x208      0
Rx        0x208      2         Force: N    Torque: N-m

[5] Read Matrix
Tx        0x202      1         0
Rx        0x202      8         C3   5F   32   26   42   E3   C4   4E
Rx        0x203      8         45   EF   80   B5   C6   EB   C4   E6
Rx        0x204      8         C5   2A   A3   FA   46   E0   7A   DA
Tx        0x202      1         1
Rx        0x202      8         C5   C2   AC   E0   47   1    E4   51
Rx        0x203      8         45   6F   C8   EE   C6   88   AB   D4
Rx        0x204      8         45   52   D5   AA   C6   83   2C   B3
Tx        0x202      1         2
Rx        0x202      8         47   1C   46   29   44   BF   6A   AE
Rx        0x203      8         47   21   87   76   44   E9   2F   56
Rx        0x204      8         47   16   C8   46   44   97   69   31
Tx        0x202      1         3
Rx        0x202      8         C2   76   3F   CC   43   63   F0   E6
Rx        0x203      8         C4   1B   95   E8   C3   19   CE   15
Rx        0x204      8         44   22   51   45   C2   BB   95   83
Tx        0x202      1         4
Rx        0x202      8         44   3E   DD   49   41   EE   98   FD
Rx        0x203      8         C3   DB   CB   4E   43   3E   D    6
Rx        0x204      8         C3   AE   8B   57   C3   51   A4   EC
Tx        0x202      1         5
Rx        0x202      8         42   6E   F    66   C3   D2   71   2D
Rx        0x203      8         43   6    FA   F8   C3   DA   68   26
Rx        0x204      8         42   2E   58   F0   C3   D1   6D   53

------------------------Matrix of Coefficients------------------------
        G0         G1         G2         G3         G4         G5
Fx   -223.20     113.88    7664.09  -30178.45   -2730.25   28733.43
Fy  -6229.61   33252.32    3836.56  -17493.91    3373.35  -16790.35
Fz  40006.16    1531.33   41351.46    1865.48   38600.27    1211.29
```

```
Tx    -61.56     227.94    -622.34    -153.81     649.27     -93.79
Ty    763.46      29.82    -439.59     190.05    -349.09    -209.64
Tz     59.52    -420.88     134.98    -436.81      43.59    -418.85
```

At the end the user is asked to specify manually the velocity timeouts for the joint space and the Cartesian space:

```
Set velocity timeouts :
```

This step is added for safety reasons, but is redundant if the user, before using the `sensor` command, manually sets the Cartesian velocity timeout to a value small enough but still greater than the sampling time, defined at the end of the loop by the function `Sleep(1)` (see later). Actually the `Sleep(1)` function stops the execution of the while loop for one millisecond, so no data request is sent to the sensor and the velocity reference is maintained for this amount of time, which is basically the sampling time. During this period the framework will move in the Cartesian space the end-effector until a new reference is issued. At the last iteration, since the data acquisition phase is finished, a new reference won't but produced, so the framework will maintain the last one for a period equal to the Cartesian velocity timeout that, by default is set to 3 seconds. To avoid any dangerous behaviour, each time the `sensor` command is used, the user will be asked to set the velocity timeouts. It is recommended to set the Cartesian one (second number) between the sampling time and two times the sampling time, while the joint timeout won't affect the execution of the command:

```
Set velocity timeouts :  1000 2
```

The second operation needed is the initialization of channel two, that needs the device node path (`/dev/pcan1`). Then the parameters, needed for the computation of forces and torques, are requested in the order described in section 4.3. During this phase all these information are printed in the console terminal.

Then a finite while loop starts and at each iteration the function `ReadRawData` asks for the strain gauge data and the status of the sensor. If for any reason the sensor produces an error code, it will be reset just before the execution of the loop may break down.

At this stage the force torque readings should be zero, but the sensor is affected by a strong initial bias, to be subtracted from the future readings. For the first 100 iterations the function `BiasCorrector(bias,newbias)` performs the mean between the mean of the precedent bias (`bias`) and the new sensed one, saving it in the variable `newbias`. Although this method to estimate the exact bias is extremely simple, it works fine for this application. A great advantage of this method is that if some heavy hardware is mounted on the sensor its weight won't be considered as an active force, so it won't produce any velocity reference.

Once the bias is finally estimated, the new raw data readings are converted in a meaningful form by the function `ForceTorqueComputation`, that uses the previously saved parameters of the sensor and saves forces and torques inside the array `forceTorque[6]`.

Important note: The end-effector frame and the axis of forces and torque are not coincident. The sensor frame and the end-effector one have in common the same z axis, but the first one is rotated with respect to the second of 60° around the vertical axis, counter clock wise. This aspect doesn't affect the sensed forces and torques, but must be taken into consideration during the computation of the velocities.

Then, before computing the velocities, the last step is about the Cartesian angular position (P_a). The `VelComputation` function will generate a linear velocity reference accordingly to the user's request, and a small and constant angular velocity ($\dot{x}_a$) in the event that the angular error is greater than a predefined threshold (figure 5.6):

$$\dot{x}_a = \begin{cases} 0, & \text{if } |P_a| < 0.01rad \\ -sgn(P_a) \cdot 0.03, & \text{if } |P_a| \geq 0.01rad \end{cases} \tag{5.18}$$

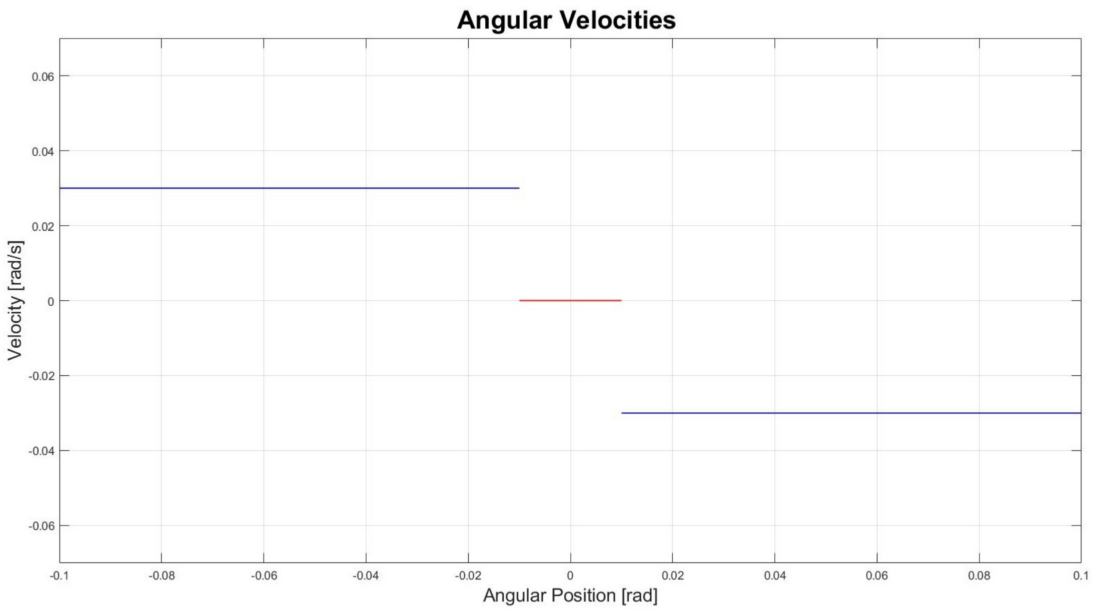


Figure 5.6: Angular error compensation

Finally a velocity reference is given to the arm controller and, before restarting the loop, the `Sleep(1)` function stops the execution for 1 millisecond, while the arm performs the requested movement. The last operation to perform is closing the channel since, if kept open, a new call of the sensor command will generate an error during the initialization phase.

Since the loop is limited to a certain amount of iteration but is not possible to know how much time the test will takes, a possible improvement could be make the while loop endless until a predefined keyboard input arises, or still maintain it limited but, before exiting, ask the user if the test is finished or if the loop has to continue.

5.4 Limits and possible upgrades

The biggest limit of the *Black-box approach* is related to the fact that, once the velocity reference is generated and passed to the Cartesian velocity control of the arm it is guaranteed that the end-effector will move in the same direction of the sensed force, but it isn't guaranteed that the real velocity is equal to the reference. When the arm is moving towards a singular configuration the framework will automatically reduce the velocity and, in some cases, even stop completely the operation. During the tests of the mechanism this behaviour won't damage nor the manipulator or the mechanism, but is necessary top place the active element of the mechanism in such a way that the LWA 4P doesn't have to move near singularities.

For what concerns the code two small upgrades can be performed:

1. Each time the command `sensor` is called from the console terminal, the user has to specify the velocity timeouts, as described in section 5.3.2, because the default one (3000 milliseconds) could cause problems once the *while* loop is completed. A good idea could be to substitute this operation with a two function, one that at the beginning automatically sets the Cartesian velocity timeout accordingly to the sampling time, and the other that resets to the default value this parameter once the execution of the command is almost completed.

2. The second possible upgrade regards the *while* loop. Since there is no way to know how much a test will take, it would be better to use an endless loop instead of a limited one, adding the option to break it in any moment with an input from the keyboard.

Another possibility is to try to reproduce the impedance control algorithm, simulating the dynamics of a satellite in space. Starting from equation (5.1), that describes a mass-spring-damper system, it's possible to simplify it to:

$$m\ddot{x}(t) = f_{ext}(t) \tag{5.19}$$

considering that in orbit, if the gravitational effect are counterbalanced by the centrifugal one, a mass is subject just to external forces and its inertia. Again, the only practical option is to give to the arm controller a velocity reference after each sampling period, so, manipulating equation (5.19):

$$m\ddot{x}(k) = m\frac{\dot{x}(k) - \dot{x}(k-1)}{k} = f_{ext}(k) \tag{5.20}$$

Using as initial condition $\dot{x}(0) = 0$, and choosing arbitrarily the mass, with a fixed sampling time $k = 1$ ms, a new input can be iteratively computed, according to the following equation:

$$\dot{x}(k) = \frac{k \cdot f_{ext}(k)}{m} + \dot{x}(k-1) \tag{5.21}$$

This approach will be more challenging for the controller of the prototype and the chosen mass can't be too high, in order to avoid damaging the hardware. For safety reasons, even in this case, the maximum velocity must be limited and only linear movements should be allowed. To implement this control strategy most of the code reported in Appendix B can be re-used and, probably, the most efficient way could be implementing it inside the command `sensor` and the `Sensor.h` and `Sensor.cc` files, asking at the beginning which strategy the user prefers.

Chapter 6

Conclusions

The focus of this thesis project was to implement a suitable control strategy for a six degrees of freedom robotic manipulator and a force/torque sensor, to test a docking mechanism for servicing missions, characterized by an unmanned servicer and a cooperative target.

A brief discussion on the most common mating techniques in space introduces a detailed description of the main elements of the second design iteration of the mechanism which has been tested, in order to better understand its main features and how they have to be put to the test.

Then all the information collected about the Schunk LWA 4P and the G.R.A.A.L framework are presented, with a particular attention to the finite state machine, the controller and the commands available to the user. Practical examples are added to create a solid knowledge of the system and how to operate it safely.

The CAN bus protocol, used to communicate both with the sensor and the arm, is presented starting from the ISO 11898 and its layers. The standard data frame and the main features of the protocol are deeply illustrated to introduce and better understand how it was implemented for the Schunk FTM 115, and minutely describes its functions and the procedure to properly obtain the data.

It is then followed by a description of the test, that recalls directly the phases of a real docking maneuvers. Although the impedance control strategy would be suitable to complete all the needed operations, some impediments of practical nature led to an easier but still effective approach, based on the available functionalities of the existing software.

Two appendices are added, one to guide future users through the possible issues that they can encounter while operating this system, the other to report all the code relative to the sensor and the control strategy implemented.

Appendix A

Procedures and known issues

The purpose of this appendix is to give to the reader the essential information to properly and safely use the Schunk LWA 4P, the FTM115 and the G.R.A.A.L framework. Then it follows a brief list of warnings and a description of how to manage particular situations and the known issues.

A.1 Start-up and shut-down procedures

This section contains all the steps to safely start-up both the arm and the framework.

- Verify that the orange contactor behind the protection cage of the arm is in safe position (down, as shown in figure A.1).

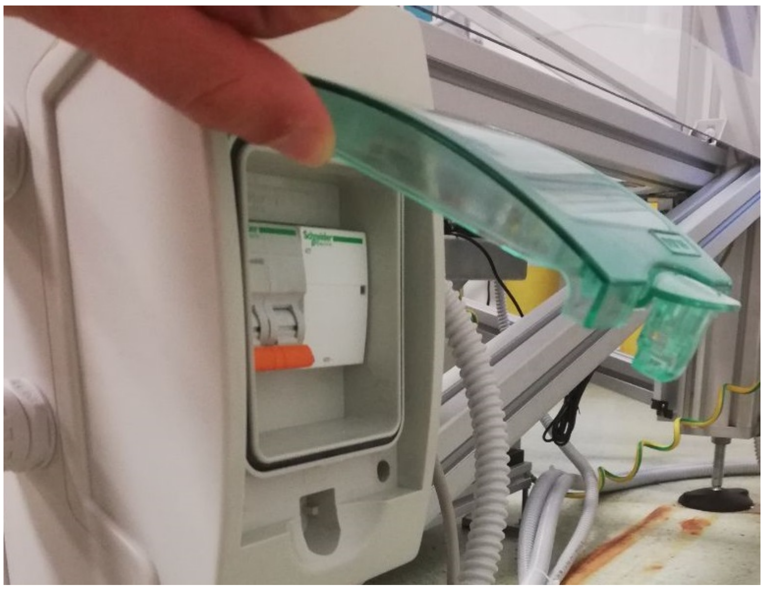


Figure A.1: Contactor for the LWA 4P

- Connect the three plugs of the two power supplies to three 220 V electric sockets.

- Switch on both the power supplies and verify that:
  1. The outputs are correctly enabled.
  2. The output voltages are both set at 24 V.
  3. The maximum output current for the one that supplies the control logic is set to 2.2A (to set this limit the output must be disabled).
  4. The maximum output current for the one that supplies the motors is set to 7A.

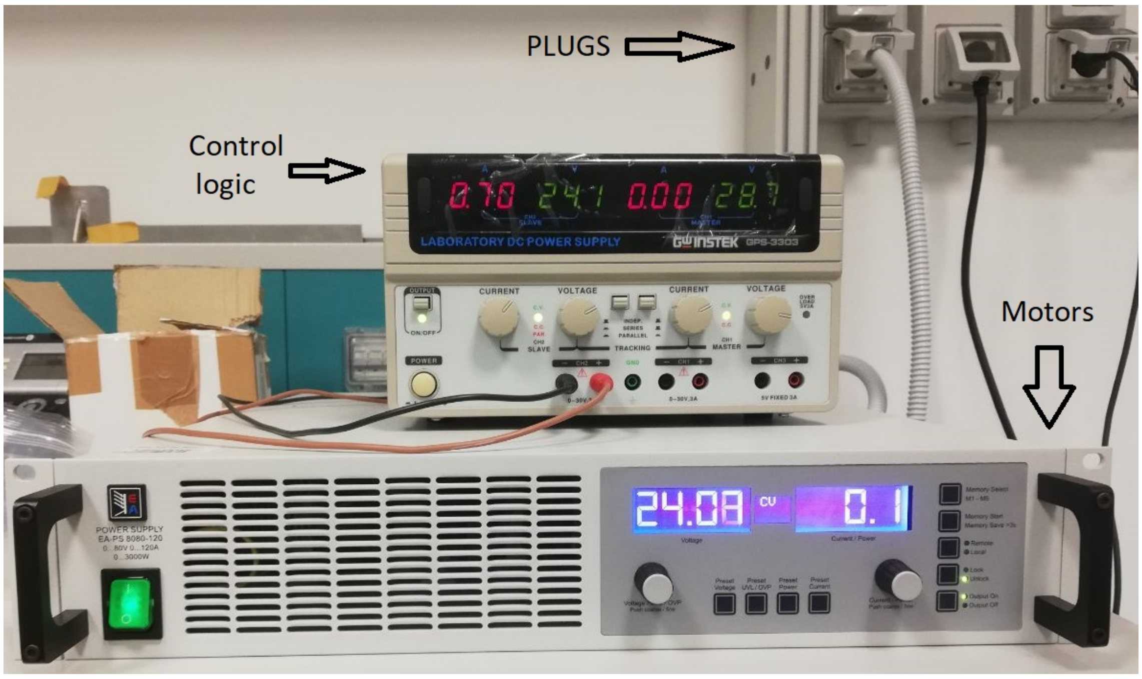


Figure A.2: Power supplies for the control logic and the motors

- Enable the orange contactor; after this operation the current readings on the displays of the power supplies should be the one reported in figure A.2 and the four leds behind the arm base plate should be all green.
- Switch on the control pc and open three terminals.
- On two of the three terminals enter as *Super User* typing:

  ```
  su
  ```

  and after pressing Enter use the password **tbratbra**.
- Use one of the two terminal to start-up the workframe typing:

```
workframe start && console start
```

This command will also create the two device nodes needed by the PCAN-PCI adapter.

- Once this operation is completed use the second super user terminal to start-up the controller and wait for the completion of the preliminary operations:

  ```
  controller-powerball start && controller-powerball console
  ```

  A list of all the available commands will be printed on this terminal.

- Finally, on the workframe terminal type:

  ```
  start
  ```

  At this point is possible to control the arm using the commands described in Chapter 3 just the console.

- To open the GUI use the third terminal and type:

  ```
  cd Desktop
  ./PowerballGUI
  ```

In order to safely turn off the arm and the workframe:

- Disarm the manipulator using or the dedicated button in the GUI, or the command from the console terminal.

- On the workframe terminal use the command:

  ```
  shutdown
  ```

- Exit from the super user mode in both the workframe and console terminals using the command:

  ```
  exit
  ```

- Close the GUI and the three terminals.
- Disable the contactor and turn off the power supplies.

To compile the software relative to the Schunk manipulator enter the *powerballctrl* folder from the terminal using the **cd devel/powerballctrl/** command and then issue the following commands to clean, compile and install (waiting that each one completes its operations):

```
make rtai_clean
make rtai_all
make rtai_install
```

A.2 Known issues

In this section are listed the problems encountered while using the framework developed by G.R.A.A.L, the Schunk LWA 4P manipulator, the FTM115 force/torque sensor and the PCAN drivers, in order to avoid possible dangerous configurations for both the hardware or the future users.

- **Encoders reset**: Sometimes, mainly during the arming procedure, the joint position gets corrupted. In this case the controller will enter automatically in the GOHOMING state. To solve the problem is necessary to follow the steps described in section 3.2 to reset the encoders, shutdown and restart the workframe.
- **Arming errors due to contact**: If the robot was disarmed stopped with the emergency button in a pose where one of its components touches with the ground, the safety cage or itself, the next arming procedure will generate errors and in most of the cases the controller won't automatically switch in the GOHOMING state. For example, if the FTM115 is mounted, it is better to not use the full range of joint 5 because the sensor will touch the component that connects joint 3 and 4 with 5 and 6. To solve the problem is necessary to give manually to the stuck elements a little bit of clearance, re-arm and perform the encoders reset procedure. It is recommended to use only small velocities, to limit the range of the joints and always be ready to press the emergency button to avoid this case.
- **Arming errors due to joint 6**: To attach the sensor to the arm, two c-shaped spring loaded elements are used. By tightening two bolts the relative position of the c-shaped components is reduced around the conic base of the sensor, ensuring a solid connection. But when nor the sensor or other additional elements are used with the arm, it is important to not tighten too much the previously mentioned

bots. In this case joint 6 will produce an error during the arming procedure and the controller will remain endlessly in the ARMING state. This is just a temporary state dedicated to automatic operation and checks before entering the ARMED state, so the user can't use any command and can't even force the controller to enter in the GOHOMING one. Fortunately it's sufficient to simply loosen the bolts and restart the workframe.

- **CAN errors**: these are sporadic errors and usually it is not so simple to find why they happen. For sure the first operation to perform is controlling that the CAN cable of the motors of the arm is properly connected to channel 0 of the PCAN-PCI board. If this is not the problem, probably it is correlated to the control pc and its interaction with the PCAN-PCI, so usually is enough to restart the pc and the workframe.

- **GUI compilation problem**: if the code is re-compiled the trajectory manager won't function properly anymore, and the sliders will still maintain their function, but won't graphically represent the proper range of the joints.

- **GUI presets corrupted readings**: using the `Read` button (see section 3.2.2) near the zero position of the joints, some of the saved joint values could be corrupted, so, before using the new preset issue the `jpos` command from the console terminal to print all the current joint values, in order to double check that the reading was successful.

- **PCAN drivers**: the PCAN driver is properly configured to work with the workframe, but if for any reason they have to be re-installed, follow the *PCAN Driver for Linux -User Manual* guide, specifying during the compilation and installation process the option for no real time support:

  ```
  make clean
  make RT=NO_RT
  su -c \make install"
  ```

  If the real time support, that by default is active, will be kept in this state, it won't be possible to start-up the workframe.

- **FTM115 frame misalignment**: an important aspect to consider while using the force/torque sensor is that x and y axis of the frame related to the measurement of forces and torques is rotated of about sixty degrees clock-wise with respect to the end-effector frame. In listing B.3 it's possible to see that, to compensate for this problem, in the function *VelComputation* the sensed F_x and F_y

(ft[0] and ft[1]) are rotated to obtain forces in the same direction of the x and y axis of the end-effector (realFx and realFy).

Appendix B

Code

In this appendix all the code relative to the FTM115 is listed, from the data acquisition to the transformation of this data in Cartesian velocity references. In listing B.1 are reported the parts of code relative to the new command sensor inside the class **PBConsole**. The command is automatically created by the class constructor, once it is called, by the function *AddCommand*.

Listing B.1: PBConsole.cc

```
#include "Sensor.h"
.
.
.
//PBConsole contructor:
PBConsole::PBConsole(void)
{
    .
    .
    .
    AddCommand("SENSOR",    "", "Use the sensor to drive the arm",
                (ConsoleMemberFunction)&PBConsole::CommandSENSOR);
    .
    .
    .
}
.
.
.

int PBConsole::CommandSENSOR(void)
{
    //PCAN PARAMETERS
    DWORD err;
    HANDLE deviceHandle_;

    //FTM115 PARAMETERS
    uint8_t ucF;
    uint8_t ucT;
    uint8_t activeCalibration;
    uint32_t CpF;
    uint32_t CpT;
    float matrix[36];
    int16_t rawData[6];
    char FTMstatus[16];
    float forceTorque[6];
    int16_t newbias[6];
    int16_t bias[6];

    char yawEnable = 'n';

    //CARTESIAN VELOCITY PARAMETERS
    int32_t ret=CTRL_RV_OK;
    float linearVel[3];
    float angularCompensation[3];

    //CartesianVelocityReference is defined in ctrl_defines.h
    CartesianVelocityReference cvelref;

    cvelref.enable[0] = true;
    cvelref.enable[1] = true;
    cvelref.enable[2] = true;
    cvelref.enable[3] = true;
    cvelref.enable[4] = true;
    cvelref.enable[5] = true;

    //Cartesian position for angular compensation
    //CartesianPositionFeedback is defined in ctrl_defines.h (yaw pitch roll x y z)
    CartesianPositionFeedback cposfbk;
    float angularError[3];


    cout << ">>>>>>>>>>>>>>>>>>>>>>>>>INITIALIZATION PHASE<<<<<<<<<<<<<<<<<<<<<<<<<" << endl;
    err = PcanInit(SENSOR_PATH, deviceHandle_);
    err = ReadFTSerialNumber(deviceHandle_);
    err = SetActiveCalibration(deviceHandle_, activeCalibration);
    err = ReadCountsPerUnits(deviceHandle_, CpF, CpT);
    err = ReadUnitCodes(deviceHandle_, ucF, ucT);
    err = ReadMatrix(deviceHandle_,matrix);
    PrintMatrix(matrix);


    cout <<"Set velocity timeouts (write '1000 2' and press Enter): ";
    CommandSetParameterVelocityTimeouts();
    cout << "Do you want to enable the yaw axis ? (y=YES, n=NO) ";
    cin >> yawEnable;

    if(yawEnable=='y'||yawEnable=='Y')
        cout << endl << "Yaw Enable = YES";
    else
        cout << endl << "Yaw Enable = NO";

    cout << endl << ">>>>>>>>>>>>>>>>>>>>>>>ACQUIRING DATA<<<<<<<<<<<<<<<<<<<<<<<" << endl;
    //PrintFTHeader();

    int i = 0;
    while(i<25000){

        if(i<100){// Bias computation
            err = ReadRawData(deviceHandle_, newbias, FTMstatus);
            BiasCorrector(bias,newbias);

        }else{
            err = ReadRawData(deviceHandle_, rawData, FTMstatus);

            if(FTMerrorCheck(FTMstatus)){
    //Computation of Fx,Fy,Fz,Tx,Ty,Tz starting from the raw data coming from the sensor
    //(ATTENTION: the sensor frame is rotated of 60 deg ccw, this aspect is taken into
    //consideration in the computation of the linear velocities, but just for Fx and Fy)
                ForceTorqueComputation(forceTorque, rawData, bias, matrix, CpF, CpT);
                //PrintForceTorque(forceTorque);

                //Reading the cartesian position to eventually compensate errors for roll,
                //pitch and yaw
                pbCtrlInterface_->ReadCartesianPositionFeedback(cposfbk);
                angularError[0] = cposfbk.x[2];
                angularError[1] = cposfbk.x[1];
                angularError[2] = cposfbk.x[0];

                //The forces are converted into linear velocities and eventually
                //the angular compensation
                VelComputation(linearVel,angularCompensation,forceTorque,angularError);
                //PrintLinearVel(linearVel);

                if(yawEnable=='y'||yawEnable=='Y'){
                    angularCompensation[2] = YawComputation(forceTorque[5]);
                }

                cvelref.reference[0] = angularCompensation[0];  //roll
                cvelref.reference[1] = angularCompensation[1];  //pitch
                cvelref.reference[2] = angularCompensation[2];  //yaw
                cvelref.reference[3] = linearVel[0];            // x
                cvelref.reference[4] = linearVel[1];            // y
                cvelref.reference[5] = linearVel[2];            // z

                ret = pbCtrlInterface_->WriteCartesianVelocityReference(cvelref);
                ret = WriteCommand(CTRL_COMMAND_CVELCTRL);

            }else{//If something goes wrong the sensor is resetted
                cout << "Resetting the sensor!" << endl;
                err = Reset(deviceHandle_);
                cout << "Please shutdown the arm and the workframe, then try again! If
                        something doesen't work restart also the pc" << endl;
                break;
            }
        }

        if (ret == CTRL_RV_OK){
            Sleep(1);//ms
            ++i;
        }else{
            break;
        }
    }


    //Closing the can channel of the sensor
    err = CAN_Close(deviceHandle_);

    // Returning in Holding state
```

```
        ret = WriteCommand(CTRL_COMMAND_HOLD);
        if(ret!=CTRL_RV_OK)
            return ret;

    return ret;
}
```

Listings B.2 and B.3 contain the code to manage the CAN communication with the sensor, the error handling and the transformation of the raw data in usable one. The files *Sensor.cc* and *Sensor.h* are in the same folder of the code relative to the LWA 4P (*powerballctrl*). Since they were added, in the makefile, to the list of files to be compiled, during the compilation and installation procedure described in Appendix A, they will be automatically integrated inside the framework.

Listing B.2: Sensor.h

```
#ifndef _SENSOR_H_
#define _SENSOR_H_

#include <iostream>
#include <iomanip>
#include <bitset>
#include <cmath>
#include <vector>
#include <string>
#include <stdint.h>
#include <math.h>
#include "libpcan.h"
#include "time.h"
#include "fcntl.h"


// DEFAULT PORT FOR THE SENSOR (ON PCAN0 IS MOUNTED THE ARM)
#define SENSOR_PATH "/dev/pcan1"

//--------FORCE/TORQUE INDEX--------
#define FX                                      (0)
#define FY                                      (1)
#define FZ                                      (2)
#define TX                                      (3)
#define TY                                      (4)
#define TZ                                      (5)

//--------MESSAGES--------
#define STATUS_SG0_SG2_SG4                  (0x200)
#define SG1_SG3_SG5                         (0x201)
#define MATRIX_SG0_SG1                      (0x202)
#define MATRIX_SG2_SG3                      (0x203)
#define MATRIX_SG4_SG5                      (0x204)
#define FT_SERIAL_NUMBER                    (0x205)
#define SET_ACTIVE_CALIBRATION              (0x206)
#define COUNTS_PER_UNIT                     (0x207)
#define UNIT_CODES                          (0x208)
#define RESET                               (0x20C)
#define SET_BASE_ID                         (0x20D)
#define SET_BAUD_RATE                       (0x20E)
#define FIRMWARE_VERSION                    (0x20F)
```

```

//------STATUS------
#define WATCHDOG_RESET                    (0)
#define DAC_ADC_TOO_HIGH                  (1)
#define DAC_ADC_TOO_LOW                   (2)
#define ANALOG_GROUND_OOF                 (3)
#define POWER_SUPPLY_TOO_HIGH             (4)
#define POWER_SUPPLY_TOO_LOW              (5)
#define BAD_ACTIVE_CALIBRATION            (6)
#define EEPROM_FAILURE                    (7)
#define CONFIGURATION_INVALID             (8)
#define RESERVED_1                        (9)
#define RESERVED_2                       (10)
#define TEMPERATURE_TOO_HIGH             (11)
#define TEMPERATURE_TOO_LOW              (12)
#define RESERVED_3                       (13)
#define CAN_BUS_ERROR                    (14)
#define ANY_ERROR                        (15)

//------UNIT CODES------
#define LBF                               (1)
#define N                                 (2)
#define KLBF                              (3)
#define KN                                (4)
#define KGF                               (5)
#define GF                                (6)
#define LBF_IN                            (7)
#define LBF_FT                            (8)
#define N_M                               (9)
#define N_MM                             (10)
#define KGF_CM                           (11)
#define KN_M                             (12)


//------PROTOTYPES------

//---CAN Functions
DWORD PcanInit(const char* path, HANDLE &deviceHandle_);
DWORD Reset(HANDLE &deviceHandle_);
DWORD SensorInit(HANDLE &deviceHandle_);
DWORD ReadFTSerialNumber(HANDLE &deviceHandle_);
DWORD SetActiveCalibration(HANDLE &deviceHandle, uint8_t &activeCalibration);
DWORD ReadCountsPerUnits(HANDLE &deviceHandle_, uint32_t &CpF, uint32_t &CpT);
DWORD ReadUnitCodes(HANDLE &deviceHandle_, uint8_t &ucF, uint8_t &ucT);
DWORD ReadMatrix(HANDLE &deviceHandle_, float m[36]);
void MatrixFeeder(const TPCANRdMsg &rM, float m[36], const int &row, const int &column);
DWORD ReadRawData(HANDLE &deviceHandle_, int16_t rawData[6], char FTMstatus[16]);
void DataStatusFeeder(const TPCANRdMsg &rM, int16_t rawData[6], char FTMstatus[16]);
void DataStatusFeeder(const TPCANRdMsg &rM, int16_t rawData[6]);
bool FTMerrorCheck(const char FTMstatus[16]);
void ForceTorqueComputation(float ft[6],const int16_t rawD[6], const int16_t b[6],
    const float m[36], const uint32_t &CpF, const uint32_t &CpT);
void BiasCorrector(int16_t bias[6], int16_t newbias[6]);
void VelComputation(float linVel[3],float angComp[3],float ft[6],float angErr[3]);
float YawComputation(const float yawTorque);

//---Utilities
TPCANMsg MsgAssembler(const DWORD id);
TPCANMsg MsgAssembler(const DWORD id, const BYTE len, const BYTE data);
```

```
void Sleep(const int &millisec);
void PrintMessage(const TPCANMsg &msg);
void PrintMessage(const TPCANRdMsg &rdMsg);
void PrintHeader();
void PrintMatrix(float m[36]);
void PrintFTMerror(const int &code);
void PrintForceTorque(const float ft[6]);
void PrintLinearVel(const float linVel[3]);
void PrintFTHeader();

//----Conversions
long long int BinToDec(const std::string &bin);
float HexToFloat(const std::string& binaryHex);
int HexToDecSigned(const std::string &bin);


#endif
```

Listing B.3: Sensor.cc

```
#include "Sensor.h"

using namespace std;

    //-------CAN FUNCTIONS-------
    /*>>>>>INITIALIZATION PHASE<<<<<*/
DWORD PcanInit(const char* path, HANDLE &deviceHandle_){
    char txtInfo [1024];
    DWORD err;
    // Opening the device node /dev/pcan1
    deviceHandle_ = LINUX_CAN_Open(path, O_RDWR);
    /* By default the sensor baud rate is 250 kbit/s, so the hardware has
    to be initialized properly*/
    err = CAN_Init(deviceHandle_,CAN_BAUD_250K,CAN_INIT_TYPE_ST);
    // Print version info
    CAN_VersionInfo(deviceHandle_,txtInfo);
    cout << "Driver Version: " << txtInfo << endl << endl;
    PrintHeader();
    return err;
}

DWORD Reset(HANDLE &deviceHandle_){
    DWORD ret;

    // Composing and transmitting the write message
    TPCANMsg writeMessage;
    writeMessage = MsgAssembler(RESET);
    ret = CAN_Write(deviceHandle_,&writeMessage);

    return ret;
}

DWORD ReadFTSerialNumber(HANDLE &deviceHandle_){

    DWORD ret;

    // Composing and transmitting the write message
    TPCANMsg writeMessage;
    writeMessage = MsgAssembler(FT_SERIAL_NUMBER);
    ret = CAN_Write(deviceHandle_,&writeMessage);
    if(ret != 0)
        return ret;

    // Receiving the response
    TPCANRdMsg readMessage;
    ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
    if(ret != 0)
        return ret;

    // Printing (optional)
    cout << endl << "[1] Read FT Serial Number ------> Serial Number=  ";
        for(int i=0; i<readMessage.Msg.LEN; ++i)
            cout << uppercase << readMessage.Msg.DATA[i];
        cout << endl;
    PrintMessage(writeMessage);
    PrintMessage(readMessage);

    return ret;
}

DWORD SetActiveCalibration(HANDLE &deviceHandle_, uint8_t &activeCalibration){

    DWORD ret;

    // Composing and transmitting the write message
    TPCANMsg writeMessage;
    writeMessage = MsgAssembler(SET_ACTIVE_CALIBRATION, 1, 0);
    ret = CAN_Write(deviceHandle_,&writeMessage);
    if(ret != 0)
        return ret;

    // Receiving the response
    TPCANRdMsg readMessage;
    ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
    if(ret != 0)
        return ret;
    activeCalibration = readMessage.Msg.DATA[0];

    // Printing (optional)
    cout << endl << "[2] Set Active Calibration ------> activeCalibration= 0" <<
    endl;
    PrintMessage(writeMessage);
    PrintMessage(readMessage);

    return ret;
}

DWORD ReadCountsPerUnits(HANDLE &deviceHandle_, uint32_t &CpF, uint32_t &CpT){

    DWORD ret;

    // Composing and transmitting the write message
    TPCANMsg writeMessage;
    writeMessage = MsgAssembler(COUNTS_PER_UNIT);
    ret = CAN_Write(deviceHandle_,&writeMessage);
    if(ret != 0)
        return ret;

    // Receiving the response
    TPCANRdMsg readMessage;
    ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
    if(ret != 0)
        return ret;
    CpF = (readMessage.Msg.DATA[0]<<24) | (readMessage.Msg.DATA[1]<<16) |
          (readMessage.Msg.DATA[2]<<8) | readMessage.Msg.DATA[3];
    CpT = (readMessage.Msg.DATA[4]<<24) | (readMessage.Msg.DATA[5]<<16) |
          (readMessage.Msg.DATA[6]<<8) | readMessage.Msg.DATA[7];

    // Printing (optional)
    cout << endl << "[3] Read Counts per Units -----> CpF= " << dec <<
            CpF << ", CpT= " << dec << CpT << endl;
    PrintMessage(writeMessage);
    PrintMessage(readMessage);

    return ret;
}

DWORD ReadUnitCodes(HANDLE &deviceHandle_, uint8_t &ucF, uint8_t &ucT){

    DWORD ret;

    // Composing and transmitting the write message
    TPCANMsg writeMessage;
    writeMessage = MsgAssembler(UNIT_CODES);
    ret = CAN_Write(deviceHandle_,&writeMessage);
    if(ret != 0)
        return ret;

    // Receiving the response
    TPCANRdMsg readMessage;
    ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
    if(ret != 0)
        return ret;
    ucF = readMessage.Msg.DATA[0];
    ucT = readMessage.Msg.DATA[1];

    // Printing (optional)
    cout << endl << "[4] Read Unit Codes -----> ucF= " << (ucF) << ",
            ucT= " << (ucT) << endl;
    PrintMessage(writeMessage);
    cout << left << setw(10) << "Rx" << "0x" << setw(10) << hex << readMessage.Msg.ID <<
    setw(10) << (readMessage.Msg.LEN + 0);

    switch(ucF){
        case 1:
            cout << setw(12) << "Force: lbf";
            break;
        case 2:
            cout << setw(12) << "Force: N";
            break;
        case 3:
            cout << setw(12) << "Force: Klbf";
```

```
                break;
            case 4:
                cout << setw(12) << "Force: kN";
                break;
            case 5:
                cout << setw(12) << "Force: kgf";
                break;
            case 6:
                cout << setw(12) << "Force: gf";
                break;
            default:
                cout << setw(12) << "Force: ERROR";
                break;
        }
        switch(ucT){
            case 1:
                cout << setw(13) << "Torque: lbf-in" << endl;
                break;
            case 2:
                cout << setw(13) << "Torque: lbf-ft" << endl;
                break;
            case 3:
                cout << setw(13) << "Torque: N-m" << endl;
                break;
            case 4:
                cout << setw(13) << "Torque: N-mm" << endl;
                break;
            case 5:
                cout << setw(13) << "Torque: kgf-cm" << endl;
                break;
            case 6:
                cout << setw(13) << "Torque: kN-m" << endl;
                break;
            default:
                cout << setw(13) << "Torque: ERROR" << endl;
                break;
            }

        return ret;
}

DWORD ReadMatrix(HANDLE &deviceHandle_, float m[36]){
        cout << endl << "[5] Read Matrix " << endl;
        DWORD ret;

        for(int i=0; i<6; ++i){
            // Composing and transmitting the write message
            TPCANMsg writeMessage;
            writeMessage = MsgAssembler(MATRIX_SG0_SG1, 1, i);
            ret = CAN_Write(deviceHandle_,&writeMessage);
            if(ret != 0)
                return ret;
            PrintMessage(writeMessage);

            // Receiving the response
            TPCANRdMsg readMessage;
            ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
            if(ret != 0)
                return ret;
            MatrixFeeder(readMessage, m, i, 0);
            PrintMessage(readMessage);

            ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
            if(ret != 0)
                return ret;
            MatrixFeeder(readMessage, m, i, 2);
            PrintMessage(readMessage);

            ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
            if(ret != 0)
                return ret;
            MatrixFeeder(readMessage, m, i, 4);
            PrintMessage(readMessage);
        }

        return ret;
}

void MatrixFeeder(const TPCANRdMsg &rM, float m[36], const int &row, const int &column){

        string binaryHex1 = bitset<8>(rM.Msg.DATA[0]).to_string() +
                            bitset<8>(rM.Msg.DATA[1]).to_string() +
                            bitset<8>(rM.Msg.DATA[2]).to_string() +
                            bitset<8>(rM.Msg.DATA[3]).to_string();

        string binaryHex2 = bitset<8>(rM.Msg.DATA[4]).to_string() +
                            bitset<8>(rM.Msg.DATA[5]).to_string() +
                            bitset<8>(rM.Msg.DATA[6]).to_string() +
                            bitset<8>(rM.Msg.DATA[7]).to_string();

        m[6*row + column] = HexToFloat(binaryHex1);
        m[6*row + column+1] = HexToFloat(binaryHex2);
}


/*>>>>>ACQUIRING DATA<<<<<*/
DWORD ReadRawData(HANDLE &deviceHandle_, int16_t rawData[6], char FTMstatus[16]){
        DWORD ret;

        // Composing and transmitting the write message
        TPCANMsg writeMessage;
        writeMessage = MsgAssembler(STATUS_SG0_SG2_SG4);
        ret = CAN_Write(deviceHandle_,&writeMessage);
        if(ret != 0)
            return ret;
        //PrintMessage(writeMessage);

        // Receiving the response (Status, sg0, sg2 and sg4)
        TPCANRdMsg readMessage;
        ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
        if(ret != 0)
            return ret;
        DataStatusFeeder(readMessage, rawData, FTMstatus);
        //PrintMessage(readMessage);
        // Receiving the response (sg1, sg3 and sg5)
        ret = LINUX_CAN_Read(deviceHandle_,&readMessage);
        if(ret != 0)
            return ret;
        DataStatusFeeder(readMessage, rawData);
        //PrintMessage(readMessage);

        return ret;
}

void DataStatusFeeder(const TPCANRdMsg &rM, int16_t rawData[6], char FTMstatus[16]){
        // Status
        string binaryHexstatus = bitset<8>(rM.Msg.DATA[0]).to_string() +
                                 bitset<8>(rM.Msg.DATA[1]).to_string();
        for(int i=0; i<16; ++i)
            FTMstatus[i] = binaryHexstatus.at(i);

        // SG0, SG2 and SG4
        string binaryHexSG0 = bitset<8>(rM.Msg.DATA[2]).to_string() +
                              bitset<8>(rM.Msg.DATA[3]).to_string();
        rawData[0] = HexToDecSigned(binaryHexSG0);
        string binaryHexSG2 = bitset<8>(rM.Msg.DATA[4]).to_string() +
                              bitset<8>(rM.Msg.DATA[5]).to_string();
        rawData[2] = HexToDecSigned(binaryHexSG2);
        string binaryHexSG4 = bitset<8>(rM.Msg.DATA[6]).to_string() +
                              bitset<8>(rM.Msg.DATA[7]).to_string();
        rawData[4] = HexToDecSigned(binaryHexSG4);
}

void DataStatusFeeder(const TPCANRdMsg &rM, int16_t rawData[6]){
        // SG1, SG3 and SG5
        string binaryHexSG1 = bitset<8>(rM.Msg.DATA[0]).to_string() +
                              bitset<8>(rM.Msg.DATA[1]).to_string();
        rawData[1] = HexToDecSigned(binaryHexSG1);
        string binaryHexSG3 = bitset<8>(rM.Msg.DATA[2]).to_string() +
                              bitset<8>(rM.Msg.DATA[3]).to_string();
        rawData[3] = HexToDecSigned(binaryHexSG3);
        string binaryHexSG5 = bitset<8>(rM.Msg.DATA[4]).to_string() +
                              bitset<8>(rM.Msg.DATA[5]).to_string();
        rawData[5] = HexToDecSigned(binaryHexSG5);
}

bool FTMerrorCheck(const char FTMstatus[16]){
        bool ret = true;
        for(int i=0; i<16; ++i){
            if(FTMstatus[i] == '1'){
                cout << "FTM115 STATUS ERROR CODE: " << i << endl;
                PrintFTMerror(i);
                ret = false;
            }
        }
        return ret;
}

void ForceTorqueComputation(float ft[6],const int16_t rawD[6], const int16_t b[6],
                            const float m[36], const uint32_t &CpF, const uint32_t &CpT){
        ft[0]=0;
        ft[1]=0;
        ft[2]=0;
        ft[3]=0;
        ft[4]=0;
        ft[5]=0;
        for(int row=0; row<6; ++row){
            for(int col=0; col<6; ++col){
            //these are forces and torques AROUND THE AXIS OF THE SENSOR
            //(x is rotated of 60 deg counter clock wise with respect to
            //the end-effector axis)
                if(row<3)
                    ft[row] += (m[6*row + col] * (rawD[col] - b[col]))/CpF;
                else
                    ft[row] += (m[6*row + col] * (rawD[col] - b[col]))/CpT;
            }
        }
}

void BiasCorrector(int16_t bias[6], int16_t newbias[6]){
        for(int i=0; i<6; ++i){
            bias[i] = (bias[i]+newbias[i])/2;
        }
}

void VelComputation(float linVel[3],float angComp[3],float ft[6],float angErr[3]){
        // To compute the linear velocities for the end-effector Fx and Fy of the sensor
        //has to be projected along the real end effector axis (x of the sensor is roteted
        //counter clock wise with respect to x of the end-effector)
        //---------LINEAR VELOCITIES
        float cos60 = 0.5;
        float cos30 = sqrt(3)/2;
        float realFx = ft[0]*cos60 - ft[1]*cos30;
        float realFy = ft[0]*cos30 + ft[1]*cos60;

        ft[0] = realFx;
        ft[1] = realFy;

        for(int i=0; i<3; ++i){

            linVel[i] = 3*(ft[i]/1000.0);

            if(abs(ft[i])<2)
                linVel[i] = 0;
            else if(linVel[i]>0.2)
                linVel[i] = 0.2;
        }

        //---------ANGULAR COMPENSATIONS
        //ROLL
        if(angErr[0]>0.01)
            angComp[0] = -0.03;
        else if(angErr[0]<-0.01)
            angComp[0] = 0.03;
        else
            angComp[0] = 0.0;
        //PITCH  (for some reason pitch works in the opposite way)
        if(angErr[1]>0.01)
            angComp[1] = 0.03;
        else if(angErr[1]<-0.01)
            angComp[1] = -0.03;
        else
            angComp[1] = 0.0;
        //YAW
        if(angErr[2]>0.01)
            angComp[2] = -0.03;
    else if(angErr[2]<-0.01)
        angComp[2] = 0.03;
    else
        angComp[2] = 0.0;

}


float YawComputation(const float yawTorque){
    float yawVel=0;
    float max=1;
    float velSat=0.1;
    if(abs(yawTorque)<0.05)
        yawVel=0;
    else if(yawTorque>max)
        yawVel = 0.1;
    else if(yawTorque<-max)
        yawVel = -0.1;
    else
        yawVel = (velSat/max)*yawTorque;

    return yawVel;
}

//------USEFULL FUNCTIONS------

TPCANMsg MsgAssembler(const DWORD id, const BYTE len, const BYTE data){
    TPCANMsg msg;
    msg.ID = id;
    msg.MSGTYPE = MSGTYPE_STANDARD;
    msg.LEN = len;
    msg.DATA[0] = data;

    return msg;
}

TPCANMsg MsgAssembler(const DWORD id){
    TPCANMsg msg;
    msg.ID = id;
    msg.MSGTYPE = MSGTYPE_STANDARD;
    msg.LEN = 0;
    return msg;
}

void Sleep(const int &millisec){
    struct timespec req = {millisec/1000, millisec%1000 * 1000000L};
    //double sec = millisec/1000;
    //cout << "Sleeping for " << sec << " seconds" << endl;
    nanosleep(&req,NULL);
}

void PrintMessage(const TPCANMsg &msg){
    cout << left << setw(10) << "Tx" << "0x" << setw(10) << hex << msg.ID <<
    setw(10) << (msg.LEN + 0);
    if(msg.LEN != 0){
        for(int i=0; i<msg.LEN; ++i)
            cout << left << setw(5) << hex << (msg.DATA[i]+0);
    }
    cout << endl;
```

```
}

void PrintMessage(const TPCANRdMsg &rdMsg){
    TPCANMsg msg = rdMsg.Msg;
    cout << left << setw(10) << "Rx" << "0x" << setw(10) << hex << msg.ID << setw(10) <<
            (msg.LEN + 0);
    if(msg.LEN != 0){
        for(int i=0; i<msg.LEN; ++i)
            cout << left << setw(5) << hex << uppercase << (msg.DATA[i]+0);
    }
    cout << endl;
}

void PrintHeader(){
    cout << left << setw(10) << "Tx/Rx" << left << setw(12) << "ID" << setw(10) <<
    "DLC" << setw(5) << "D0" << setw(5) << "D1"<< setw(5) << "D2"<< setw(5) << "D3"<<
    setw(5) << "D4"<< setw(5) << "D5"<< setw(5) << "D6" << setw(5) << "D7"<< endl;
}

void PrintMatrix(float m[36]){
    cout << endl << "------Matrix of Coefficients------" << endl;
    cout << right << setw(14) << "G0" << setw(12) << "G1"
                  << setw(12) << "G2" << setw(12) << "G3"
                  << setw(12) << "G4" << setw(12) << "G5";

    cout << fixed << setprecision(2) << endl;
        for(int row=0; row<6 ; ++row){
            cout << left;
            if(row == 0)
                cout << setw(4) << "Fx";
            else if(row == 1)
                cout << setw(4) << "Fy";
            else if(row == 2)
                cout << setw(4) << "Fz";
            else if(row == 3)
                cout << setw(4) << "Tx";
            else if(row == 4)
                cout << setw(4) << "Ty";
            else if(row == 5)
                cout << setw(4) << "Tz";

            for(int col=0; col<6; ++col){
                cout << right << setw(12) << m[6*row +col] ;
            }
            cout << endl;
        }

}

void PrintFTMerror(const int &code){

    switch(code){
        case WATCHDOG_RESET:
            cout << "Error description: Watchdog Reset" << endl;
            break;
        case DAC_ADC_TOO_HIGH:
            cout << "Error description: DAC/ADC check result too high (CRITICAL)" << endl;
            break;
        case DAC_ADC_TOO_LOW:
            cout << "Error description: DAC/ADC check result too low (CRITICAL)" << endl;
            break;
        case ANALOG_GROUND_OOF:
            cout << "Error description: Artificial analog ground out of range
                    (CRITICAL)" << endl;
            break;
        case POWER_SUPPLY_TOO_HIGH:
            cout << "Error description: Power supply too high (CRITICAL)" << endl;
            break;
        case POWER_SUPPLY_TOO_LOW:
            cout << "Error description: Power supply too low (CRITICAL)" << endl;
            break;
        case BAD_ACTIVE_CALIBRATION:
            cout << "Error description: Bad active calibration (CRITICAL)" << endl;
            break;
        case EEPROM_FAILURE:
            cout << "Error description: EEPROM failure (CRITICAL)" << endl;
            break;
        case CONFIGURATION_INVALID:
            cout << "Error description: Configuration invalid" << endl;
            break;
        case RESERVED_1:
        case RESERVED_2:
        case RESERVED_3:
            cout << "Error description: Reserved" << endl;
            break;
        case TEMPERATURE_TOO_HIGH:
            cout << "Error description: Sensor temperature too high (CRITICAL)" << endl;
            break;
        case TEMPERATURE_TOO_LOW:
            cout << "Error description: Sensor temperature too low (CRITICAL)" << endl;
            break;
        case CAN_BUS_ERROR:
            cout << "Error description: CAN bus error" << endl;
            break;
        case ANY_ERROR:
            cout << "Error description: Any error causes this bit to turn on" << endl;
            break;
    }
}

void PrintForceTorque(const float ft[6]){
    cout << setprecision(3);
    for(int i=0; i<6; ++i)
        cout << setw(13) << ft[i];
    cout << endl;
}

void PrintFTHeader(){
    cout << "  " << left << setw(13) << "Fx" << setw(13) << "Fy" << setw(13) << "Fz" <<
    setw(13) << "Tx"<< setw(13) << "Ty"<< setw(13) << "Tz"<< endl;
}

void PrintLinearVel(const float linVel[3]){
    cout << setprecision(3);
        for(int i=0; i<3; ++i)
            cout << setw(13) << linVel[i];
        cout << endl;
}
```

```
//———CONVERSIONS FUNCTIONS———

long long int BinToDec(const string &bin){
    int result = 0;
    for(unsigned int i=0; i<bin.length(); ++i){
        if(bin.at(i) != '0')
            result += pow(2,bin.length()-i-1);
    }
    return result;
}

float HexToFloat(const string& binaryHex){
    string sign = binaryHex.substr(0,1);
    float s = 0;
    string exponent = binaryHex.substr(1,8);
    float e = 0;
    string mantissa = binaryHex.substr(9,23);
    float m = 0;
    //Determining the sign
    (sign == "0")?(s=1):(s=-1);
    //Calculating the exponent
    e = pow(2,BinToDec(exponent)-127);
    //Calculating the mantissa
    for(int i = 0; i<int(mantissa.length()); ++i){
        if(mantissa.at(i) != '0')
            m += pow(2,-i-1);
    }
    m += 1;

    return (s*e*m);
}

int HexToDecSigned(const string &bin){
    int result = 0;
    (bin.at(0) == '1')? result = -32768 : result = 0;
    for(int i=1; i<int(bin.length()); ++i){
        if(bin.at(i) != '0')
            result += pow(2,bin.length()-i-1);
    }
    return result;
}
```

Bibliography

[1] Wigbert Fehse. *Automated rendezvous and docking of spacecraft.* Cambridge University Press, 2003.

[2] Steve Corrigan. *Introduction to the Controller Area Network (CAN).* Texas Instrument, 2008.

[3] Genoa Robotics and Automation Laboratory (GRAAL). *Framework User's Guide.* Version 2.0. 2009.

[4] B. Siciliano et al. *Robotics, Modelling, Planning and Control.* Springer, 2009.

[5] *National Aeronautics and Space Administration: On-Orbit Satellite Servicing Study.* 2010. URL: `https://sspd.gsfc.nasa.gov/images/nasa_%20satellite%20servicing_project_report_0511.pdf` (visited on 05/15/2020).

[6] Peng Zhang. *Advanced Industrial Control Technology.* Elsevier, 2010.

[7] ATI Industrial Automation. *Schunk Light Weight Arm F/T Integration. Installation and Operation Manual.* 2011.

[8] PEAK-System Technik GmbH. *PCAN Driver for Linux. User Manual.* Version 7.1. 2011.

[9] Sudhakar Maradana. *CAN Basics.* 2012. URL: `https://automotivetechis.wordpress.com/2012/06/01/can-basics-faq/` (visited on 03/18/2020).

[10] R. Toulson and T. Wilmshurst. *Fast and Effective Embedded Systems Design.* Second Edition. Elsevier, 2012.

[11] PEAK-System Technik GmbH. *PCAN-PCI. PCI to CAN Interface.* Version 2.2.1. 2013.

[12] Wolfhard Lawrenz. *CAN System Engineering. From Theory to Practical Application.* Ed. by Wolfhard Lawrenz. Second Edition. Springer, 2013.

[13] Curtis Bradley. "Robotic Arm Calibration and Control. 6-DOF Powerball LWA 4P". In: (2014).

[14] Genoa Robotics and Automation Laboratory (GRAAL). *Schunk Powerball Manipulator. Software Guide.* Version 1.0. 2014.

[15] Kevin M. Lynch and Frank C. Park. *Modern Robotics. Mechanics, Planning, and Control*. Cambridge University Press, 2017.

[16] Tharek Mohtar. *Design, modeling, and testing of a space docking mechanism for cooperative on-orbit servicing*. Politecnico di Torino, 2017.

[17] SCHUNK GmbH & Co.KG. *Assembly and Operating Manual LWA 4P. Powerball lightweight arm*. Version 4. 2018.