POLITECNICO DI TORINO

MSc Degree in Electronic Engineering

Master's Thesis

Development of the control system for an Inflatable Robot

Supervisor
Prof. Stefano Mauro
Ing. Matteo Gaidano
Ing. Pierpaolo Palmeri
Ing. Mario Troise

Candidate
Francesco Gambino

Academic Year 2021-2022

A mia madre

Abstract

Since the advent of robotics, manipulators have been made of rigid materials, and most developments in the field have pursued increasingly precise and dynamic robots. However, in recent years there has been a growing interest in soft robotics, which has led to the development of new mechanical structures based on flexible and deformable elements. Thanks to these properties, soft robots are able to perform new tasks in new fields of application, ranging from non-invasive surgical techniques to the manipulation of delicate and irregular objects.One of the fields in which soft robots can excel and bring significant benefits is space application: lightweight robotic systems and the use of flexible and deformable elements allow the manipulator to be easily contained in a small package and deployed when required. Such approaches also offer an increase of the payload-to-weight ratio, making it possible to reduce the cost of getting a manipulator into orbit or onto the surface of another celestial body such as the moon or Mars. Despite the promising properties, there are particular criticalities that make the development of soft manipulators challeging: great attention is paid to the study of innovative materials to create flexible structures and actuators or to new control approach to such structures. Indeed, both the modelling and control of rigid robotics cannot be applied successfully to soft robots because of their highly non-linear dynamics. The goal of this thesis is to develop a control system for the 3-degree-of-freedom manipulator, the POPUP Robot, whose mechanical structure has been designed and developed by the research group of Professor Stefano Mauro. The manipulator is an anthropomorphic arm with electrical actuation, where the typical rigid link have been replaced by inflatable cylinders, made out of PVC. It therefore presents complex kinematic and dynamics equations that are difficult to obtain analytically. The proposed system is able to calculate the amount of deformation of the inflatable links by processing the measurements of a set of sensors opportunely placed along the robot structure. The designed control algorithm is able to counteract the deformation both in steady-state conditions and during the execution of planned trajectories.

Acknowledgements

Contents

List of Tables

List of Figures

Part I

First Part

Chapter 1

Introduction

1.1 Objectives

The branch of soft robotics has been developing for about 15 years with the aim of simulating the behaviour and properties of the handling and locomotion structures typical of biological beings. Rigid robots allow high precision and high force exertion, but they are heavy and take up a certain volume of space even when not in use. This makes rigid manipulators unsuitable for some particular applications, such as space: the volume occupied and the weight are critical characteristics that, in general, want to be minimised to reduce mission costs.

Soft robots provide adaptability, compliance and low mass at the cost of lower force output and control difficulties. A particular family of soft robots are inflatable robots. These are made of soft material that allow the structure to be folded when not in use, so that it can be easily carried in compact, lightweight packages and then inflated for deployment. Space applications are not new to inflatable structures: several examples can be given, such as the Mars Pathfinder Inflatable Air-Landing Systems,the Inflatable Solar Array, the Inflatable Antenna Experiment or the Inflatable Life Support Module docked on the ISS.[2]

This thesis aims to develop a control system for an inflatable, lightweight and large manipulator for space application. The main goal is to achieve performance that are comparable with those of rigid robots while keeping the algorithm computationally efficient so that it can be run on a microcontroller. Initially a simple controller is developed, which does not take into account the typical deformations of inflatable structures. Then sensors are implemented on the structure of the manipulator. A bend estimator is designed to fuse the data coming from different sensors. It will return the bending of the links, which information is used by the developed control strategy to correct the pose of the manipulator considering the deformation of the links.

1.2 Organization of the Thesis

This is a brief overview of the parts and chapters content.
The first part present all the theoretical consideration:
Chapter 1 explains the thesis' objectives and summarize of the used methodologies.
Chapter 2 presents the soft robots used during the thesis development, with a detailed explanation of the fabrication procedure and the already existent models.
Chapter 3 offers an overview about the control problem and the previous developed control scheme. Then the Corrected Inverse Kinematic controller is discussed.

In the second part the proposed electronic system is presented:
In *Chapter 4* a complete overview of the system is offer.
Chapter 5 presents the selected actuators,sensors and communication protocols.
Chapter 6 highlights the Main Controller subsystem and its development.
Chapter 7 discuss about the Link board subsystem.

In the third part the practical implementation is shown:
Chapter 8 is about the systems integration and calibration, testing procedure and obtained results.
Chapter 9 concludes the thesis with personal consideration and pose the basis for future work.

Chapter 2

Popup Robot

2.1 Mechanical Structure

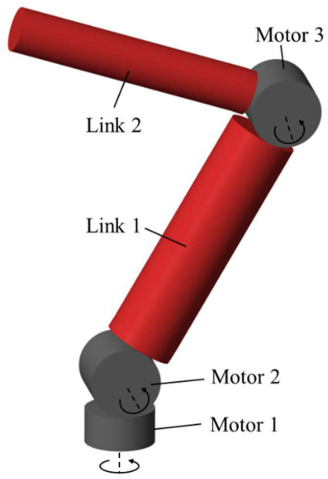

Figure 2.1. Representation of the inflatable robot concept

Popup robot is a novel and lightweight manipulator concept designed by Ing. Pierpaolo Palmeri as his PhD Thesis. It consist in an anthropomorphic arm composed by two inflatable links and three rigid and motorized joints. The structure is shown in Fig. 2.1. The inflatable links have cylindrical shape and are made out PVC fixed to a 3D printed support which connect the link to the actuator. Another fundamental component of the mechanical design is the Pneumatic line: it allows the links to be inflated and deflated, providing the necessary pressure which is in the range of 10-60 kPa. It consist in a pressurized tank made out of composite material, in order to reduce its weight. A reducing valve and two digital valves for each link. (fig pressurized tank + valve). The valves are Normally closed (NC) and are selected of small dimension because the inflation or deflation stages is not a requirement. Also, since they works as NC, energy is consumed only during the inflation or deflation procedure.

The deployment capabilities of the system allows the robot to be stored in a small box. The following stages highlights the necessary procedure to deploy the manipulator. Notice that the concept try to use the same motors used for the motion of the link as winding or unwinding actuator. This is possible thanks to the particular structure that has been designed for each link.

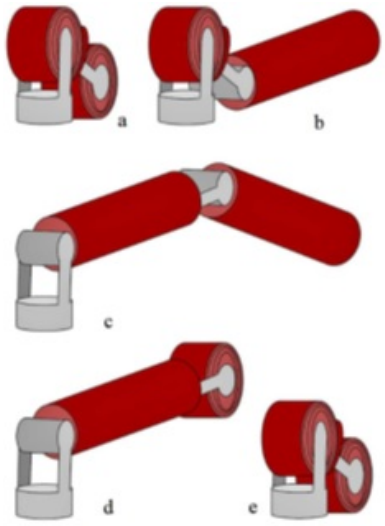

Figure 2.2. Scheme of the stages for the deployment and withdrawing phases

- In the starting configuration (Fig. 3a), link 1 and link 2 are deflated and wound around the shafts of joint 2 and joint 3, respectively.
- The link 2 is unrolled through the action of the motor of the joint 3, and it is inflated with the air supply, activating the valve V2in, to assume the deployed form (Fig. 3b).
- Subsequently, the link 1 is unwound utilizing the motor 2 and inflated (Fig. 3c), commuting the valve V1in.

After the deployment phase, the robot reaches its working configuration (Fig. 3c). When the withdrawing of the robot is necessary, the following steps are expected:

- The link 2 is deflated, through the commutation of the valve V2out, and rolled around the shaft of the joint by using the motor 3 (Fig. 3d).
- Then, the link 1 is deflated activating the valve V1out, and rolled around the shaft through the motor 2. (Fig. 3e) The robot comes back to its starting configuration and can be stored in the box. [10]

2.2 Previous developed models

As discussed in the introduction, one of the main issue while working with soft robots is the definition of a precise model, due to the natural non-linearity of this structures. Two different approaches have been tested with the aim of reproducing the static and dynamic characteristics of the robot through a mathematical model.

2.2.1 Rigid Link Modeling

The first model simply consider each link as rigid, therefore the forward kinematic of the manipulator can be derived starting from the Denavit-Hartenberg parameters. Table 2.1 reports the computed DH parameters and Fig 2.3 shows the resulting kinematic scheme represented using the Robotic Toolbox in MATLAB.

Link	θ_i	d_i	a_i	α_i
1	θ_1	0.15	0	$\pi/2$
2	θ_2	0.05	0.745	0
3	θ_3	-0.12	0.685	0

Table 2.1. Rigid Robot Denavit-Hartenberg parameters.

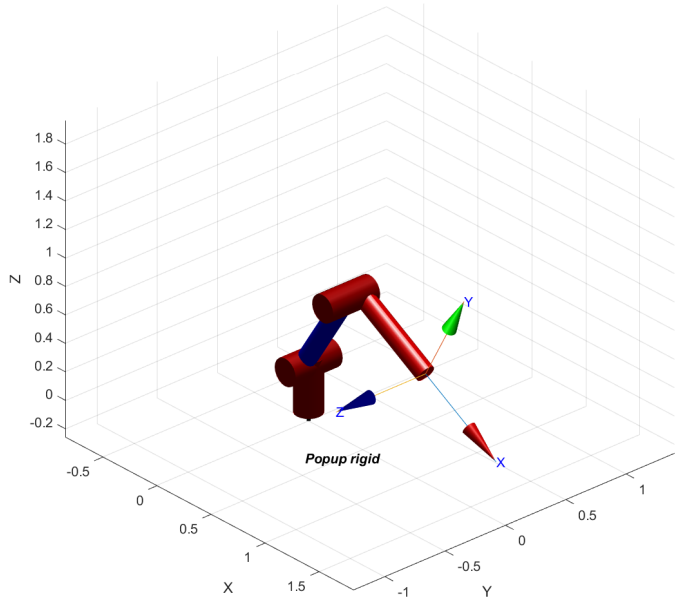

Figure 2.3. Kinematic scheme considering Rigid Links.

From the DH Parameters is possible to easily derive an homogeneous transformation matrix which represent the forward kinematic of an anthropomorphic arm, so the functional relationship between the joint variables and the end-effector position and orientation.

$$T_0^3(q) = \begin{bmatrix} c_{123} - c_1 s_{23} & -c_{12}s_3 - c_{13}s_2 & s_1 & 0.745c_{12} - 0.07s_1 - 0.685c_{123} - 0.685c_1 s_{23} \\ c_{23}s_1 - s_{123} & -c_2 s_{13} - c_3 s_{12} & -c_1 & 0.07c_1 + 0.745c_2 s_1 - 0.685s_{123} - 0.685c_{23}s_1 \\ c_2 s_3 + c_3 s_2 & c_{23} - s_{23} & 0 & 0.745s_2 + 0.685c_2 s_3 + 0.685c_3 s_2 + 0.15 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the last column, the forward kinematic equation can be rewritten as

$$\begin{cases} p_x = 0.745cos(q1+q2) - 0.07sin(q1) - 0.685cos(q1+q2+q3) - 0.685cos(q1)sin(q2+q3) \\ p_y = 0.745cos(q2)sin(q1) + 0.07cos(q1) - 0.685sin(q1+q2+q3) - 0.685cos(q2+q3)sin(q1) \\ p_z = 0.745sin(q2) + 0.685cos(q2)sin(q3) + 0.685cos(q3)sin(q2) + 0.15 \end{cases} . \tag{2.1}$$

To going further in the modelling of a rigid link is necessary to consider the Inverse Kinematic problem, which consist of the determination of the joint variable from a given end-effector (EE from now on) cartesian position. The solution of the IK problem cover a fundamental aspect because even if the control strategy act always on the joint space, is typically preferable to giving Cartesian input, regarding the position, and using one of the possible representation to indicate the wanted orientation of the EE.

Considering the manipulator as a rigid one, the IK problem should recall the one of a simple 3-DOF Anthropomorphic arm, largely discussed in the bibliography [1]. However, some difference are present between the ideal structure and the structure of the Popup robot. In particular, an offset is present between the base reference frame and the EE frame. This offset, reported in table 2.1 , make the analytical computation of an IK function much difficult. For this reason I reconsidered the DH parameter in order to eliminate such offset. First, the base frame and the one of the first link has been moved to the intersection between the first and the second joint. In this way the offset d_1 has been eliminated. Then, the other two offset are merged together to form a single offset between the second and the third joint. The following figure shows the resulting kinematic scheme and the table reports the simplified DH parameters.

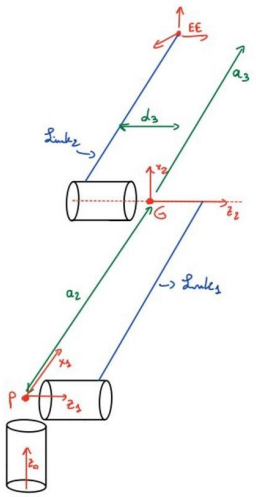

Figure 2.4. Simplified Kinematic Scheme of Popup Robot.

Link	θ_i	d_i	a_i	α_i
1	θ_1	0.0	0	$\pi/2$
2	θ_2	0.0	0.745	0
3	θ_3	-0.07	0.685	0

Table 2.2. Simplified Rigid Robot Denavit-Hartenberg parameters.

With this configuration it was easy to compute an analytical expression for solving the IK problem. Starting from the expression of the forward kinematic, the cosine of the third joint variable can be expressed as

$$c_3 = \frac{X^2 + Y^2 + Z^2 - L_1^2 - L_2^2 - OFF^2}{2L_1L_2} \tag{2.2}$$

$$s_3 = \pm\sqrt{1 - c_3^2} \tag{2.3}$$

and thus

$$\theta_3 = Atan2(s_3, c_3) \tag{2.4}$$

giving two solutions according to the sign of s_3.

For the joint 2 there are four possible solution, according to the sign of s_3:

$$\theta_{2,1} = Atan2\Big((L_1 + L_2c_3)Z - L_2s_{3p}\sqrt{(X^2 + Y^2 - OFF^2)}, \\ (L_1 + L_2c_3) * \sqrt{(X^2 + Y^2 - OFF^2)} + L_2s_{3p}Z\Big) \quad (2.5)$$

$$\theta_{2,2} = Atan2\Big((L_1 + L_2c_3)Z + L_2s_{3p}\sqrt{(X^2 + Y^2 - OFF^2)}, \\ -(L_1 + L_2 * c_3)\sqrt{(X^2 + Y^2 - OFF^2)} + L_2s_{3p}Z\Big) \quad (2.6)$$

$$\theta_{2,3} = Atan2\Big((L_1 + L_2c_3)Z - L_2s_{3m}\sqrt{(X^2 + Y^2 - OFF^2)}, \\ (L_1 + L_2c_3)\sqrt{(X^2 + Y^2 - OFF^2)} + L_2s_{3m}Z\Big) \quad (2.7)$$

$$\theta_{2,4} = Atan2\Big((L_1 + L_2c_3)Z + L_2s_{3m}\sqrt{(X^2 + Y^2 - OFF^2)}, \\ -(L_1 + L_2c_3)\sqrt{(X^2 + Y^2 - OFF^2)} + L_2s_{3m}Z) \quad (2.8)$$

Finally, for the joint 1 other two solutions exist. From

$$c_1 = (X/K + (YOFF)/K^2)/(1 + (OFF^2)/K^2); \quad (2.9)$$

$$s_1 = \sqrt{1 - c_1^2} \quad (2.10)$$

where

$$K = \sqrt{X^2 + Y^2 - OFF^2}; \quad (2.11)$$

Once solved, the two possible solution are

$$\theta_{1,1} = Atan2(s_1, c_1) \quad (2.12)$$

$$\theta_{1,2} = Atan2(-s_1, -c_1) \quad (2.13)$$

So there exist four possible solutions for a given wrist position, which are illustrated in the Fig

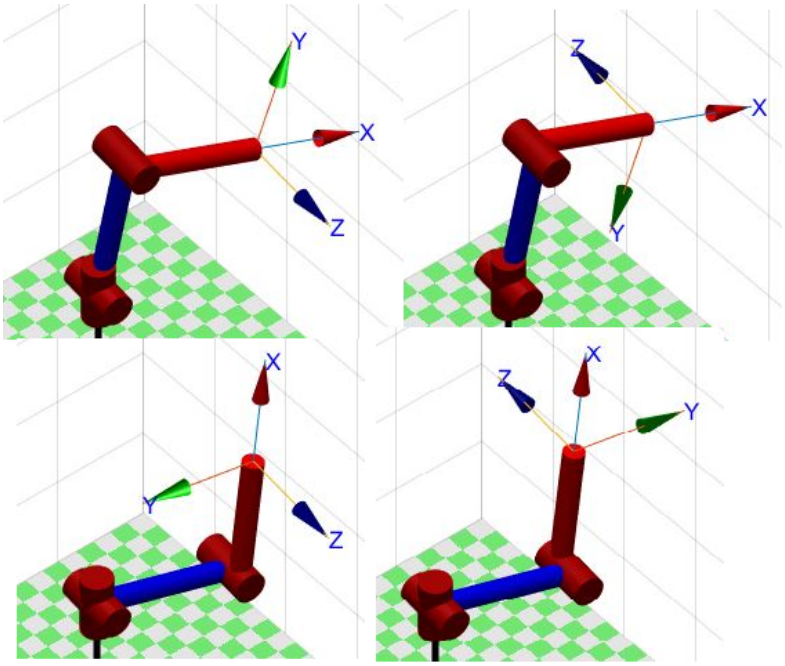

Figure 2.5. The four configuration of the arm for a given wrist position

2.2.2 Elastostatic Modeling

The elastostatic approach consider the links as Pseudo-rigid bodies, each composed by two rigid bodies with length l1 and l2, connected by an hinge with a torsional spring whose behaviour is described by the Hooke's law:

$$\kappa\theta = \tau$$

where κ is the spring stiffness, θ the angular deflection and τ the reaction torque.

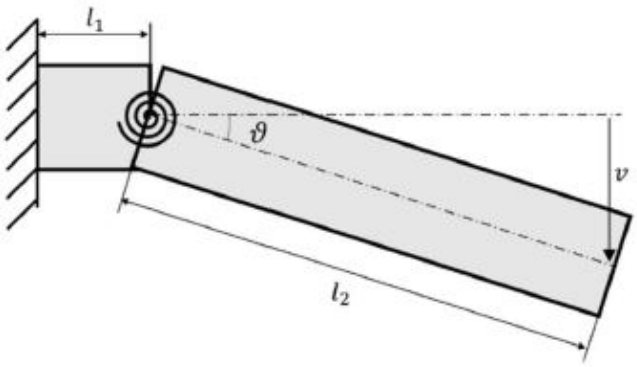

Figure 2.6. 2D scheme of an Inflatable link with pseudo-rigid body model applied.

This model correctly approximate the behaviour of an inflated link as soon as the air pressure is maintained at a certain level. When the deflection of the link increase without an increase of the reaction moment, wrinkling occurs. The Wrinkling moment is defined as the bending load that causes the first Wrinkle appear.
As suggested by previous work[ref][ref], the formulation

$$Mw = (\frac{\pi}{4})\pi p r^3 \tag{2.14}$$

can be used to estimate the wrinkling moment starting from the information about the link pressure.

Therefore the kinematic scheme of the robotic arm must be update considering two virtual torsional springs for each link. The robotic arm reaches 7 degrees of freedom, 3 for the actuated joint and 2 for each link introduced by the virtual spring. The new kinematic model is shown in Fig 2.7 and the newly computed DH parameter are listed in Table 2.3. [13]

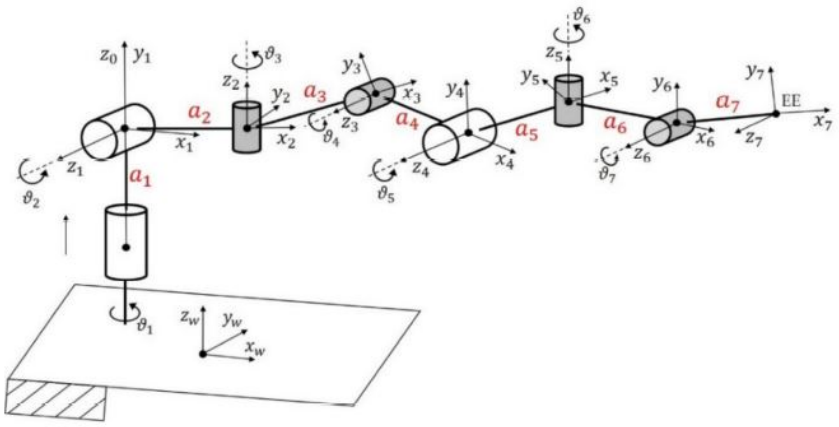

Figure 2.7. Kinematic scheme with additional degrees of freedom.

Link	Ai	alfai	Di	Thetai
1	0	$pi/2$	0	$theta1$
2	$l1$	$-pi/2$	0	$theta2$
3	0	$pi/2$	0	$theta3$
4	$l2$	0	0	$theta4$
5	$l1$	$-pi/2$	0	$theta5$
6	0	$pi/2$	0	$theta6$
7	$l2$	0	0	$theta7$

Table 2.3. Pseudo-Rigid Robot Denavit-Hartenberg parameters.

Chapter 3

The Control Problem

In order to execute a commanded task satisfying transient and stationary requirements, it is necessary to define the time history of the control torques to be applied to the joint motors. The technique used to control the manipulator plays a fundamental role in the final performance and imposes further limitations that result in particular hardware and software implementations. Two kinds of general control schemes might be considered: a joint space control scheme and a operational space control scheme. The joint space control scheme problem consist in solving the Inverse Kinematic problem for a wanted Cartesian target. The actual joint controller is designed so that the actuator tracks the reference input. The operational space control requires an higher computational load: the inverse kinematic algorithm is embedded in the control loop and must be executed continuously [1]. The following section firstly presents a model-based control scheme that has been previously designed to counteract the deformation of the links, then discuss about the control strategy implemented in the context of this thesis work.

3.1 Elastostatic Inverse Kinematics Controller

This approach is based on a recursive algorithm developed by Ing. Mario Troise. The following figure show the control scheme: first, the motor joint variables q_m are calculated to be used for the motor control; then, the virtual spring joint variables q_k are estimated, given the external load γ_e and the virtual spring stiffness k, though a recurrent algorithm, based on the same electrostatic approach previously described. This algorithm makes the robot capable of reaching a point in the workspace while compensating the deformation of the link, and therefore the EE position, starting from a known value of the link stiffness constant K.

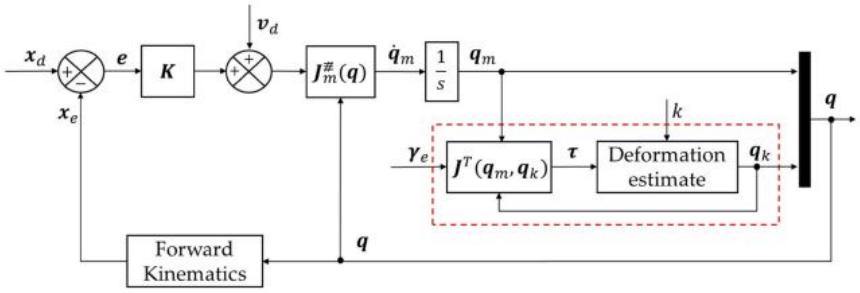

Figure 3.1. Kinematic scheme with additional degrees of freedom.

Experimental tests have been carried out to retrieve the stiffness k. Using a manipulator from Universal Robots, the UR5, in conjunction with a force sensor positioned on the EE of the UR5 it was possible to draw different characteristic which relates the applied torque with the bending of the link, such ones shown in the following pictures.

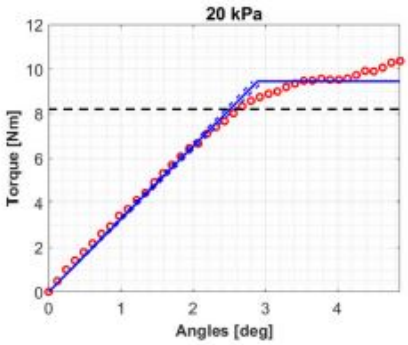

Figure 3.2. Experimental data and linear model

From the results an estimate of average stiffness K has been retrieved. However experimental data have not shown a clear correlation between stiffness and link pressure. On the contrary, a clear relation between the inflation pressure and the maximum wrinkling moment has been found. Results demonstrate that the link characteristic can be approximated as linear function when the applied torque is limited. Equation 2.14 is therefore able to predict the wrinking moment and, thus, the range in which wrinkles does not occurs, leading to a estimated maximum payload of 2 kg when inflated at 30 kPa [13].

3.2 Decentralized Controller

A decentralized controller is considered as the simplest control strategy. The manipulator is formed by n independent systems (the n joints) and controls each joint as a SISO system. This simple strategy can be successfully implemented when the system dynamic is linearized by the presence of reduction gear of high ratios, at the price of elasticity and backlash that limit the system performance.[CIT] This is not the case of the POPUP robot, whose elasticity of links cause the dynamic of the manipulator to be highly non-linear. However, more advanced control strategy needs the knowledge of the dynamic, that, as the previous work demonstrated, it result difficult to estimate. For this reason, initially the control problem has been face considering the manipulator as a rigid anthropomorphic arm actuated by electric motor with reduction gears, following the kinematic analysis developed in the previous chapter. A simple control strategy acting in the joint space has been developed.

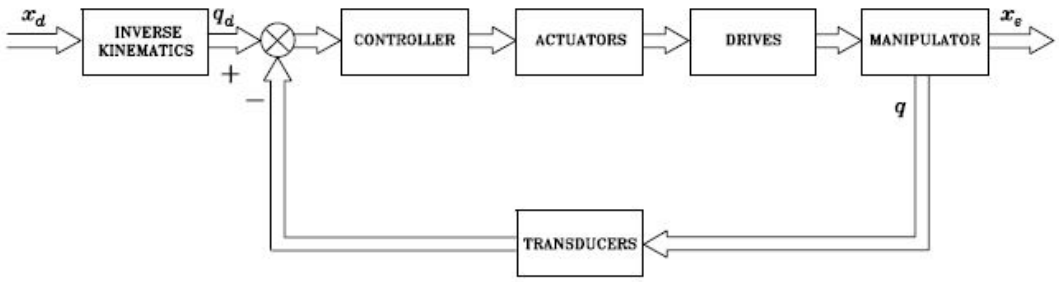

Figure 3.3. General Scheme of a joint space control.

As shown in Fig 3.3 , the control problem can be divided into two sub problems. First, the inverse kinematics of the manipulator must be computed to transform the motion requirement in the operational space into the corresponding motion in the joint space. Then, the controller itself generate the control command so that the actuator execute the required motion. The general scheme of the controller is presented in Fig 3.4.

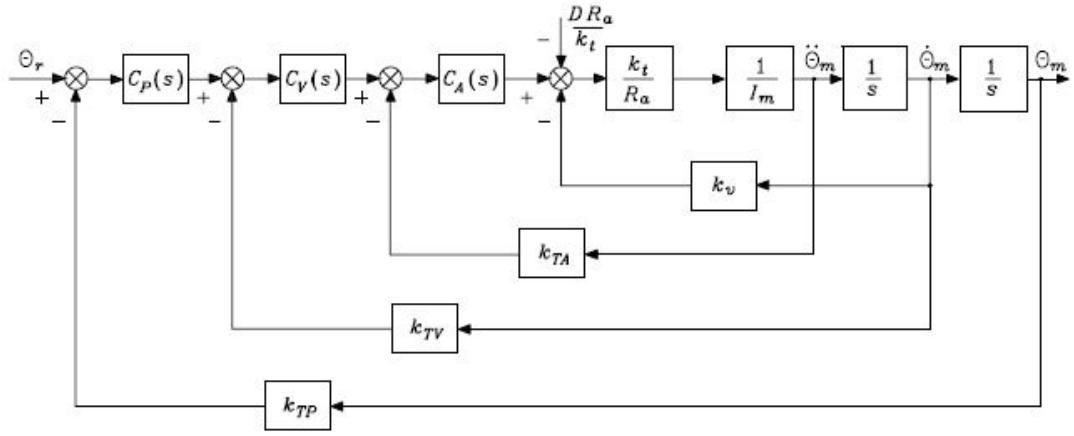

Figure 3.4. General Scheme of a joint space control for a single joint.

C_p, C_v, C_a respectively represent position, velocity and torque controller. The presence of three feedback loop allow to have the complete control about the dynamic of the actuator, having the possibility to limit both the velocity and acceleration. The control action can be expressed as

$$C_P(s) = K_P \quad C_V(s) = K_V \quad C_A(s) = K_A \frac{1 + sT_a}{s} \tag{3.1}$$

From these expressions it would be possible to derive the transfer function of the control system. Subsequently, expressions could be derived from the transfer function to assign values to the controller gains. However, these relationships depend on factors such as damping and natural frequency which, again, depend on the structure and dynamics of the manipulator, which is difficult to calculate. On the contrary, the gains of the controllers have been found by means of a trial and error strategy, first using a simulation and then moving to the real system. The simulation has been performed using a model designed in Simulink, as shown in Fig. 3.5.

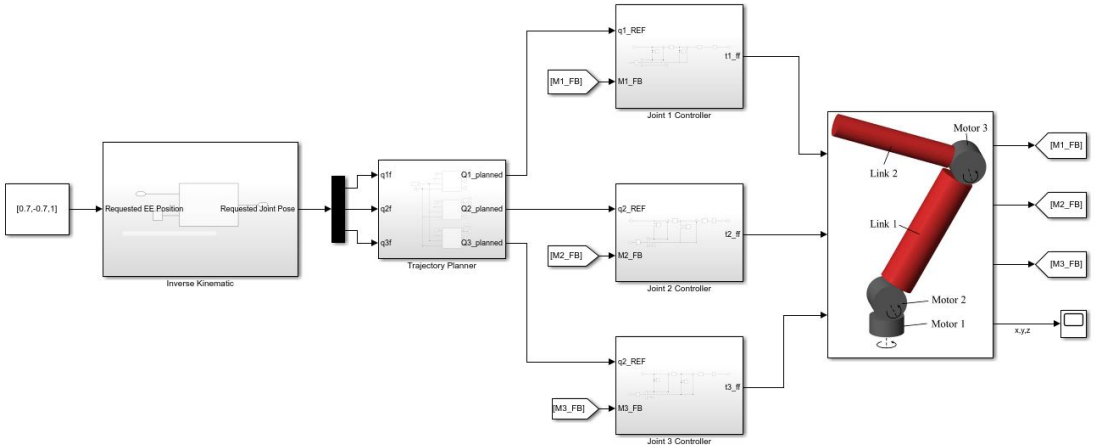

Figure 3.5. General Scheme of the control system developed in Simulink.

Starting from the left, the cartesian target is given. This target is fed into the Inverse Kinematic block, which contains a MATLAB Function block which recall the IK function. The complete MATLAB code is listed in Appendix A. The output is a tuple of joint position targets. This tuple is then used as input for the Trajector Planner block.
Trajectory planning plays a fundamental role in robotics since one of the most common requirements is to move the EE from a pose A to pose B [3]. A trajectory is a path, from A to B, with specified timing. This timing causes the position and orientation to vary smoothly with time, which means that, typically, velocity and acceleration are continuous. A good candidate for such appplication is a polynomial function of time. A fifth-order polynomial can be expressed as

$$S(t) = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F \tag{3.2}$$

The first and second derivatives are also smooth polynomials

$$\dot{S}(t) = 5At^4 + 4Bt^3 + 3Ct^2 + 2Dt + E \tag{3.3}$$

$$\ddot{S}(t) = 20At^3 + 12Bt^2 + 6Ct + 2D \tag{3.4}$$

Applying boundary conditions

$$\dot{q_0} = 0 \tag{3.5}$$

$$\dot{q_f} = 0 \tag{3.6}$$

$$\ddot{q_0} = 0 \tag{3.7}$$

$$\ddot{q_f} = 0 \tag{3.8}$$

to Eq 3.2 to 3.4 gives six equations

$$F = q_0 \tag{3.9}$$

$$E = \dot{q_0} \tag{3.10}$$

$$D = 0.5\ddot{q_0} \tag{3.11}$$

$$C = \frac{1}{2t_f^3}(20(q_f - q_0) - (8\dot{q_f} + 12\dot{q_0})t_f - (3\ddot{q_f} - \ddot{q_0})t_f^2); \tag{3.12}$$

$$B = \frac{1}{2t_f^4}(30(q_f - q_0) + (14\dot{q_f} + 16\dot{q_0})t_f + (3\ddot{q_f} - 2\ddot{q_0})t_f^2); \tag{3.13}$$

$$A = \frac{1}{2t_f^5}(12(q_f - q_0) - 6(\dot{q_f} + \dot{q_0})t_f - (\ddot{q_f} - \ddot{q_0})t_f^2); \tag{3.14}$$

.

Applying the polynomial function to interpolate two configuration q_0 and q_f leads to the set of equations that represent the temporal evolution of the joint variables and their derivatives.

$$q(t) = At^5 + Bt^4 + Ct^3 + Dt^2 + Et + F \tag{3.15}$$

$$\dot{q}(t) = 5At^4 + 4Bt^3 + 3Ct^2 + 2Dt + E \tag{3.16}$$

$$\ddot{q}(t) = 20At^3 + 12Bt^2 + 6Ct + 2D \tag{3.17}$$

The following figure shows an example of the planned variables. The complete MATLAB code is listed in Appendix B.

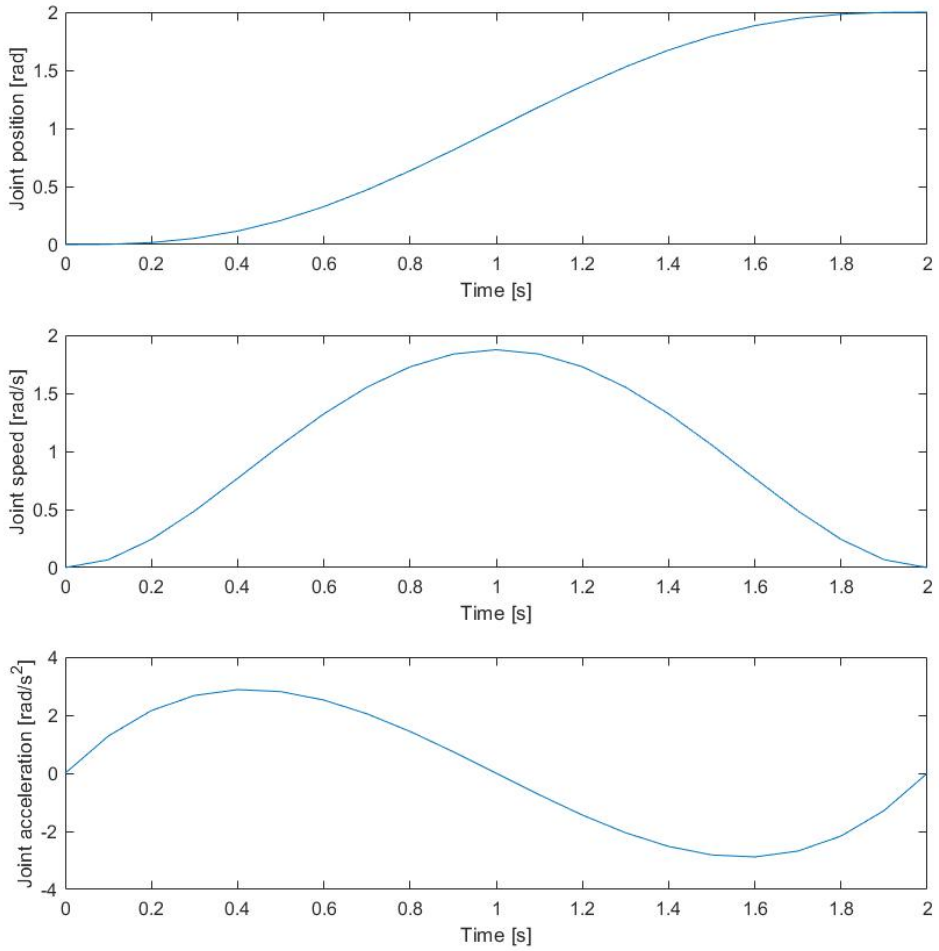

Figure 3.6. Quintic polynomial trajectory.

Going back to the general scheme, the outputs of the Trajector Planner block are passed to the actual controllers, one for each joint. As stated before, the controllers are constituted by two control loop: an inner speed control loop and an outer position control loop as shown in Fig 3.7.

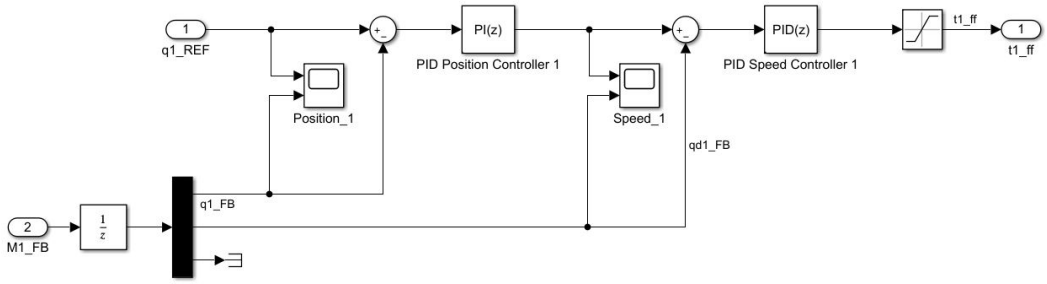

Figure 3.7. Position and Speed control loop.

The torque controller discussed some page before is integrated in the actuator firmware. This is discussed later in Chapter 5. Therefore, the output of the speed control loop represent a torque set point for the most inner control loop.

The last block of the general scheme of Fig 3.5 is a model of the POPUP robot designed using Simscape. Simscape is a Simulink Toolbox with allow to rapidly create models of physical systems. It is largely used to develop control systems and test its performance. The following images briefly resume the main component of the model and then a general view is offered.

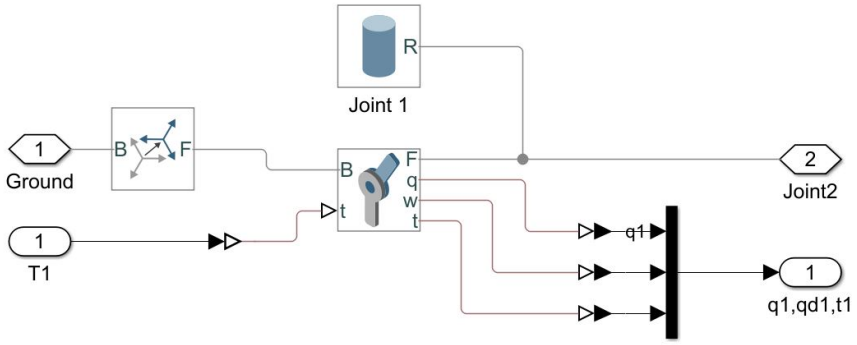

Figure 3.8. Simscape Model of a joint.

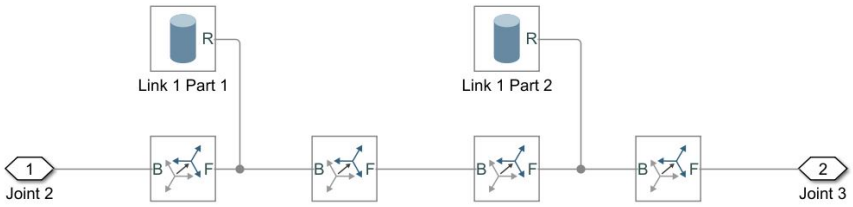

Figure 3.9. Simscape Model of a link.

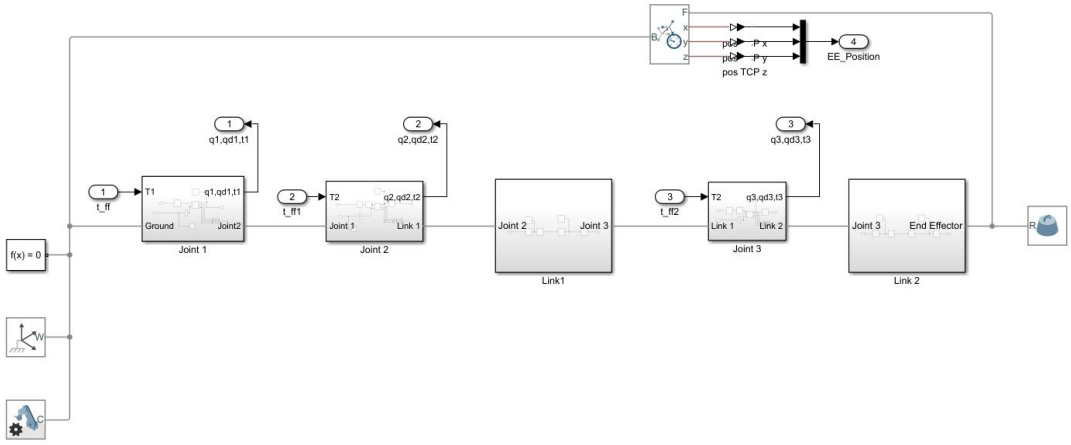

Figure 3.10. Simscape Model of the POPUP Robot.

Using this model allows to find some important starting values for the definition of the gains of the controllers. Indeed, assigning the real physical parameter to joints and links make the model to request real torque values to perform the motion. This real value can be compare to the expected one and to the limit imposed by, for example, the actuator to understand if the control action can be effectively transferred to the real plant.

In the following images are reported the results of the simulation. The cartesian target is set to

$$P_{cartesian} = [0.5, -0.5, 0.7] \tag{3.18}$$

. The IK block gives out three joint angle

$$P_{joint} = [-0.5716, 1.526, -1.817] \tag{3.19}$$

.

which are passed to the Trajectory Planner. The imposed motion time is 10s with a time step of 0.1s. The initial joint position is [0,0,0], which correspond to have the manipulator lying parallel to the X axis. The planned joint variable are show in Fig. 3.11

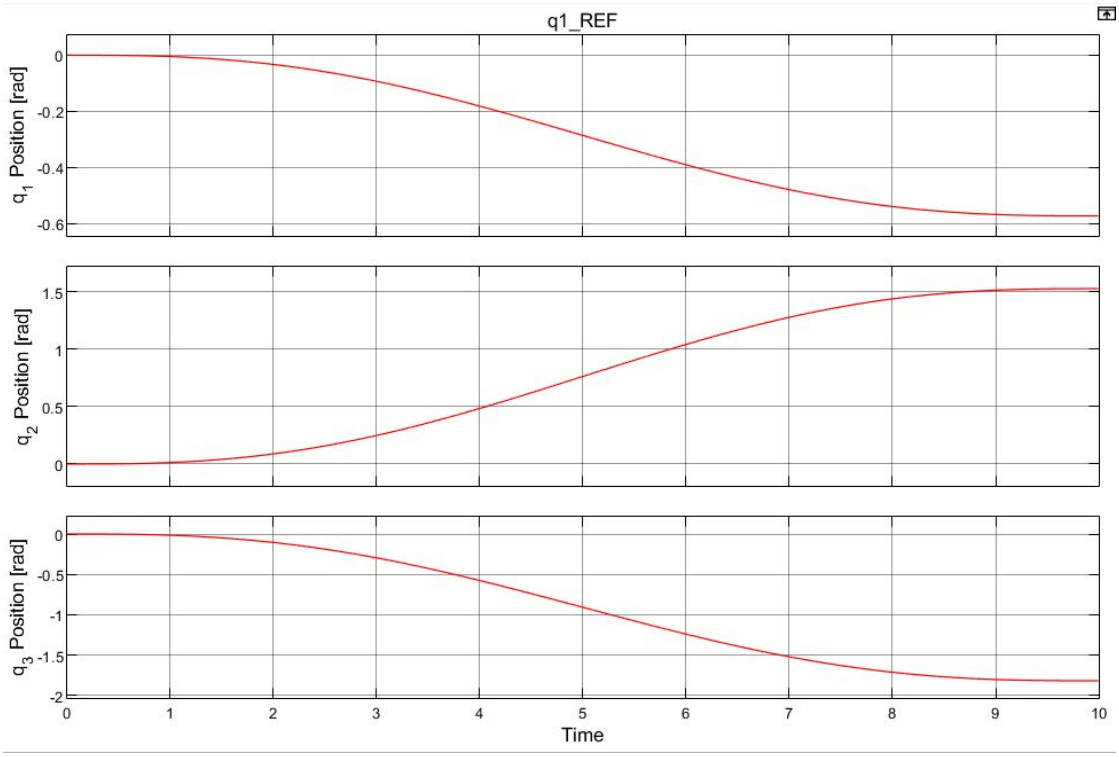

Figure 3.11. Planned joint variables.

This are passed as position reference to the position controllers, which in turns generate the reference for the speed controllers, that run ten time faster than the position controllers. Fig 3.12 to 3.14 shows the position reference and the received feedback from the joint simulated actuator.

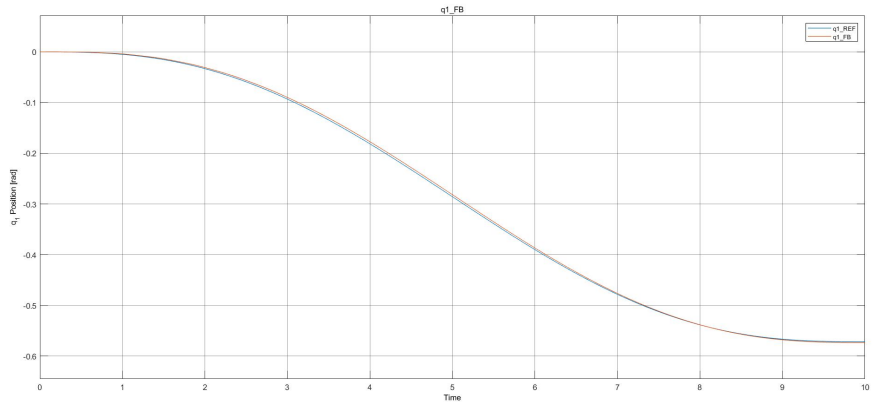

Figure 3.12. Joint 1 reference and feedback.

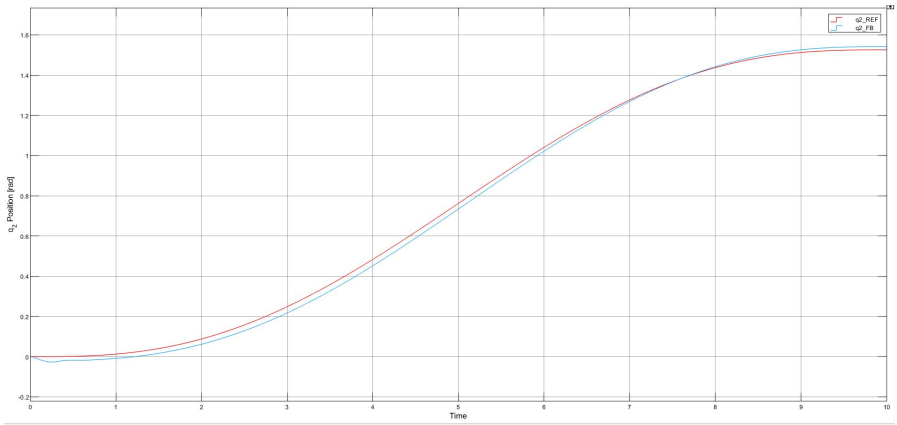

Figure 3.13. Joint 2 reference and feedback.

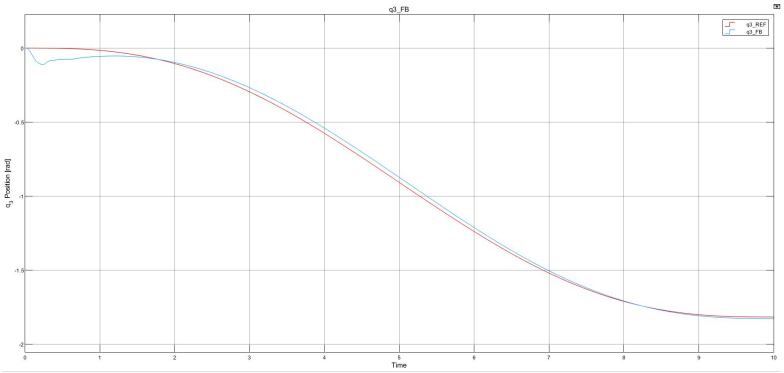

Figure 3.14. Joint 3 reference and feedback.

A simple way to evaluate the performance of the controllers is to look at the positioning error of the EE with respect to the given Cartesian target.

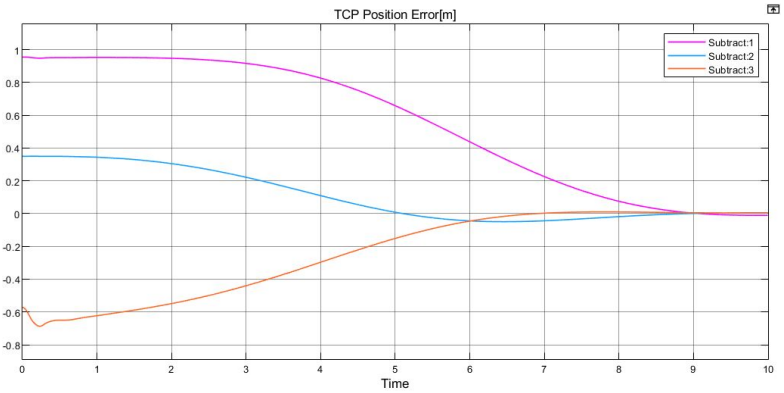

Figure 3.15. EE positioning error.

As expected, the error tend to zero while the motion occur with good smoothness. However, a non-zero error is shown at the end to the simulation. This behaviour was expected as there are small differences between the model developed with Simscape and the geometric measures used in the definition of the Denavit-Hartenberg parameters and, consequently, in the calculation of the inverse kinematics. It would be possible to calibrate the inverse kinematics to cancel this error, but for the sake of brevity this will only be done in the practical application, where the final geometries of the manipulator differ from both those used for DH and those used to develop the simscape model.

3.3 Corrected Decentralized Controller

The previous approaches presents different criticalities to be applied in a practical and efficent control scheme.

Considering the elastostatic approach, the output of the controller relies on the goodness on the estimation of the stiffness constant, which may vary depending on different parameter. The internal pressure plays a fundamental role in the evaluation of the stiffness constant, as has been showed in the characteristic graphs. However, other factor affect the stiffness, most of them are not well measurable. Experimental data shows that the manufacturing procedure might affect the final characteristics: if, for example, to much glue is used during the fabrication, the resulting link will likely be stiff and the point where the wrinkling occurs may vary, leading to an error in the positioning of the EE.

On the contrary, the simple joint space decentralized controller discussed previously does not even take into account the possible deformation that can occurs along the structure of the manipulator. Moreover, no feedback about the position of the EE is provided, leading to significant errors due to multiple source, such as inaccuracies in the computed inverse kinematic, instability in the mechanical structure, backlash in the motor's reduction gears and finally the manipulator's own deformations.

This section discuss the development of a modified version of the decentralized controller in the joint space with the aim of reducing deformation of the links, oscillation of the EE while performing motion and, finally, the positioning error at steady state in presence of different payload. This approach make use of a feedback loop that, thank to a couple of sensors, is able to provide information about the deformation state of the links. The following figure shows a general scheme of the proposed strategy.

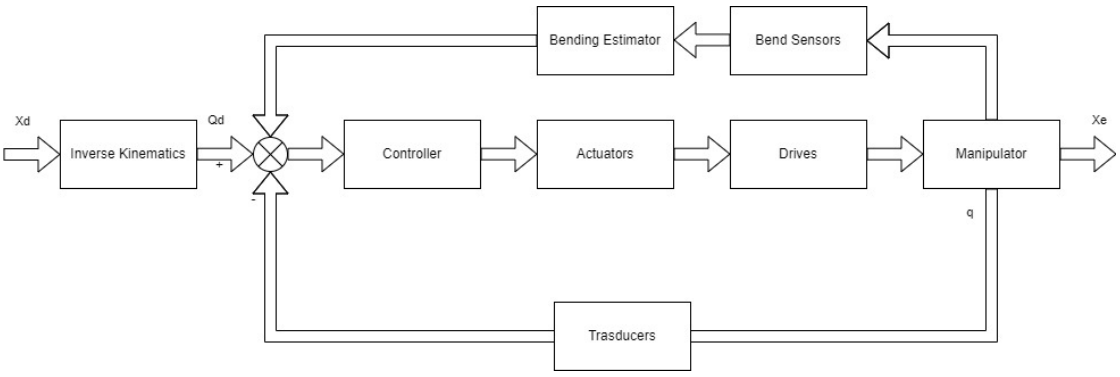

Figure 3.16. General Scheme of Corrected Decentralized Controller

With respect to the simple decentralized strategy discussed previously, looking at the scheme is possible to identify some new block. The bend sensors block recall the set of sensors that have been selected to provide the feedback about the deformation of the link. It plays a role similar to what transducers does for position, velocity and torque feedback. The other block is the Bending Estimator. This block cover a fundamental role. The information retrieved by the bend sensors are fused using a sensor fusion algorithm and subsequently used. The first approach to this problem aim to solve the positioning error caused by a static deformation of the link, as one that may occur when an heavy payload is applied on the EE. Fig. 3.17 shows this situation. In this pose a payload of 1.5Kg causes a static deformation of the link 2. Therefore the resulting EE position is wrong. To correct the EE position the joint angle must be modified to take into account the angular displacement caused by the link deformation. The Cartesian displacement of the EE considering deformation occurring on a single axis can be estimated from the bending measurement and some trigonometry as

$$Displacement_X = L_2 - L_2 cos(\alpha) \tag{3.20}$$

$$Displacement_Z = L_2 sin(\alpha) \tag{3.21}$$

where L_2 is the distance between the wrinkling point and the EE and α is the Bending angle estimated. The correction angle to be sent to the joint actuator can be computed by solving the system of equations composed by 3.20 , 3.21 and

$$\begin{cases} Displacement_X = L - Lcos(\theta) \\ Displacement_Z = Lsin(\theta) \end{cases}$$

where L is distance from the EE to the rotation axis of the actuator and θ is the unknown correction angle. Substituting both equations leads to the system

$$\begin{cases} L_2 - L_2 cos(\alpha) = L - Lcos(\theta) \\ L_2 sin(\alpha) = Lsin(\theta) \end{cases}$$ The solution is immediate

$$cos(\theta) = \frac{L - L_2 + L_2 cos(\alpha)}{L} \tag{3.22}$$

$$sin(\theta) = \frac{L_2 sin(\alpha)}{L} \tag{3.23}$$

$$\theta = atan2(sin(\theta), cos(\theta)) \tag{3.24}$$

This represent the correction angle that can be added or subtracted to the joint variable of the subject link in order to counteract the static deformation and to bring the EE again in position. The actual implementation of this approach is further discussed in the next chapters.

Figure 3.17. Lateral View of POPUP robot subject to deformation

Part II

Second Part

Chapter 4

Electronic System

The electronic system plays a fundamental role in all the aspects of this application. From the sensors conditioning circuits to the power supply, each component must work in a precise way so that the entire system behave as expected. The following sections give an overview of the complete system first, then each subsystem and component is discussed.

4.1 General Logic and Power Scheme

The figure below shows a general, high level, complete scheme of the electronic system.

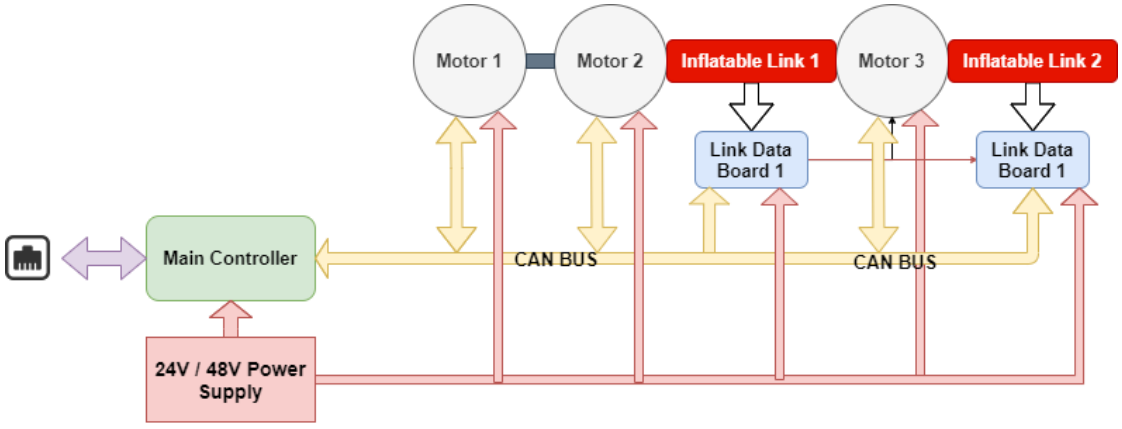

Figure 4.1. General scheme of electronic system.

The main components are:

- Power Supply and Power Bus, in charge of provide and distribute the needed power for the actuators and logic component.
- Main controller, which represent the core component of the electronic system. It manage the communication with the host PC, receive feedback and execute the control strategy discussed in Chapter 3.
- Link Data Board, an electronic subsystem developed to reduce the number of connections between the inflatable link sensors and the base of the manipulator where the main controller resides. It also frees the main controller from certain operations such as the execution of the Madgwick algorithm necessary for retrieve orientation estimate from the IMUs.
- Actuator, used to actuate the mechanical structure with the command received by the main controller.
- Communication Bus, on which command and feedback are sent among the main controller, the motors and the Link Data boards.The CAN protocol ensure reliability and robustness to EMI and noise. Moreover, it also help reducing the number of interconnection along the structure of the manipulator: a 4 poles cable is used both to carry power and control command to the motor and to retrieve data from the LDB.

The following chapter will explain each component more in deep, starting from the chosen Actuators.

Chapter 5

Actuators, sensors and communication protocols

5.1 Actuator

The motion imposed on the joint of a manipulator by the control system is achieved by means of an actuation system, which is generally electrical or hydraulic. The main components that make up an actuation system are:

- power source
- power amplifier
- motor
- transmission device

The choice of these components is based on the specifications imposed by the mechanical structure of the manipulator and by the performance that wants to be obtained. Moreover, for the project discussed in this thesis, the weight of the motor presented a fundamental aspect because a too heavy motor would cause the link deforms even in steady state. The following table resume the specification wanted by the actuator

Nominal Speed	Torque	Weight
$> 50RPM$	$> 30Nm$	$< 1Kg$

Table 5.1. Actuator Specification Requirements

5.1.1 AK80-80 Robotic Actuator

After a selection phase, where different solutions have been explored, the choice fell on the AK80-80 Robotic Actuator.It is an highly integrated, high torque actuator designed with robotic in mind.

The actuator consists of a brushless motor directly connected to a high-precision planetary gearbox with a reduction ratio of 80:1. It also integrates the inverter and the controller that receives and transmits data and commands via the CAN communication protocol.

Figure 5.1. AK8080 front Part.

Figure 5.2. AK8080 planetary gearbox.

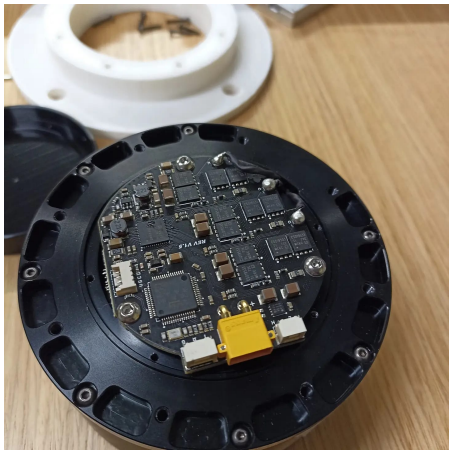

Figure 5.3. AK8080 rear part.

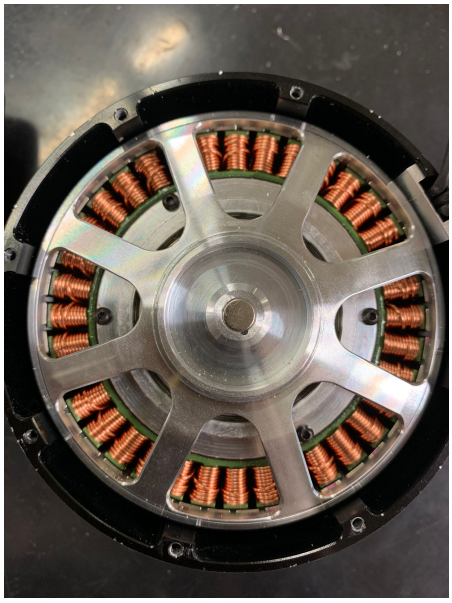

Figure 5.4. AK8080 BLDC Motor.

Fig. 5.1 to Fig. 5.4 are some closeup of the actuators. Fig. 5.1 shows the front part of the actuator, where the output shaft of the actuator can be screwed to the link joint. In the front part also resides the planetary gearbox with 80:1 reduction, illustrated in Fig. 5.2. Indeed on the back is installed the motor, an outrunner BLDC with 21 pole-pairs, a peak current of 40A at 48V for a peak torque of 144Nm at the output of the gearbox. This is shown in Fig. 5.4. A

magnet is installed on the rear cover of the motor. The rotating magnetic field produced by this magnet is read by an absolute magnetic encoder installed at the rear of the board shown in Fig. 5.3. This printed circuit board contains an STM32F446 microcontroller, which execute the control algorithm and handle the communications, a DRV8323, which is a 3-phase smart gate driver with current shunt amplifiers, and the necessary power circuitry, other than the magnetic absolute encoder discussed previously. The fact that this actuator integrates this components make it a "smart actuator", i.e. an actuator that require only some high-level commands to be operated. The fundamental part of this system is the integrated control algorithm illustrated in Fig. 5.3

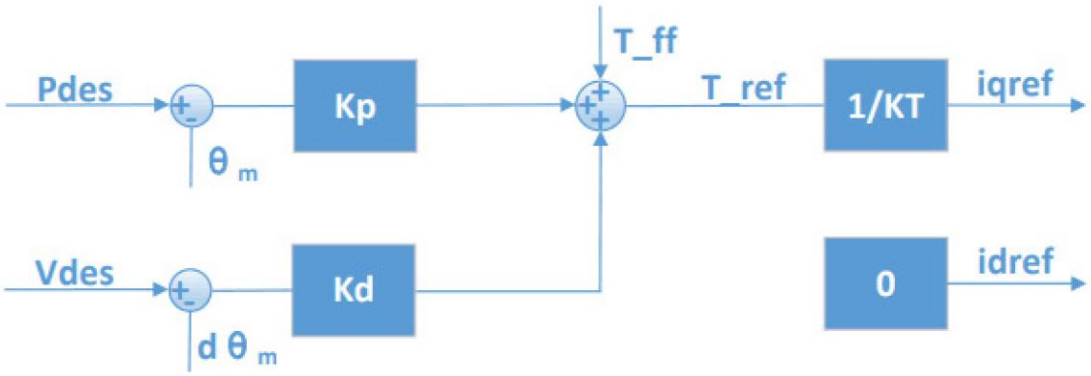

Figure 5.5. AK8080 integrated control scheme.

The simplicity of this control scheme is evident. The controllers consist solely of proportional gain blocks. Controlling these actuators using the software provided was not at all straightforward as many bugs plague this version of the product. The manufacturers did, however, support us in debugging operations by providing a firmware version to resolve serious problems presented by two motors, regarding the feedback current reading. The problem is depicted in Fig 5.4. Once solved the problem, I opted to minimise the use of the internal controller by providing the actuator with a torque command, which corresponds to the innermost loop of the control scheme.

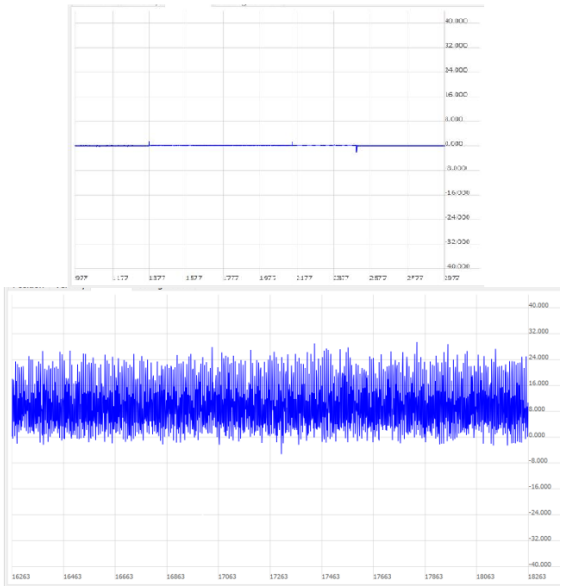

Figure 5.6. Problem encountered in reading the feedback current.

Talking about the AK8080 actuator is worth mentioning the communication protocol which it uses.Talking about the AK8080 actuator is worth mentioning the communication protocol which it uses. As the datasheet states, the AK8080 can communicate using the CAN protocol at a speed of 1 Mbps. The datasheet also report the message format that the motor expect to receive in order to function properly and the feedback message structure that the motor send back every time it receive a CAN frame. Both formats are illustrated in Fig. 5.5 [12].

Command message format

Data Field	DATA[0]	DATA[1]	DATA[2]	DATA[3]	
Data bits	7-0	7-0	7-0	7-4	3-0
Data content	Motor position 8-H	Motor position 8-L	Motor speed 8-H	Motor speed 4-L	KP 4-H

Data Field	DATA[4]	DATA[5]	DATA[6]		DATA[7]
Data bits	7-0	7-0	7-4	3-0	0-7
Data content	KP 8-L	KD 8-H	KD 4-L	Current 4-H	Current 8-L

Feedback message format

Data Field	DATA[0]	DATA[1]	DATA[2]	DATA[3]	DATA[4]
Data bits	7-0	7-0	7-0	7-0	7-4
Data content	Drive ID No.	Motor position 8-H	Motor speed 8-L	Motor speed 8-H	Motor speed 4-L

Data Field	DATA[4]	DATA[5]
Data bits	3-0	7-0
Data content	Current 4-H	Current 8-L

Figure 5.7. AK8080 CAN Message format

5.2 Sensors

In the proposed approach to the problem of controlling an inflatable manipulator, sensors play a key role: thanks to the feedback produced, the controllers are able to generate control signals so that the manipulator performs the required task with a certain margin of error. All the sensors that will be implemented in the Popup robot structure are proprioceptive, i.e. used to measure the internal state of the manipulator. Typically there are 3 quantities that characterize the internal state of the joint of a manipulator: position,speed and torque.

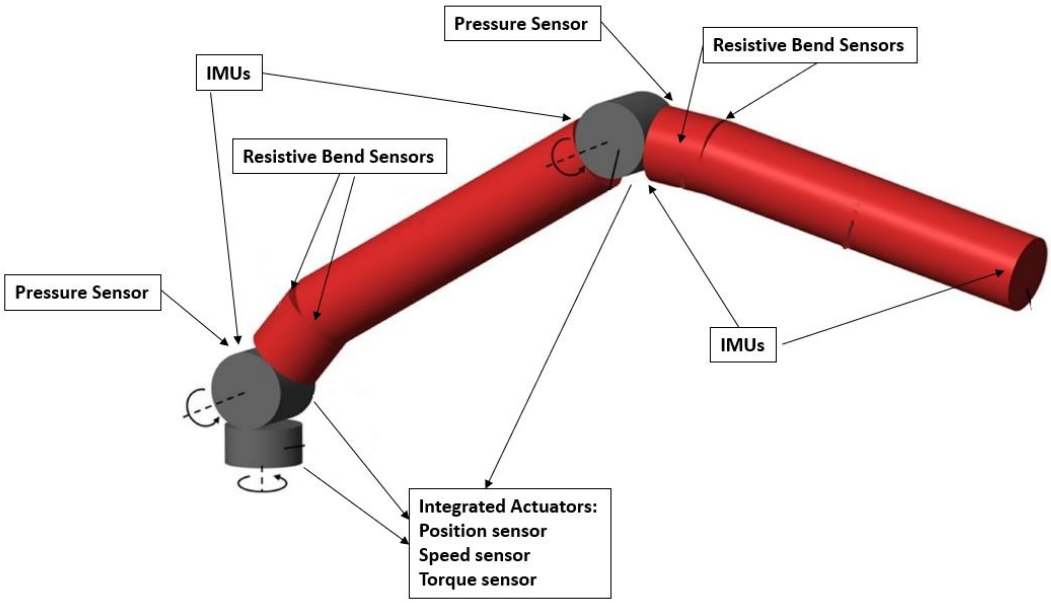

Figure 5.8. General scheme of sensors system.

Position transducers provide an analog or digital signal proportional to the linear or angular displacement of the moving part on which they are installed. The most commonly used position transducers are encoders, which can be absolute or incremental.

In this application the absolute encoder is integrated in the actuator seen before. it is a magnetic absolute encoder with 12bit precision which transmit its reading to the MCU integrated in the actuator.The MCU then provide to compute the angular displacement and the angular speed,sending everything as 4 data byte as part of the CAN Frame discussed in the actuator section.
Also the joint torque is computed by the internal controller of the actuator.No additional information are available about the way the MCU computes the torque, but it likely start by measuring the currents on each phase and then considering a sort of torque constant, typical of bldc motors. The sensors presented so far are typically used in any robotic application where control of some kind is required. The problem addressed by this thesis work requires knowledge of other manipulator parameters in order to counteract the effects of link deformation. These parameters are mainly the degrees of deformation of the flexible link, which are measured in two different ways and then fused with a sensor fusion algorithm in order to obtain an estimate as accurate as possible.

5.2.1 IMU

An Inertial Measurement Unit is an electronic device that measures the orientation of a rigid body using a combination of accellerometer, gyroscope and magnetometer. The accellerometer detect linear accelleration,such as the gravity accelleration, the rotational rate is measured by the gyroscope while the magnetometer detect the earth magnetic field, used as heading reference. IMUs are nowday probabily the most used orientation sensor, in fact almost every smartphone and tablet contain an IMU. For what concerned this thesis, the IMU's components are built using MEMS technology. This approach ensure low production cost, small dimension and an unattainable orthogonality between sensors, which leads to better performance.

Accellerometer

Accellerometer are the most popular inertial sensors, able to provide the measure of accelleration in a single or multi-dimensional axes. MEMS Accellerometer functioning is based on the displacement that an acceleration causes on a mass. This mass is etched into a silicon surface, and it is suspended by a beam, whose elasticity cause it to behave as a spring.The acceleration is computed solving the second-order equation which relates the suspended mass to the occurred acceleration.

Gyroscope

MEMS gyroscope measure angular rate by means of Coriolis accelleration. Tuning-fork gyroscope are the most popular structure: it contains a pair of masses the are driven to oscillate in opposite direction. When the sensor is rotated, the coriolis force causes a differential force, orthogonal to the main oscillation. This force cause a displacement of the structure that can be measured to estimate the rotation rate.

Magnetometer

Magnetometers are devices used to measure the magnetic field. Are widely used for measuring the Earth's magnetic field, to detect magnetic anomalies or to detect the dipole moment of magnetic material. It is also commonly as heading reference when implemented in attitide and heading reference systems for aircraft. The most common type are solid-state hall effect sensors. These sensor produce an output voltage proportional to the applied magnetic field, and are also cabaple of providing information about the direction on the field, making it perfect to detect orietation. However, the Earth's magnetic field is very weak if compared to the numerous source of magnetic field noise, sugn as magnets or electric motor, but

also ferromagnetic metals. For this reason, the magnetometer need an accurate calibration procedure with the aim of compensate the Hard Iron and Soft Iron offset. More details are given in Chapter

Invesense MPU9250

MPU-9250 is a multi-chip module (MCM) consisting of two dies integrated into a single QFN package. One die houses the 3-Axis gyroscope and the 3-Axis accelerometer. The other die houses the AK8963 3-Axis magnetometer from Asahi Kasei Microdevices Corporation. Hence, the MPU-9250 is a 9-axis MotionTracking device that combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer and a Digital Motion Processor™ (DMP) all in a small 3x3x1mm package available as a pin-compatible upgrade from the MPU6515. With its dedicated I2C sensor bus, the MPU-9250 directly provides complete 9-axis MotionFusion™ output. The MPU-9250 MotionTracking device, with its 9-axis integration, on-chip MotionFusion™, and runtime calibration firmware, enables manufacturers to eliminate the costly and complex selection, qualification, and system level integration of discrete devices, guaranteeing optimal motion performance for consumers. MPU-9250 is also designed to interface with multiple non-inertial digital sensors, such as pressure sensors, on its auxiliary I 2C port. MPU-9250 features three 16-bit analog-to-digital converters (ADCs) for digitizing the gyroscope outputs, three 16-bit ADCs for digitizing the accelerometer outputs, and three 16-bit ADCs for digitizing the magnetometer outputs. For precision tracking of both fast and slow motions, the parts feature a user-programmable gyroscope full-scale range of ±250, ±500, ±1000, and ±2000°/sec (dps), a user-programmable accelerometer full-scale range of ±2g, ±4g, ±8g, and ±16g, and a magnetometer full-scale range of ±4800µT. Communication with all registers of the device is performed using either I2C at 400kHz or SPI at 1MHz. For applications requiring faster communications, the sensor and interrupt registers may be read using SPI at 20MHz [7].

In the context of this thesis work, IMUs have been choose as one of the sensors to assess the deformation of inflatable links. Fig. 5.7 represent the ipotetical position of the IMUs installed on a link. The IMUs installed at the base of each link are the Link Reference IMU. Applying a particular orientation filter to the output of IMU sensors result in a complete measurement of the orientation relative to the direction of gravity and the earth's magnetic field. The orientation estimated by the IMU installed on the link bases should coincide with the orientation estimated by the IMU installed at the end of the links assuming the links do not deform. However, when link deformation occurs the orientation estimates will no longer be consistent. By subtracting the two orientation estimates it is possible to derive an estimate of the difference in orientation of the two ends of the link. At this

point, having obtained the angular difference, trigonometry could be applied to determine the actual bend angle of the link. Taking into account that the calculation has to be carried out online by a device with limited computational power, it was decided to eliminate the additional burden of the trigonometric calculation, observing that the maximum error due to this approximation is approximately 0.5° on an effective deformation of 10°. Considering that this value coincides with the accuracy declared by the developer of the orientation estimation algorithm, this approximation should not reduce the overall accuracy of the system

Figure 5.9. Simscape Multibody simulation on flexible link.

Madgwick's Orientation Filter

After presenting the approach of deformation estimation through the use of two IMUs it is necessary to quickly summarize the orientation estimation algorithm that was used in the practical implementation on the microcontroller. developed an orientation filter applicable to the sensor array consisting of tri-axis gyroscope, tri-axis accellerometer and tri-axis magnetometer with built-in magnetic distortion and gyroscope bias compensation. "The filter uses a quaternion representation, allowing the accellerometer and magnetometer data to be used in a derived and optimised descending gradient algorithm to calculate the direction of the gyroscope measurement error as a quaternion derivative." The results shows that this filter exceeds the accuracy levels of algorithms based on Kalman estimator. Furthermore, the Madgwick filter requires less computational power because it does not require the linear regression iterations that are fundamental to the kalman process [9].

5.2.2 Resistive Bend sensor

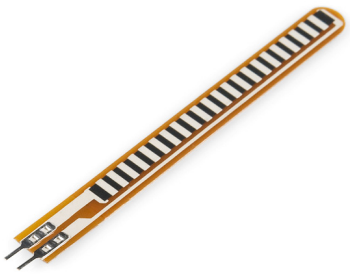

Figure 5.10. Flex Sensor

Resistive Bend Sensor (RBS fron now on) are used to convert a change in bend into an electrical resistance variation. Among the multiple solution available for angle measurement, such as IMUs or Optical Coordinate system, RBS offers a low-cost, low-complexity approach. RBS are made of electrical conductive pattern, tipically in an ink form, drawn on a flexible substrate. The operating principle is very simple: The conductive ink presents micro crack when placed in tension. The elongation causes the crack to become wider, thus causing the resistance to increase.When the material is returned to its position, the distance between the crack faces shrink and the resistance decreases. Given such behavior, the conductive material has to be placed upon the convex side of the bent surface, otherwise flexion would act in compression rather than in stretching [11].

Four RBS are intended to be installed at the base of each link because, as said, RBS are typically single axis, so to be able to measure bending in each directions requires four sensors. The choice of an RBS fallen on the ones sell by Spectra Symbols. They provide two different version which differ for the length. The selected version is the one 115mm long, which datasheet report a resistance of 10K Ohms when flat and a tolerance of +-30%

In Chapter 8 the implementation of these sensors is further discussed, focusing on the selected conditioning circuit.

5.2.3 Pressure Sensor

The inflation pressure of the structures that make up the manipulator plays a key role in its behaviour. In [cit], an inflatable link has been statically characterised. The results showed that there is no direct correlation between the inflation pressure and the stiffness of the virtual spring used to model the link (elastostatic approach discussed previously). However, it is observed that as the link pressure increases, the maximum applicable moment before wrinkling occurs increases. It was therefore decided to measure the link pressure in real time so that the main controller, taking into account the static characteristic of the link can reduce any control torques to a level that does not cause wrinkling for a given applied payload and end-effector trajectory

Figure 5.11. Gage Pressure Sensor

A pressure sensor is a device capable of measuring the pressure of gases or fluids and returning an electrical signal as a function of the applied pressure. There are several types of pressure sensors, including - absolute pressure sensor, which measures pressure relative to a perfect vacuum - gauge pressure sensor, which measures pressure relative to atmospheric pressure

These types of sensors are further classified according to the technology they use: some exploit piezoresistive effects to detect the deformation caused by pressure on a membrane, others use a diagram and a cavity to form a variable capacitance, and many other approaches that I will not discuss for the sake of brevity.

The choice fell on the HSCSAND001BGAA5, a piezoresistive silicon pressure sensor with ratiometric analog output. Moreover, the HSC series is fully calibrated and temperature compensated so that no calibration or characterization is required [6]

5.3 Communication Protocols

To complete this section of the thesis, where the component forming the system have been presented it is worth presenting the communication protocol that have been selected to allow each component and subsystem to share information

5.3.1 I2C

I^2C,I2C or IIC, stands for Inter-Integrated Circuit is popular bus invented in 1981 and used for multiple masters and multiple slaves communication. Among the others, one of the largest benefits is that I^2C requires only 2 wires. This thanks to the use of an open-drain with an input buffer on the same line. The open-drain output can pull the bus down or release the bus so that, for example, another master can use the bus. The physical I^2C interface consist of the serial clock (SCL) and serial data (SDA) lines. The lines are mantained at high level by 2 pull-up resistors that have to be sized considering the capacitance present on the lines. A slave device cannot transmit data unless it is addressed by the master, for this reason each device using I^2C has a specific address. If the master want to send data, first a START condition must be sent. A START condition consist of an high-to-low transition on the SDA while the SCL is mantained high. Then the master send the address of the slave device, which is composed by 7 bits. if the address is correct the slave responds with an ACK bit. At this point the master is free to send the data, always one data bit during each pulse on the SCL. Reading from a slave is very similar, the difference reside in the instruction that the master has to send to the slave about which register it wishes to read. This is done by the master starting off the transmission in a similar fashion as the write, by sending the address with the R/W bit equal to 0 (signifying a write), followed by the register address it wishes to read from. Once the slave acknowledges this register address, the master will send a START condition again, followed by the slave address with the R/W bit set to 1 (signifying a read). This time, the slave will acknowledge the read request, and the master releases the SDA bus, but will continue supplying the clock to the slave. During this part of the transaction, the master will become the master-receiver, and the slave will become the slave-transmitter [8].

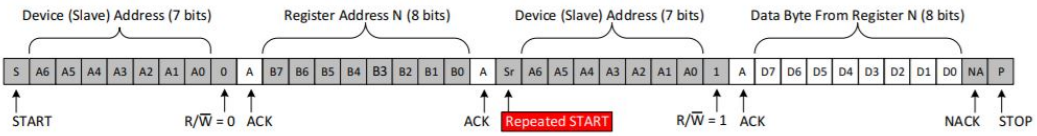

Figure 5.12. I^2C master read 1 byte from slave register

5.3.2 CAN

CAN is a serial communication bus developed by Bosch for the automotive industry to replace complex wiring harness with two-wire bus. The CAN bus is a multi-master, message broadcast system with a maximum signaling rate of 1 Mbps in its standard format, typically used to broadcast short message among different nodes. The CAN communication protocol, ISO-11898, is a carrier-sense, multiple-access protocol with collision detection and arbitration on message priority. The priority of a message is encoded in the identifier field. Fig. 5.13 shows the frame format of standard CAN. Bit fields have a precise meaning:

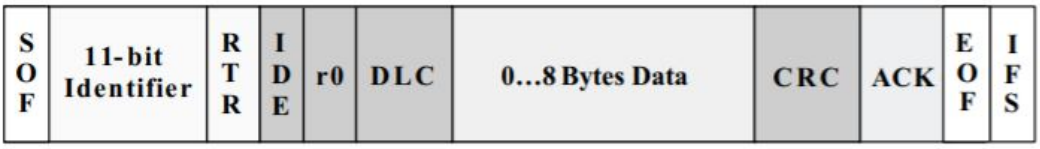

Figure 5.13. Standard CAN Frame

- SOF is the start of frame bit used to syncronize the nodes on a bus after a idle period
- ID are 11-bit that assign the priority of the message. Lower ID means higher priority.
- RTR means remote tramsission request and is a bit used when a node require data from another node.
- IDE is a bit that inform if the message use Standard 11-bit ID or Extended 29-bit ID
- r0 is a reserved bit
- DLC are 4 bit containing the number of bytes of data being trasmitted
- Data are up to 64 bits of data to be trasmitted
- CRC are 16-bit of cyclic redundancy check that perform error detection on the tramsitted data
- ACK are 2 bit. The ACK bit is dominant if a receiving node receives the message without errors. Looking at it the trasmitter is aware if the packet fails
- EOF are 7 bit that marks the end of a CAN message.

- IFS is an interframe space to allow the controller to move a received frame in a message buffer area.

CAN is famous for its robustness, making it the typical choice for communication bus in harsh and noisy environment. This propriety is due its abundant error-checking proecudures: three at message level and two at the bit level. Every time a message fails one of this error detection methods, an error frame is generated from the receiver and the trasmitter is forced to resend the message. At message level error checking is contained in the 16-bit CRC and in the ACK. Also some bit are checked because they sohuld be always recessive. Those bit are the SOF, EOF, ACK Delimiter and the CRC delimiter. The robustness is also derived from the nature of the CAn bus signal, which are differential. Balanced differential signaling reduced noise coupling and allows for high signaling rates over twisted-pairs. The High-Speed ISO 11898 standard specifies the characteristic of the physical CAN Bus. The maximum signaling rate of 1 Mbps can be reached with a maximum bus lenght of 40m and a maximum of 30 nodes. Fig. 5.14 shows the network topology suggested by the ISO 11898 STandard. It define as bus a single line of twisted-pair cable terminated at both ends with a 120 ω resistor in order to match the characteristic impedance and to avoid signal reflection [4].

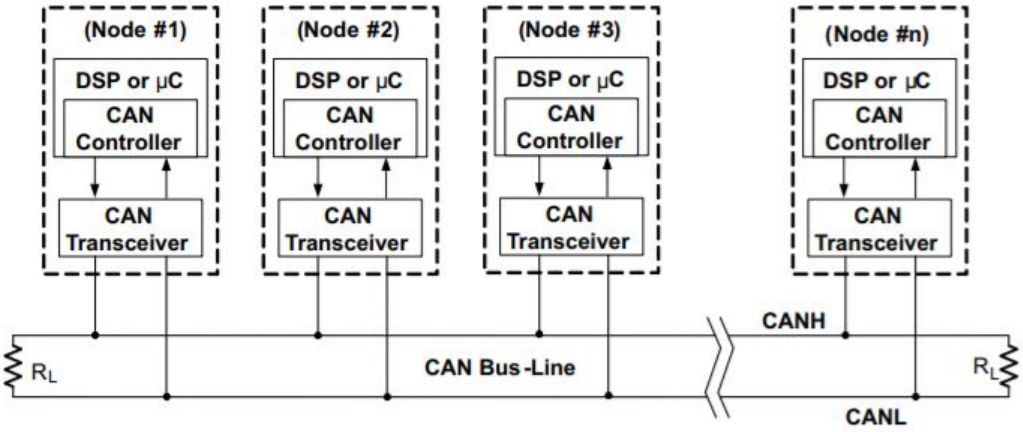

Figure 5.14. Standard CAN Frame

Chapter 6

Main Controller Board

The main controller board (MCB from now on)is the electronic subsystem located at the base of the manipulator: it acquires the data sent from the Link Data Boards and motors via CAN, executes the control algorithms ,drives the motors and send back telemetry through a serial connection. The heart of this subsystem is the STM32H743ZI2 microcontroller used thanks to the NUCLEOH-H743 development board. The stm32h743 contains the ARM Cortex-M7 core, capable of operating at a maximum frequency of 480 MHz and scored 1027 DMIPS, making it one of the most powerful microprocessors on the market.

With 35 communication interfaces (including CAN and Ethernet), analogue peripherals (ADC and DAC) and multiple timers, the STM32H743ZI2 is a development platform that contains the necessary peripherals and expresses sufficient computing power to allow the execution of complex functions such as those required by the problem in question.

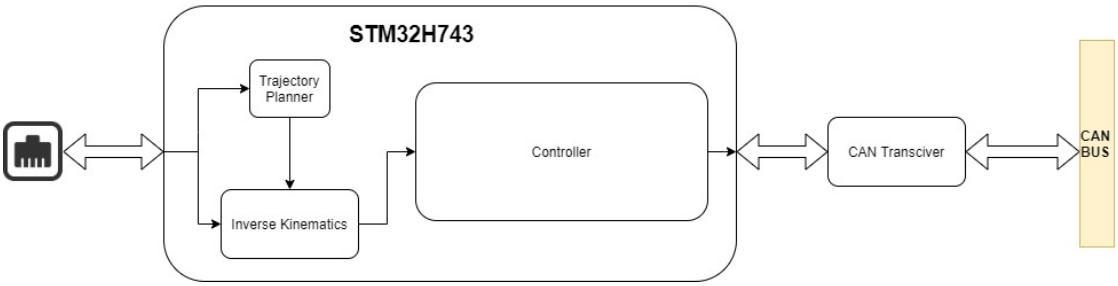

Figure 6.1. Block Scheme of MCB.

Fig 6.1 reports the block scheme of the MCB. As said, the core element is the STM32H743ZI2 which execute the control algorithm discussed in Chapter 3. After an initialization procedure, The MCB receives the wanted pose through a serial

connection with the host PC. The inverse kinematic function is executed to obtain the target joint position. Knowing the actual joint position, the trajectory planner function is executed. Then the output is passed to the Position controller function, which is executed at 100Hz. Subsequently, the compensated error produced by the position controller is passed as reference speed to the speed controller, that is executed at 1KHz frequency. The produced command is sent via CAN to the motors, which thanks to their internal controller,execute the motion. The main controller also receives from the link data board all the information regarding the deformation of the links and tries to minimize oscillation and deformation with the strategy discussed in Chapter 3.

6.1 MCB Hardware

The hardware development of the MCB was simpler due to the few components present. This is why we opted to build the circuit in the laboratory with a millefori board and copper-plated cable for the connections. The figure below shows the wiring diagram of the board installed above the NUCLEO-STM32H7.

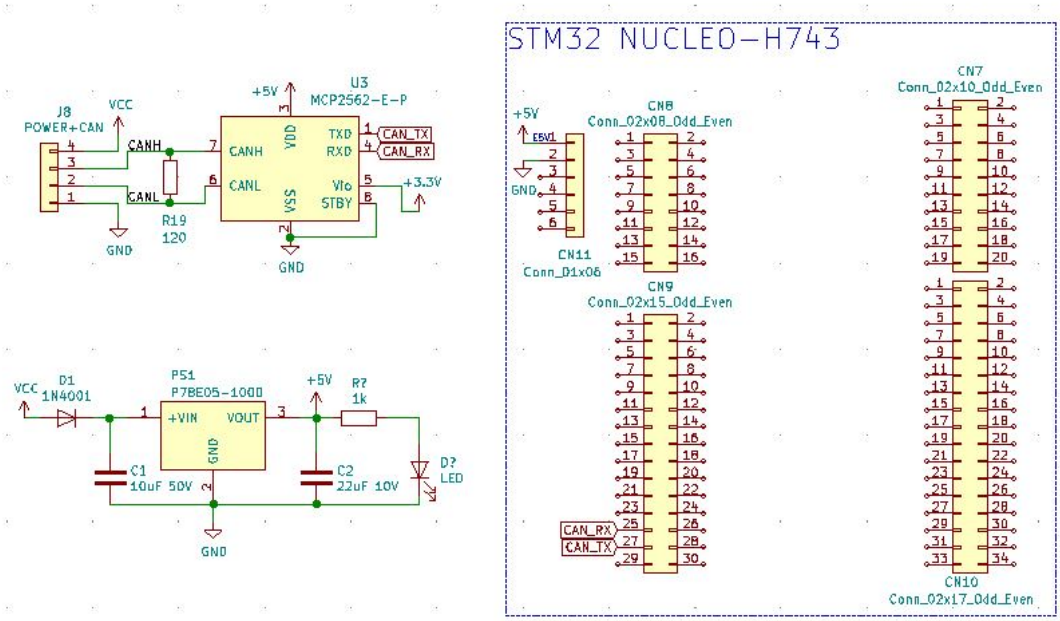

Figure 6.2. MCB Circuit Schematic.

Generally three components form the MCB hardware other than the already

discussed MCU: the CAN Transceiver, the DC-DC converter and the connectors. The input connector J8, a 4-pin, 0.1" pitch screw terminal, carries both power and communication.The 24 Vdc input voltage is lowered to 5 Vdc by means of a DCDC converter, PS1, produced by CUI. This converter has a maximum current of 1A, sufficient to safely power the NUCLEO which, as the datasheet states, can draw up to 500mA. Some external capacitors, C1 and C2, are required to ensure stability. Also a LED has been installed on the 5Vdc output to verify the presence of power. The CAN Transceiver input, CAN LOW and CAN HIGH are directly connected to J8 and so to the CAN bus. A 120 ohm resistor is placed in parallel to the CAN Bus lines as requested to ensure a proper bus functioning. Not all node will have a 120 ohm resistor, only the first and the last, as explained before in the section about the CAN communication. The output of the Transceiver is connected to the devoted pin on the NUCLEO-STM32H743 by means of female header pin with 0.1" pitch. Fig. 6.3 shows the developed circuit on top of the NUCLEO-STM32H743 board.

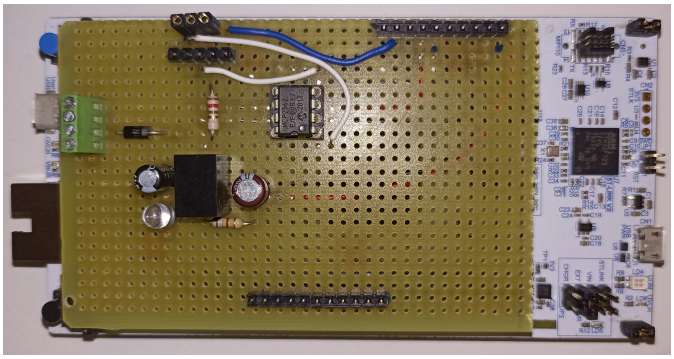

Figure 6.3. MCB Circuit Schematic.

6.2 MCB Firmware

The firmware for the MCB has been developed in C using the Integrated Developement Editor from STmicroelectronics, STM32CubeIDE. Before starting programming the needed function, the STM32H743 needs to be configured. To do that, STmicroelectronic developed a very useful and intuitive software called STM32CubeMX. It is a MCU configurator with a suggestive graphical interface. Fig. 6.4 depict a screenshot taken during FW development.

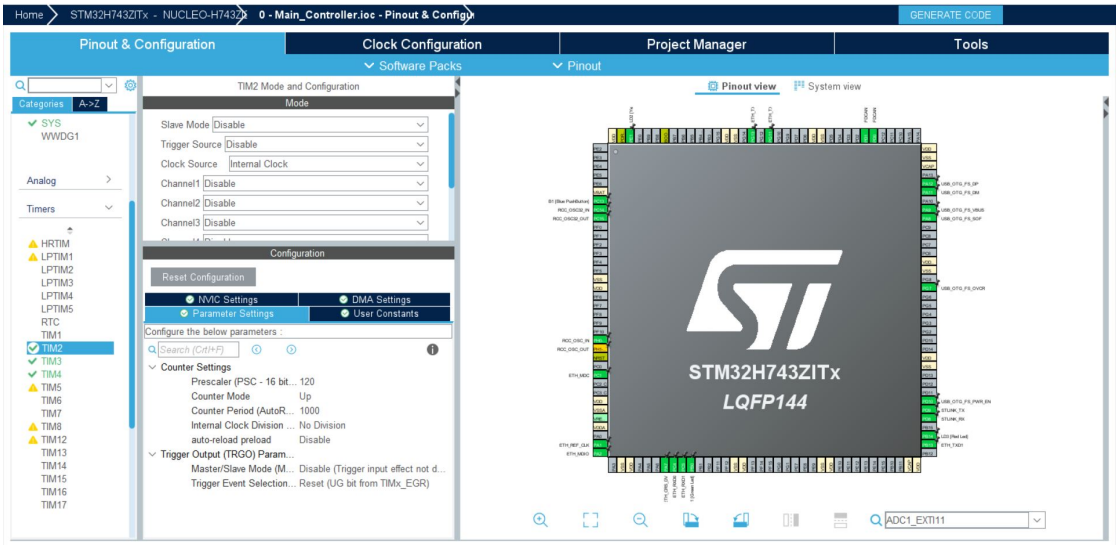

Figure 6.4. STM32CubeMX configurator.

The first step using CubeMX was to select the MCU for which you want to generate the project, in this case the STM32H743ZI2. At this point the software asks if you want to initialise certain peripherals independently. This is because being installed on the NUCLEO board, some pins of the MCU are already connected for example to the leds or to the serial port. The next step is clock configuration. In general, when choosing the clock frequency, the power consumption is taken into account, which in certain applications should be limited.Since I have no limits on power consumption, I decided to set a clock frequency of 240 MHz for the CPU.

After deciding the system clock speed, CubeMX automatically calculates the clock frequencies for various peripherals, such as timers. In particular, the value associated with the APB1 Timer clock and APB1 Peripheral clock will be needed later. This frequency has been set to 120MHz, the maximum for timers and peripherals. Knowing the frequency of the APB1 bus from the clock configurator, it was possible to configure the necessary peripherals: the FDCAN and the timers. Timers are probably the most common peripheral in any MCU-based design. In

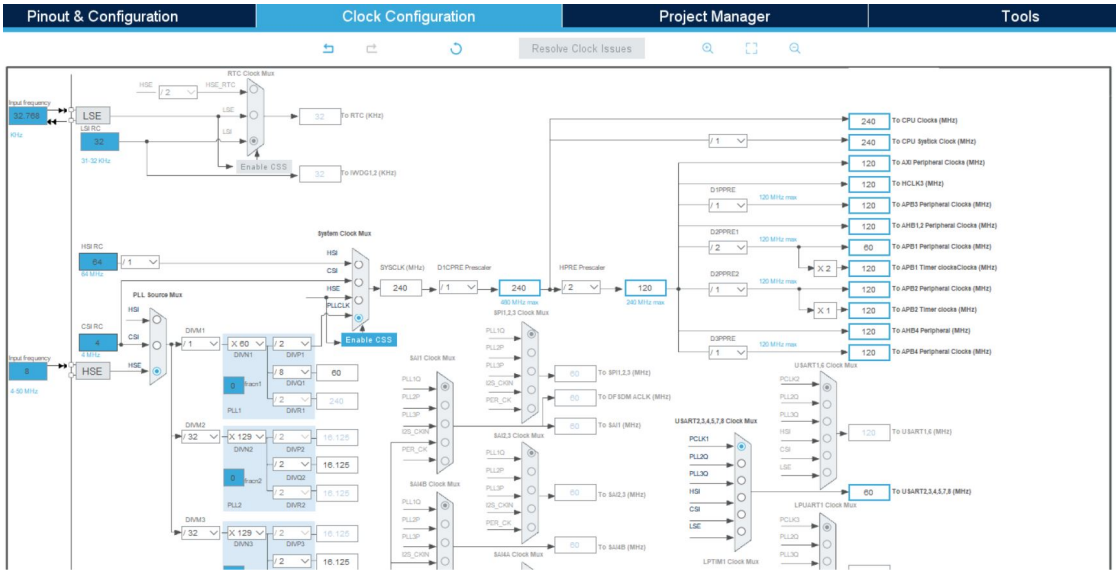

Figure 6.5. STM32CubeMX Clock configuration

general, they are counters to which an overflow value is set that, when reached, triggers an action, usually a system interrupt. By exploiting the properties of system interrupts and configuring timers correctly, it is possible to use them to execute certain portions of code at specific times. This is a necessity when, as in the case of this thesis, we work with acquisition and control systems that require a precise sampling time.

This project required a total of 3 timers to be configured and used. I chose to use TIM2,TIM3 and TIM4 among the 15 available timers peripheral present in the STM32H743. TIM2 is used to execute the speed controller. Every time the timer counter reaches the set value, it generates an interrupt that executes a function that sets a flag variable to one. Meanwhile in the loop an if statement is used to check the status of the flag variable. when it is set to one the position controller is executed. subsequently the flag variable is reset to zero, waiting for the next interrupt.

To execute the speed controller at a frequency of 1000 Hz two parameters must be defined, the prescaler and the counter period. The prescaler is a divider that reduces the frequency of the APB1 the timer bus. Choosing a value of 120 results in the timer counter being updated every microsecond. At this point, remembering that a 1000 Hz frequency corresponds to a 1 ms period, it is necessary to set the counter period to 1000. The computation is the following

$$T_{out} = \frac{PSCxCounterPeriod}{F_{CLK}} \tag{6.1}$$

$$0.001sec = \frac{120xCounterPeriod}{120MHz} \tag{6.2}$$

$$CounterPeriod = 1000 \tag{6.3}$$

The figure below shows the configuration set on CubeMX. In the NVIC setting tab there is a box which, if checked, enables the global interrupt associated with the timer. NVIC configuration is discussed later.

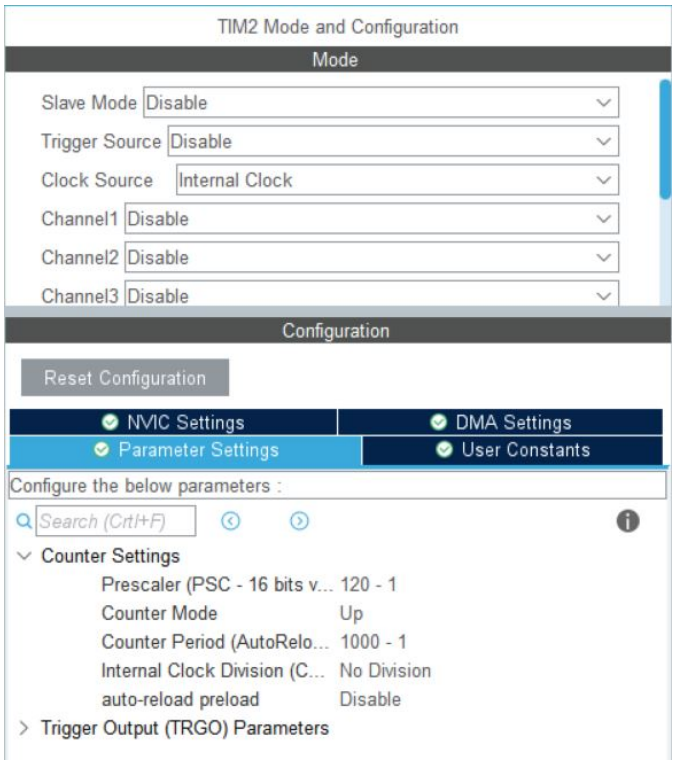

Figure 6.6. STM32CubeMX Timer configuration.

The other two timers, TIM3 and TIM4, have been configured to trigger an interrupt respectively at 100 and 10 Hz. The configuration steps are the same, so i will skip it for the sake of brevity.

The configuration of the FDCAN peripheral requires similar considerations: First of all, the bit rate of the can bus was chosen to be 1 Mbps. This due to the default communication speed configured in the AK8080 actuators. As for timers, some parameter must be inserted in CubeMX prior to generate the project. The first parameter to be selected is the frame format. The CAN peripheral of the STM32H743 is in fact able to manage both Classic CAN and the more recent

FDCAN, with variable data rate. For this application we have chosen to use Classic CAN. The fundamental parameters for configuring the FDCAN peripheral are those relating to timing. In fact, it is necessary to carry out simple calculations starting from the peripheral's clock frequency.

The Nominal Bit Rate of the network is given by

$$f_{NBT} = \frac{1}{t_{NBT}} \tag{6.4}$$

where t_{NBT} is the Nominal Bit Time. This is divided in four non-overlapping time segments, illustrated in Fig. 6.7

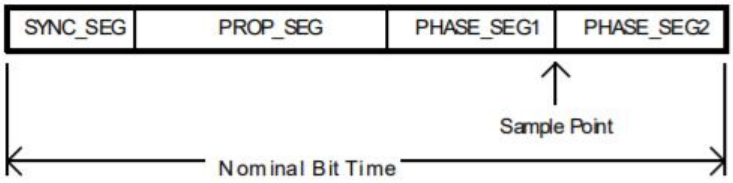

Figure 6.7. CAN Bit Segments

Each of these segments is an integer multiple of a unit of time called Time Quantum, which duration is equal to the CAN system clock, derived from the system clock passing it in a prescaler.

In this case the APB1 Peripheral Clock is 60 MHz. For a bit rate of 1 Mbps, setting a prescaler value of 4 gives a CAN system clock of 15MHz and a Time Quantum of 66ns. This gives 1000 / 66 = 15 time quanta per bit. Considering 1 time quanta for the $SYNC_SEG$, 14 time quanta left for the propagation, phase 1 and phase 2 segments. The propagation segment can be computed as

$$t_{PROP\_SEG} = 2(BUS\_PROP\_DELAY + TRANSCEIVER\_PROP\_DELAY) \tag{6.5}$$

where

$$BUS\_PROP\_DELAY = 5x10^{-9}sm^{-1} \tag{6.6}$$

$$TRANSCEIVER\_PROP\_DELAY = 235ns \tag{6.7}$$

and therefore, for a bus length of 2m

$$t_{PROP\_SEG} = 2(235ns + 10ns) = 490ns \tag{6.8}$$

that correspond to 8 time quanta.The remaining 6 time quanta that can be divided among phase segment 1 and 2. The resulting parameters are listed in Table

Bit Rate	Pre-scaler	N. of time quanta	PropSeg + Phase1Seg	Phase2Seg
$1Mbps$	4	15	12	2

Table 6.1. CAN Timing parameters

This terminated the configuration procedure. At this point, after having decided the wanted IDE to be used, CubeMX generates a C Project with all the configuration previously discussed. From now on I will discuss the algorithm that have been written in C. Fig.6.10 illustrate the flow of the firmware executed by the MCU. The complete listing is presented in Appendix C.

The first part of the main contains the inclusions of the libraries, the definitions of certain constants and variables used by the algorithms, and the prototypes of the functions used. Once in the main, functions generated by CubeMX are executed to initialise the previously configured peripherals. At the end of this phase, the POPUP_system_check() function is executed.

This function is designed to check the status of the manipulator before starting to use it. As a first operation, once this function has been executed, the MCB sends a Deactivate CAN messages to the motors to check their presence. In fact the actuators reply to the message with another message containing their id in the first frame. When a message with ID equal to 0 is received from the bus, the MCB executes the unpack_motor_FB() function, which, if executed for the first time, sets a flag variable used to indicate the presence or absence of the motor. Fig 6.8 shows the exchange of CAN messages between the MCB and motor 2.

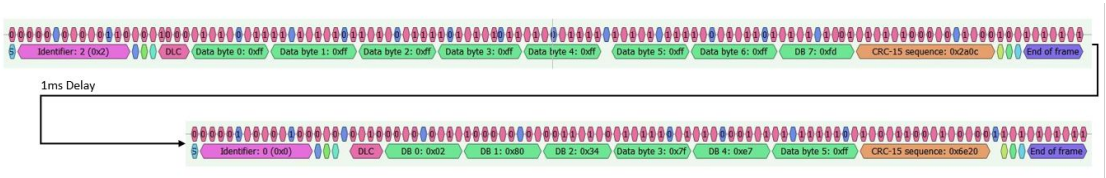

Figure 6.8. CAN Message between MCB and Motor 2.

If all the motors are found, the MCB start checking the status of the Link. This is done thanks to a status package that the Link Data Board send over the CAN bus after the first initialization. The ID associated to this message is 20. When

the MCB receive a message with this ID, it executed the unpack_link_STATUS() function. Similarly to the motor, the ID of the LDB sending the status message is in the first data frame. Then frame 1 is the IMUs status, frame 2 the ADC status while frame 3 and 4 contains the higher and lower 8 bit of a 16bit integer carrying the link pressure reading. The status packet from LDB is further discussed in the next section and illustrated in Fig 6.9. From the status data the MCB compute the value of a variable, that if it is 1 means that all the sensor of the subject link works correctly and that the Link Data Board is correctly initialized. Again, this procedure is repeated for both Link 1 and Link 2. Once the Link status check in terminated, the MCB start executing the POPUP_calibrate_link_sensors() function. This function simply send a CAN message with ID 50. The LDBs that receive this message will start the calibration of the IMUs and the compensation of the RBS offset. Once terminated, they send a calibration completed message with ID 40. When the MCB receive a message with this ID, it executed the function unpack_link_cal_check(). In the first frame there is the ID of the LDB that sent the message. This procedure terminates when both LDB send their calibration completed message. At this point the system is considered as Initialized.

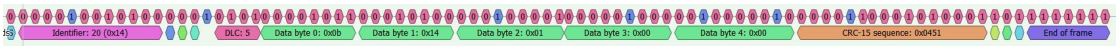

Figure 6.9. CAN Status Message between MCB and LDB.

Once the system is correctly initialized, the MCB send the activation command to all the actuators and starts the homing procedure. This aim to bring the manipulator from the zero-angle pose to a define pose from which the manipulator stays ready to execute further movements. In the POPUP_homing() function the Home Cartesian coordinate are inserted. From this, with the InverseKinematic function , the target joint position are retrieved. At this point the TrajectorPlanner function is executed, creating a matrix of point referred to precise time instant as seen in the theoretical discussion in Chapter 3. The first value computed by the TP is copied into the Position controller input variables and the system is ready to move. To start the movement two more function must be executed: the POPUP_start_controllers() function simply start the timer base of the timers configured previously. From there, the position controller function starts to be executed at 100Hz and the speed controller at 1 kHz. Because the first value computed by the TP as been copied to the input variable of the position controller, the manipulator now maintain its position. To initiate the planned movement, the function POPUP_start_plan() is executed in order to start the timer base of the timer used for the planning. Every time the timer4 goes in overflow it execute the function plan_step(), which simply copy the planned joint variable

into the position controller input variable at the planned time-step. When the plan_step counter reaches the maximum value (equal to the movement wanted duration divided the time-step) and the position error is lower than 0.005 rad the planned trajectory is considered as completed. The plan step timer is suspended and the UART receive interrupt is enabled to allow the input of a new Cartesian target. Once a new, valid Cartesian target is inserted, the procedure starts again: InverseKinematic computation , Trajectory Planning from the last position to the new wanted position and so on. While a planned movement is executed on the serial monitor a telemetry from the manipulator is printed. This contains data about joints position and speed, target position and speed,link pressure and Bending estimation from the LDB.

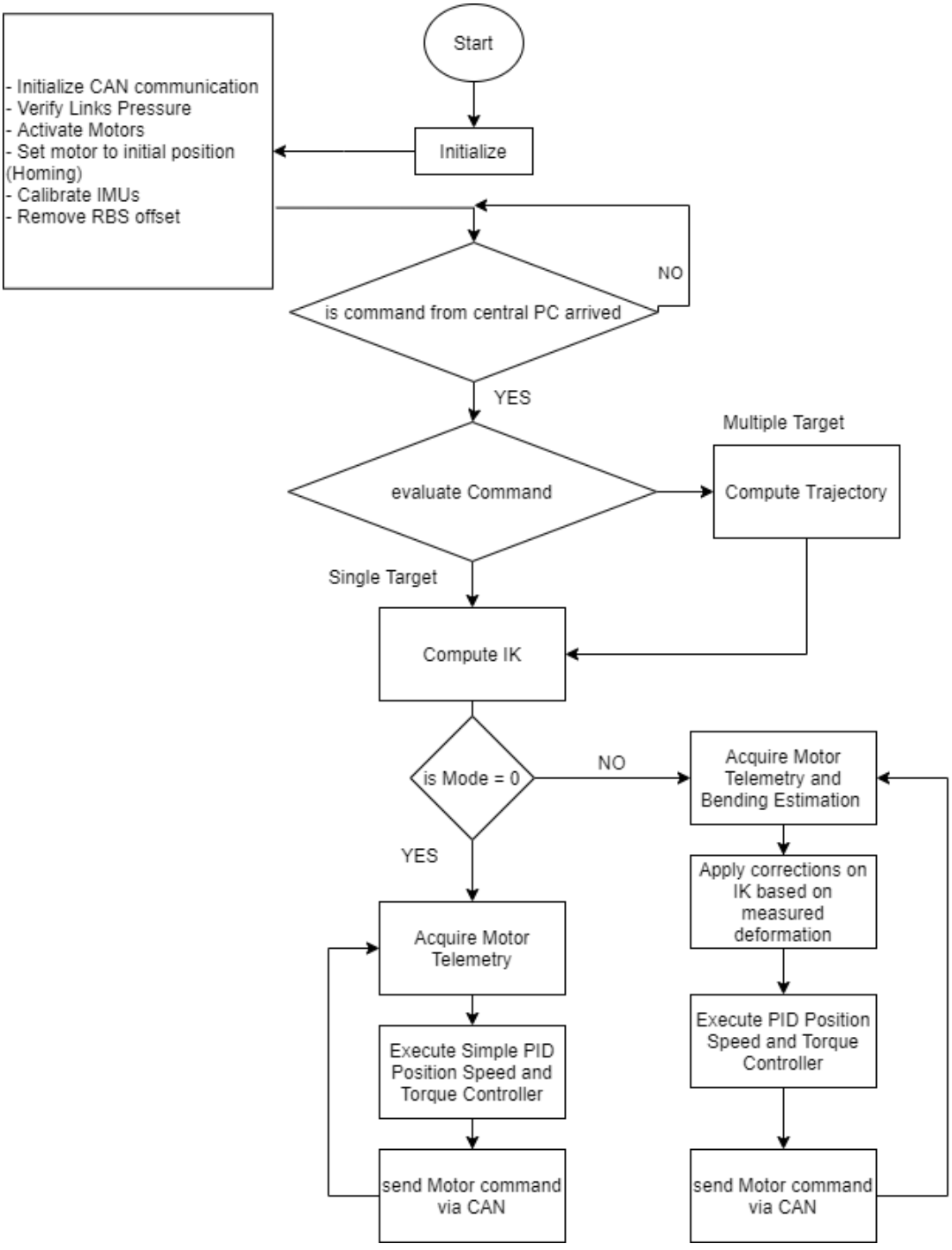

Figure 6.10. MCB Firmware Flow Diagram.

Chapter 7

Link Data Board

The Link Data Board (LDB from now on) is an electronic subsystem developed with two main objectives in mind: - reducing the computational effort required to the MCB - reducing the number of cables on the structure of the manipulator Fig. 7.1 shows a block scheme of the subsystem.

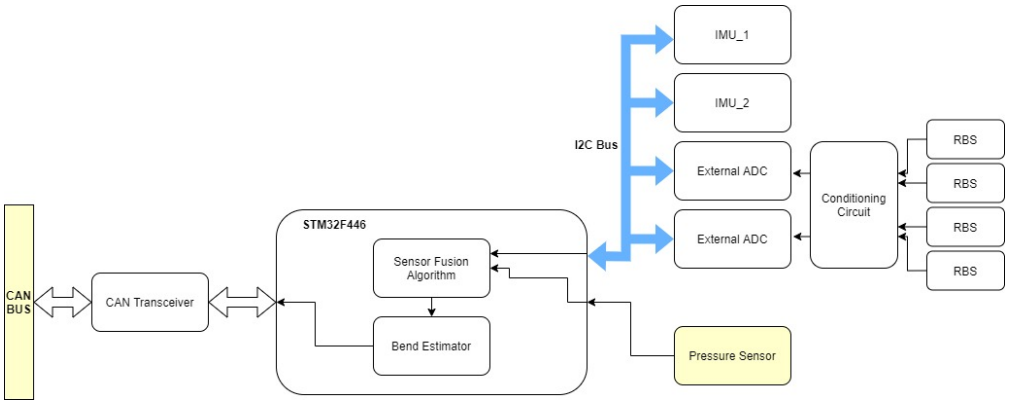

Figure 7.1. Block Scheme of JDB.

The main element of the subsystem is the STM32F446, an high-performance microcontroller from STmicroelectronics. It has been selected among other MCU for the great number of external peripheral and its good computational capabilities. As shown in Fig. 7.1, the LDB is connected to the CAN bus by means of a Transceiver. The CAN Bus is used to sent datas back to the MCB,mainly relative to the deformation state computed by the Bending Estimator function. Moreover, it is also used to receive some command, in particular during the initialization phase. As explained in Chapter 5.2, the selected IMUs uses the I2C Protocol to communicate with the microcontroller. This allow to further reduce the number

of interconnection: a single, 4-poles, cable is used as I2C bus, and each IMU is assigned with an unique ID. The RBS values are acquired through an external ADC, the ADS1115. It is an high precision 16bit ADC used mainly in data acquisition system. The differential reading capability is exploited in this approach since the RBS signal is conditioned with a Wheatstone Bridge. The differential signal, which is in the range of mV, is fed into the ADC and converted in digital value. The ADS1115 has an internal PGA that, for this application as been set to provide an FSR of +-256 mV, resulting is a resolution of 7.81 uV. Further details are given in the hardware section.

7.1 LDB Hardware

Given the higher number of interconnections required and the need for a more stable system to be installed close to the actuated joint, it was decided to implement the LDB on a printed circuit board. The free ECAD software KiCAD was used to create the schematics and design the printed circuit board due to the familiarity gained from previous experience. As for the MCB, the design process started from the definition of the specification. As said before, the goal of the LDB is to provide to the Main Controller Board information about the deformation of the link on which is installed.

Fig.7.2 shows the complete schematic of the LDB. Some components, as the CAN Transceiver or the DCDC are the same used for the MCB, therefore have been already discussed in the previous section. Fig. 7.3 and Fig. 7.4 show respectively the CAN circuit section and the DCDC section.

On the top right of the full schematic of Fig. 7.2 and in Fig. 7.5 are schematized the connectors used to connect to the IMUs and the pressure sensor. For the IMUs two 4-Pin screw terminal connectors are used,each of them carrying a separate I2C bus. The connector used for the pressure sensor is a 3-Pin screw terminal connector. Near it is visible the conditioning circuit, a voltage divider, needed to lower the maximum voltage output from the pressure of 5Vdc to 3.3 Vdc, the full scale range of the analog to digital converter integrated in the STM32F446.

The other connectors on the boards are the ones used to actually connect the LDB to the NUCLEO carrying the MCU and the screw terminal connectors used for the RBS. The connection between the LDB and the NUCLEO is made exploiting the expansion pin on the NUCLEO Board. By means of female pin head with 0.1” pitch the two board are coupled. The sensors, the transceiver and the external ADC are connected to the MCU via this pin header, as shown in Fig. 7.6

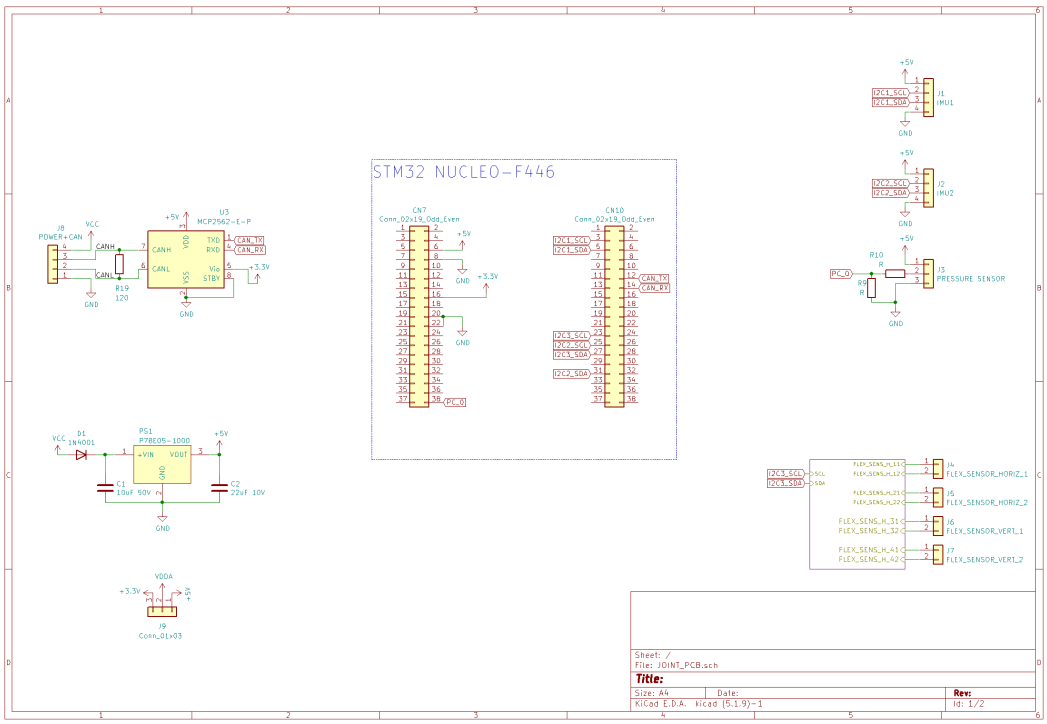

Figure 7.2. LDB Circuit Schematic.

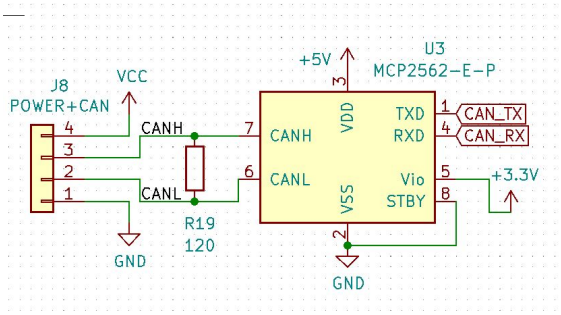

Figure 7.3. LDB Circuit Schematic CAN Section.

Last but not least, the RBS conditioning circuit section is schematized on the bottom right of Fig 7.2 and in Fig. 7.7 more in detail. This is composed of two main section, the conditioning circuit and the acquisition system. The chosen conditioning circuit to convert the resistance variation of the bend sensor into a variable voltage is the Wheastone Bridge. Also a typical voltage divider has

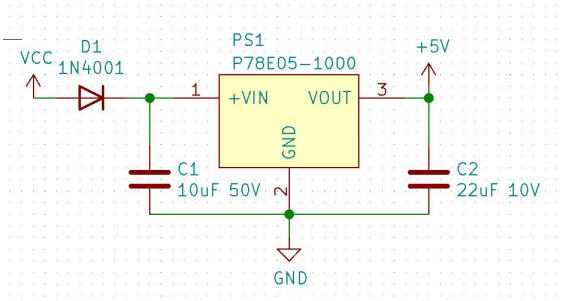

Figure 7.4. LDB Circuit Schematic Connectors Section.

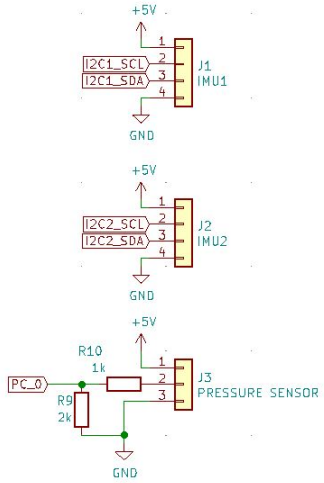

Figure 7.5. LDB Circuit Schematic DC-DC Section.

been test but with poor results due to the low sensitivity at low bending angle. This problem and its solution is further discussed in the section related to the characterization of the RBS. The acquisition system is composed by the ADC1115 16-bit analog to digital converter. It has been selected becasue it high resolution and internal Programmable Gain Amplifier (PGA) that allow full scale range as low as 256 mV. It can sample from 8 to 860SPS, providing a resolution of 7.8uV with a maximum error of 0.15% at 8 SPS. This turns out to be the perfect solution to acquire the differential voltage coming from the Wheatstone Bridges because, given the low sensibility at low bending angle , the output voltage has a range of 20mV for 0 to 10 bending degree. This integrated circuit also integrate digital programmable low pass anti-aliasing filter to reduce the bandwidth of the input signal and an I2C controller to handle the communication with the host MCU.

To produce a printed circuit board the manufacturer need some production file

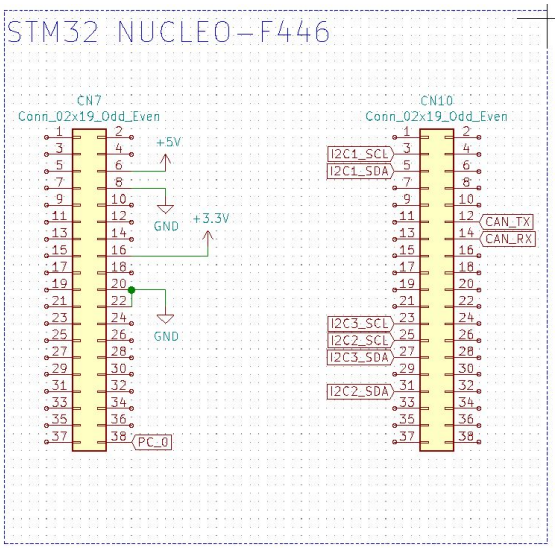

Figure 7.6. LDB Circuit Schematic Header Pin.

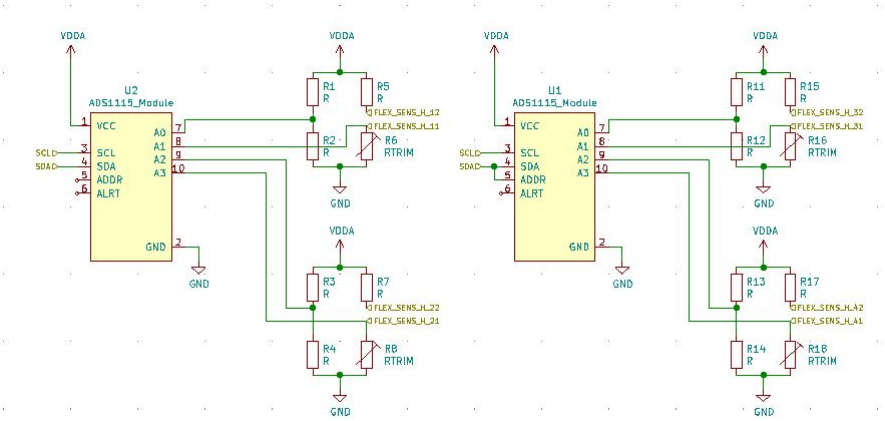

Figure 7.7. LDB Circuit Schematic RBS Conditioning Circuit.

called GERBER file. Those file are not generated from the schematic because a successive step is needed. In fact, the schematic reports the components and how they are connected among them, but does not define the geometrical disposition nor the physical dimension of the board. The following figure shows the results after having associated the footprint to each component and after the disposition and routing procedure.

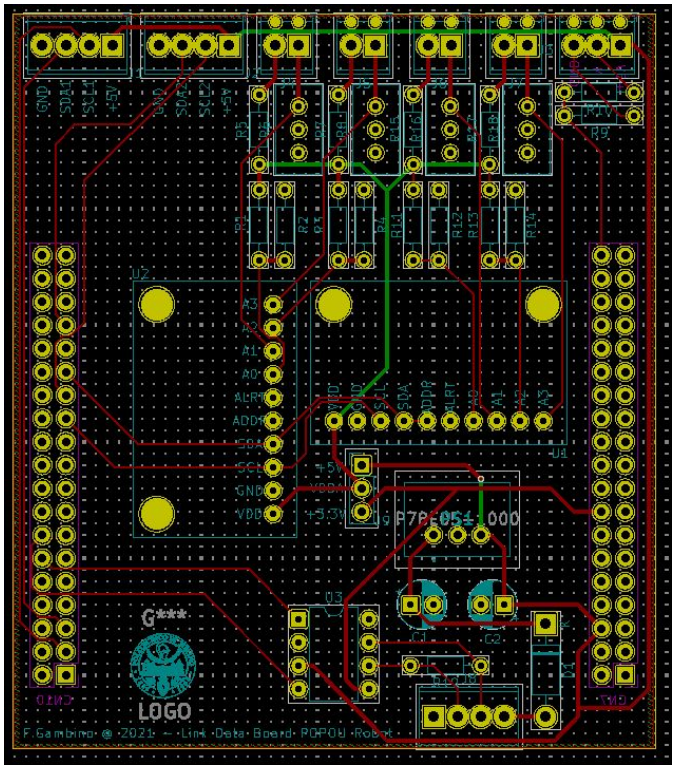

Figure 7.8. LDB PCB Routing.

At this point the GERBER files have been sent to the manufacturer that produced and delivered the board in less than a week. Fig 7.9 and 7.10 show respectively the expected result, thanks to the CAD models of the components, and the real board with some components already installed and tested.

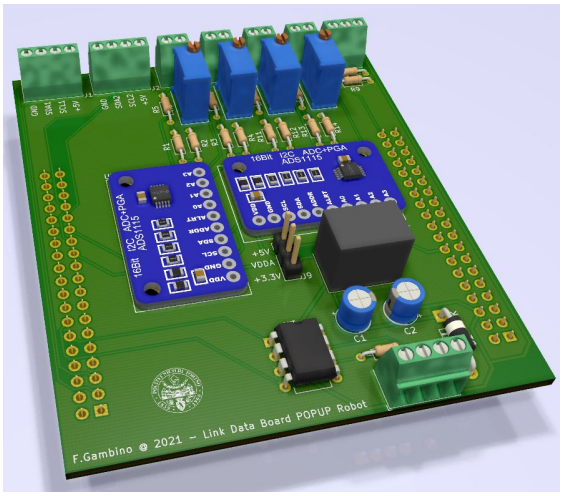

Figure 7.9. LDB 3D CAD Model.

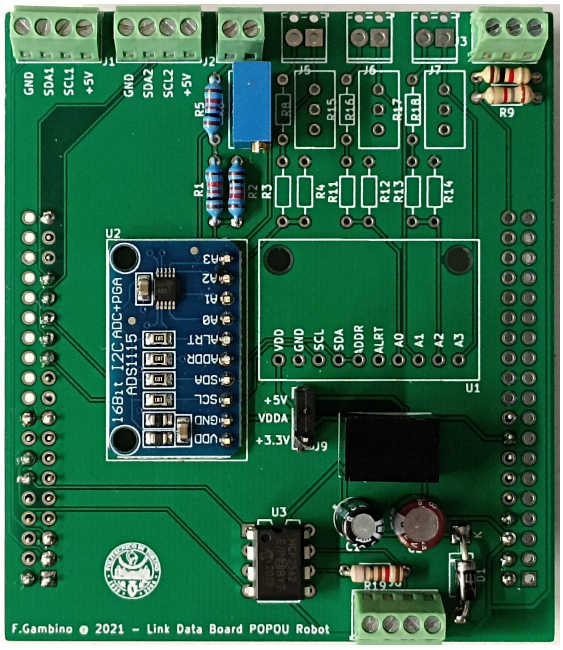

Figure 7.10. LDB Printed Circuit Board.

7.2 Firmware Development

The firmware for the LDB has been developed using the Arduino core support for STM32, that simply speaking adds the support of STM32 MCU in Arduino

IDE. This choice was made mainly for compatibility reason. In fact, the C library used to acquire data from the IMUs and to estimate the orientation applying the Madgwick filter has been entirely written to be used with Arduino. A porting procedure would have taken too long to be effectively implemented in this thesis. As for the MCB firmware, the complete source code is listed in Appendix D and a flow diagram of the LDB firmware is also illustrated in Fig, 7.11 at the end of the section. Generally speaking the programming approach is very similar to the one discussed for the MCB. After power up, the serial peripheral is configured and initiated. Then is the time of the I2C peripherals, one for each imu and one for both the external ADCs for a total of 3 I2c peripheral used. Subsequently the CAN peripheral is initialized with 1 Mbps as speed setting. The library that handle the CAN peripheral is self-written starting from the HAL provided by ST. If the CAN Transceiver is installed and works correctly, the CAN peripheral initiate correctly and the program can go on. At this point of the program, all the needed timer are defines, similar to what is done in the MCB. TIM3 is used to check if new CAN messages have been received,with an overflow frequency of 100 Hz. Other two timers are defined, the ones for the CAN transmission and the one to precisely execute the bending estimator, and, therefore, acquire the IMUs and the flex sensors. But those timers are not started yet, with respect to TIM2. From now on the LDB is able to receive CAN messages. To terminate the first part of the initialization procedure, the MCU execute two function. Init_IMUs() which verify the presence of the IMUs executing a class function that configures the IMUs. If the IMUs are found , both return 1. The imus_status variable is the multiplication of imu1_status and imu2_status and is 1 if both IMUs are ok. if IMU1 is not connected, imu1_status become 4. At the same manner, if IMU2 is not connected, imu2_status become 5. So imus_status might be 4 or 5 if one of the two IMUs is not connected or 20 if both IMUs reports an error. This encoding procedure is needed to send a single data byte representing the status of both IMUs. Then init_ADCs() is executed. In a similar manner, it retrieve the condition of the ADCs and set it in a global variable adc_Status. At this point the MCU execute the measurements of the link pressure to terminate the formation of the status message. From this moment the LDB starts sending a CAN message with id 20 containing all the information retrieved previously about the status of the sensors and the inflation pressure of the link. This packet is sent every second while the MCU wait for a check message from the MCB.

When the MCB execute the status check procedure and receive a status message from the link board it will answer with a status check if the Link status is equal to 1, which means the the link pressure is normal, both IMUs are correctly initialized and the ADCs are ready to acquire from the Wheatstone bridge. If all this condition are met, the MCB send a status check message with ID 48. If the LDB receives this packet, a status check flag is set and the program execution can

continue. But only for a short time. At this point the MCU enters again in a loop waiting for a calibration command from the MCB. When this command is received as a CAN message with ID 50 the program exit from the loop and execute the calibration of the IMUs and the compensation of the RBS offset. Both procedures are further discussed in the next section. Once terminated the calibrations, a CAN message with ID 40 is sent, meaning that the calibration is completed. After a delay of 1 second the timers previously initialized are started. At 1 kHz rate the MCU starts to executed the Bending_sensor_fusion function. This function is the core of the LDB and it is divided in 3 steps. First step, execute the function BendingAngleIMUs(). This function call the command mpu.update which make the orientation filter to compute a new value. Once computed, the function make the difference between the orientation components of both IMUs At this point the global variable imu_bending_angle_horiz is imposed to be equal to the orientation difference among the pitch estimations while the imu_bending_angle_vert is equal to the difference between Yaw estimations. The second step is the execution of the function BendingAngleRBS(). This function is used to acquire differential voltage reading from the external ADC, averaging them and to compute the bending angle seen by the RBS by applying the Characteristic function that has been developed. The third step consist in the actual sensor fusion. At this point a large variety of approach are possible, as Kalman Filter Estimators. However this would have requested long implementation time and computational load. The approach that have been tested simply take the average between the two estimation. The results are discussed in the next Chapter.

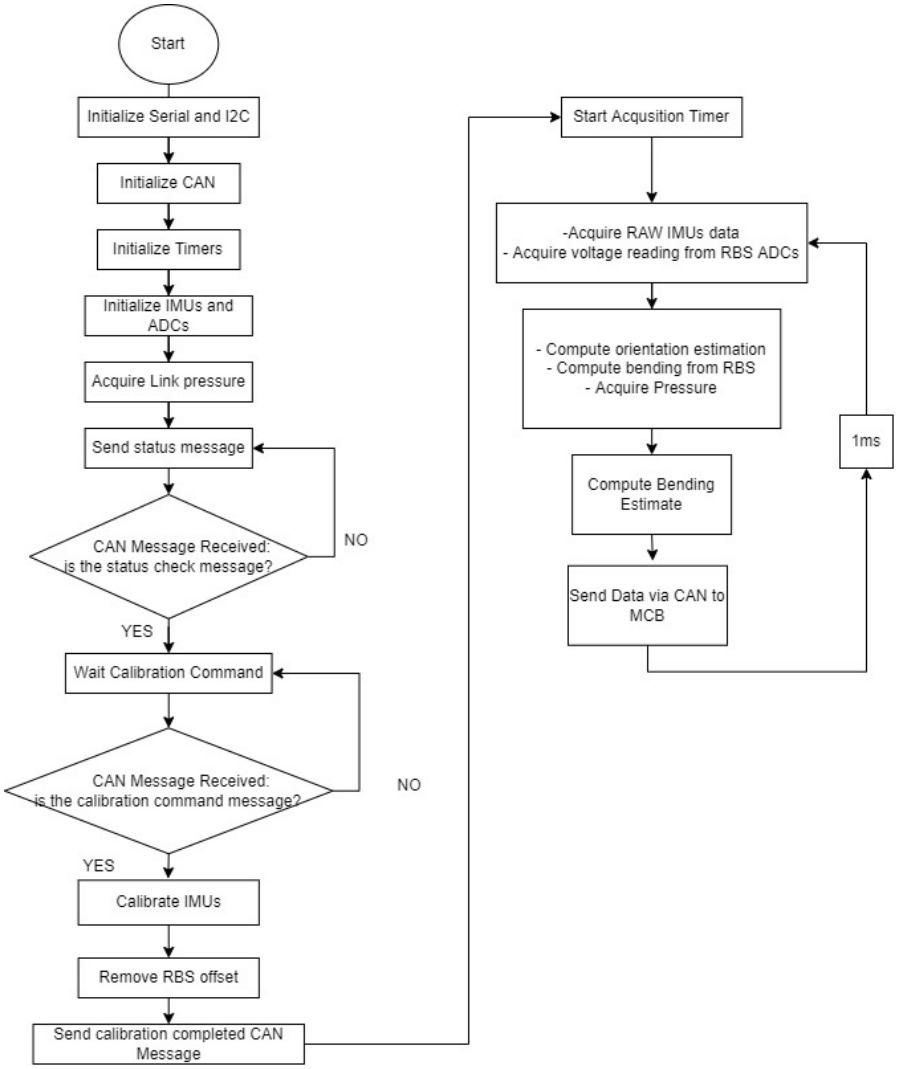

Figure 7.11. Firmware Flow of JDB.

7.3 IMUs Calibration

As explained in the dedicated section, the IMUs used are based on MEMS (micro electro mechanical systems) technology. Low cost MEMS based IMUs, are the ones used for this thesis work, are usually affected by non accurate scaling, cross-axis sensitivities, sensor axis misalignments and non zero biases. The calibration procedure refers to the process of identify those quantities. For Accellerometer and Gyroscope the procedure is quite simple. First of all the IMU is configured for bias calculation. Then raw reading from accellerometer and gyroscope are accumulated

in temporary variables. Once the end of sample accumulation, a simple average is computed from the sample readings. Those values are then pushed into the offset hardware register of the IMU. Fig 7.12 and Fig. 7.13 show the uncalibrated readings prior the calibration procedure. After the offsets are applied the raw output data become as shown in Fig 7.14 and 7.15

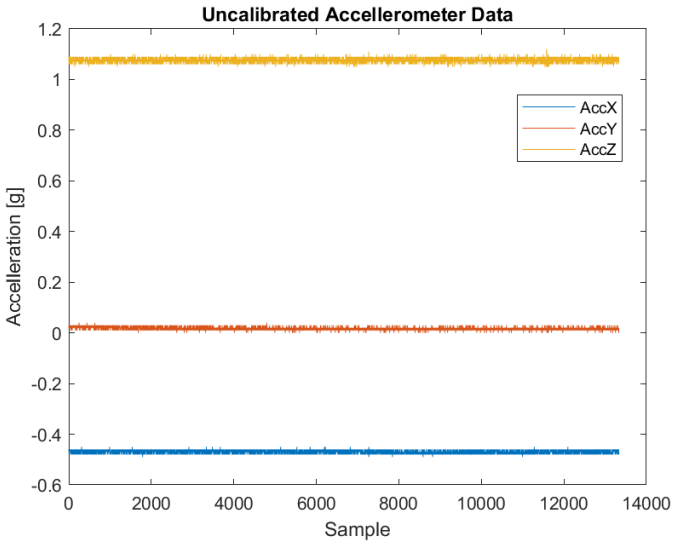

Figure 7.12. Uncalibrated Accellerometer readings.

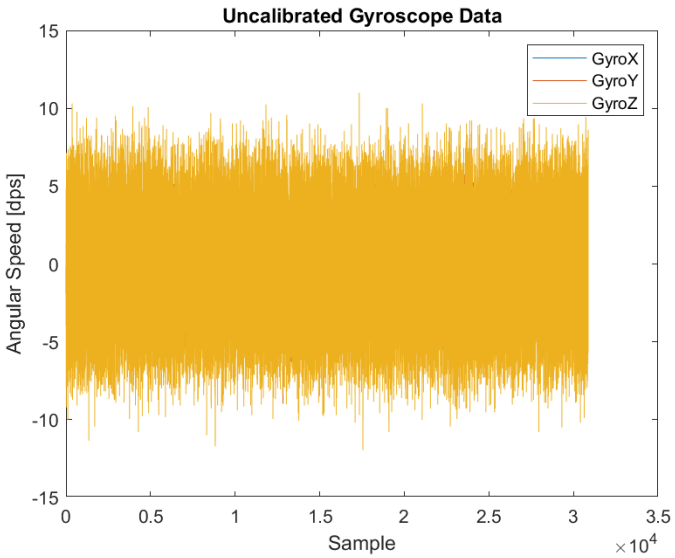

Figure 7.13. Uncalibrated Gyroscope reading.

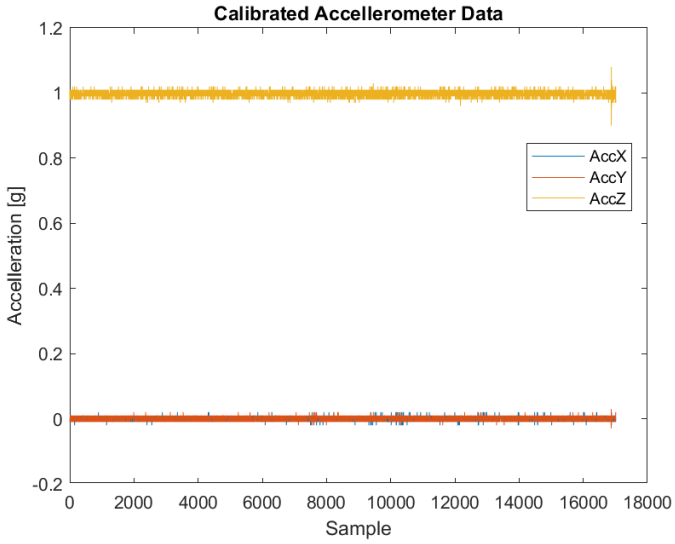

Figure 7.14. Calibrated accellerometer readings.

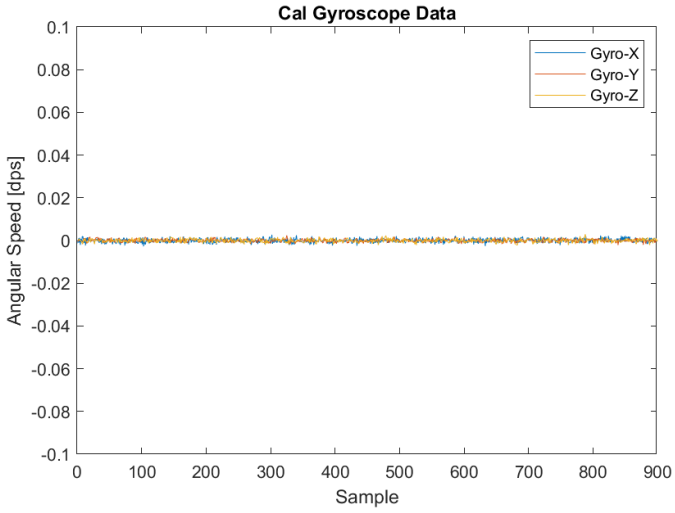

Figure 7.15. Calibrated Gyroscope reading.

As previously explained, the Magnetometer detect the strength of the magnetic field along sensor's X,Y and Z axes. In order to estimate the absolute orientation, accurate magnetic field measurements are essential. Typical low-cost MEMS based magnetometer as the one used for this work need to be calibrated to compensate for environmental noise an manufacturing defects.

An ideal 3-axis magnetometer in a magnetic-interference free environment measures the Earth0s magnetic field. If the magnetometer is rotated in every direction during the data acquisition the measurement should lie on a sphere whose radius represent the magnetic field strength. A real magnetometer will never be in this condition, and generally, two offset always affect its measurement: Hard Iron and soft Iron offset. Hard Iron effect are stationary interfering magnetic noise source, often produced by metallic object posed near the magnetometer. The effect of Hard Iron offset is a shift of the origin of the ideal sphere. The other source of noise are called Soft Iron effect. They arise from object near the sensor which distort the surrounding magnetic filed, causing the ideal sphere to become an ellipsoid. To correct these offset is possible to use the function magcal provided by the Sensor Fusion Toolbox by MATLAB. Fig. 7.16 shows the RAW magnetometer data acquired from the IMU while rotating it in all the possible direction. Both Hard Iron and Soft Iron effect are observable. Fig 7.17 illustrate the result after the correction has been applied. The readings now lie on a perfect sphere centered in the origin. From the given raw magnetometers data, the magcal function computes a 3-by-3 real matrix A and a 1-by-3 vector b by using a variety of solvers.

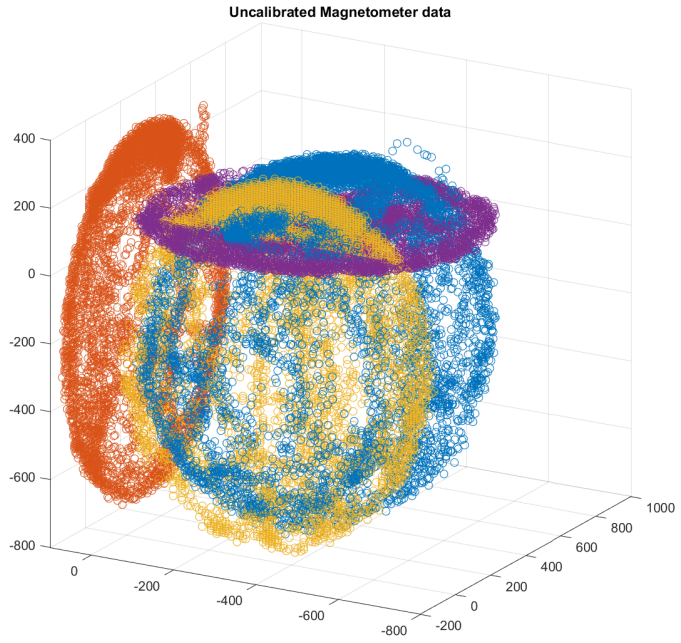

Figure 7.16. Uncalibrated magnetometer readings.

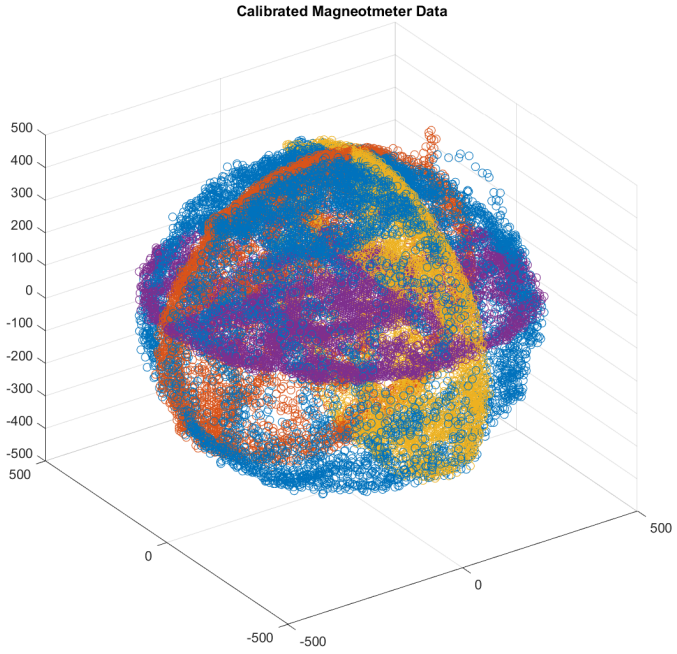

Figure 7.17. Calibrated Magnetometer readings.

7.4 RBS Characterization

Sensor characterization is the process of taking measurements from a transducer under controlled conditions. It is necessary to ensure a level of accuracy over various operating conditions.Typically the manufactures sells sensors with a characteristic curve that can be used to find a correlation between the sensor output and the wanted measured value so that to use the sensor effectively. However sometimes the manufacturer does not publish the sensor characteristic or it varies to much among different product from the same production line. This is the case of the Resistive Bend Sensor used in this thesis work. The manufacturer only states some specification such as the resistance at flat, its tolerance and the resistance variation range. No information are available about how th]e resistance vary with respect to the bending angle. In [5], an experimental approach is taken to perform the characterization of a resistive bend sensor. The results demonstrated that the characteristic is highly non-linear. In particular, the sensitivity at low bending angle is very low. To acquire the characteristic of the sensor was used a bench precision multimeter. The manual of the GWINSTEK GDM-8245 states that the accuracy of the resistance measurement for the range 5kΩ to 500kω is

$$\pm(0.1\%Reading + 2digits) \tag{7.1}$$

As done in [5], a device was designed to hold the sensor at a specific angle of bend. Fig. 7.18 shows the CAD of the device. The principle is to exploit the known pitch of a screw to act at a certain distance from a hinge. An M3 screw has a typical pitch of 0.5mm. The distance between the point of application and the hinge was designed in such a way that one turn of the screw corresponds to 0.5 degrees to the hinge.

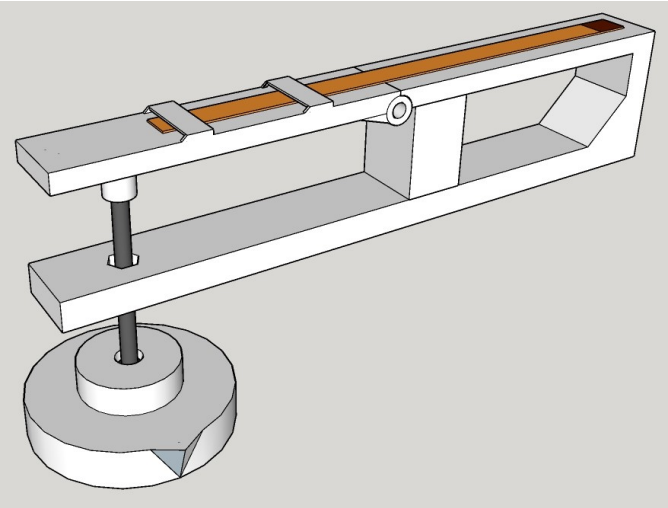

Figure 7.18. CAD model of the device used to characterize the RBS.

The CAD Model has been converted into STL file to be produced using a 3D printer. The material choice was PLA since it is not subject to particular strain. The printing procedure takes about 45 minutes. Fig. 7.19 and 7.20 show the device installed on a table with the RBS sensor already installed on it. A simple piece of elastic is glued on the end to the hinge arm so that the hinge is mantained pressed against the screw cap.

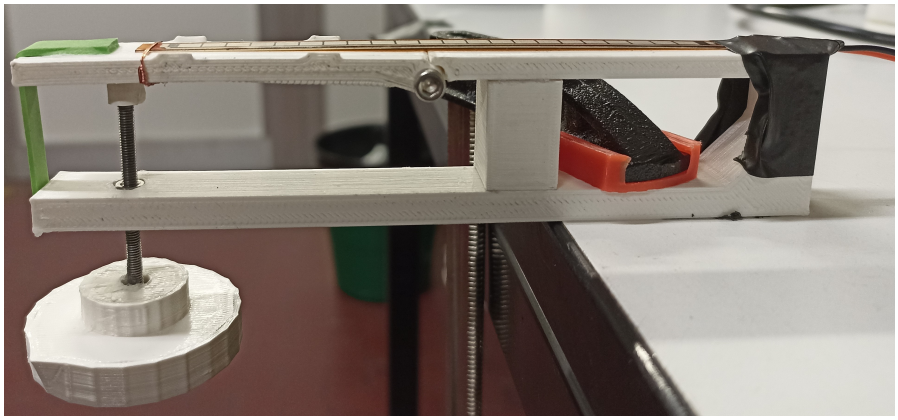

Figure 7.19. Device used to characterize the RBS at zero angle.

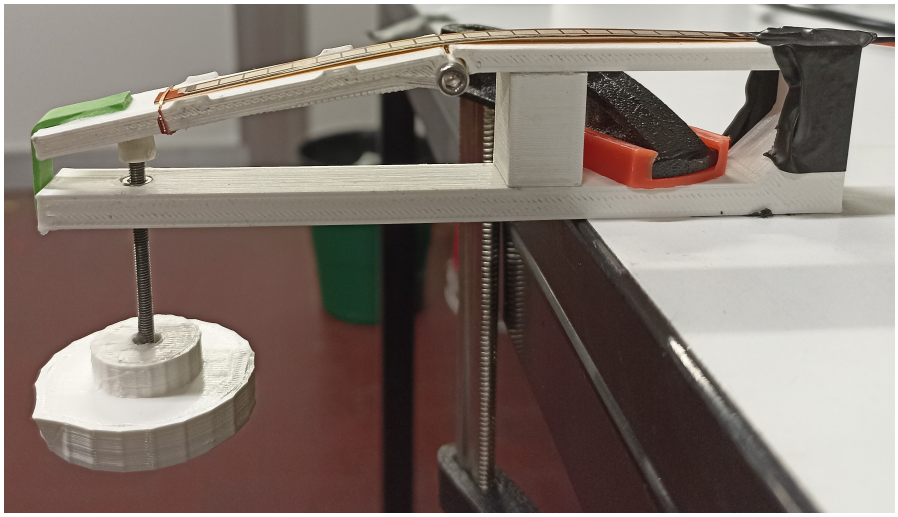

Figure 7.20. Device used to characterize the RBS at 10° angle.

With the sensor able to maintain a stable position the characterization procedure started. The procedure has been repeated two times. The first time it was decided to give an angular step of 2°. After switching on the bench multimeter and waiting a few minutes for the circuitry to come up to temperature, I started the measurements. Since it was not possible to automate the procedure, it was decided to acquire a single point, as the value on the screen remained constant during the tests.

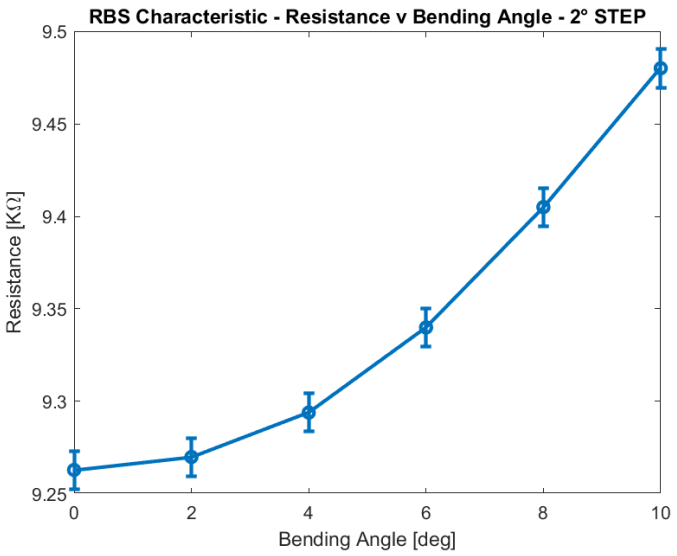

Figure 7.21. RBS Characteristic at 2° STEP

As expected, the characteristic results non linear in proximity of zero bending angle. Here the sensitivity of the sensors drop at very low value. The uncertainty has been computed considering only the accuracy of the digital multimeter. For the zero-angle, the measurement with uncertainty is

$$R_0 = (9.263 \pm 0.0103)k\Omega \tag{7.2}$$

The procedure has been repeated but giving an angular step of 0.5°. Also in this case it was evident the non-linearity at low bending.

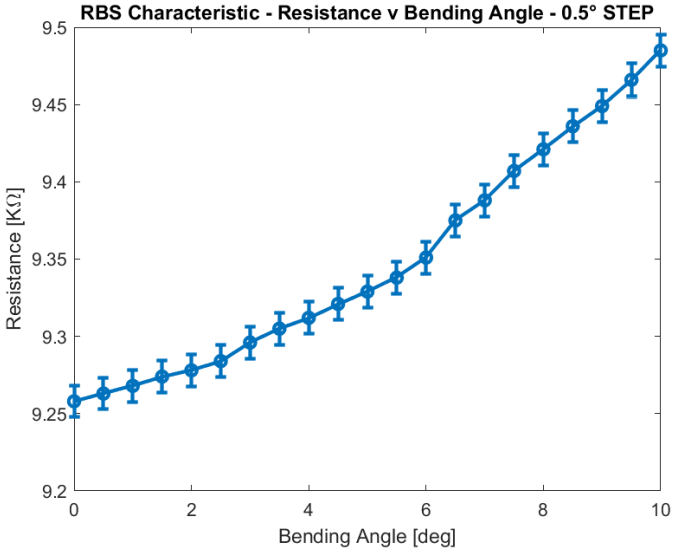

Figure 7.22. RBS Characteristic at 0.5° STEP

At this point a fitting procedure has been performed to find a mathematical relation between the measured resistance and the bending angle.

The resulting equation is the following.

$$R(x) = 0.00225x^2 - 0.000557x + 9.262 \tag{7.3}$$

SSE: 1.77e-5
R-square: 0.9995
RMSE: 0.00243

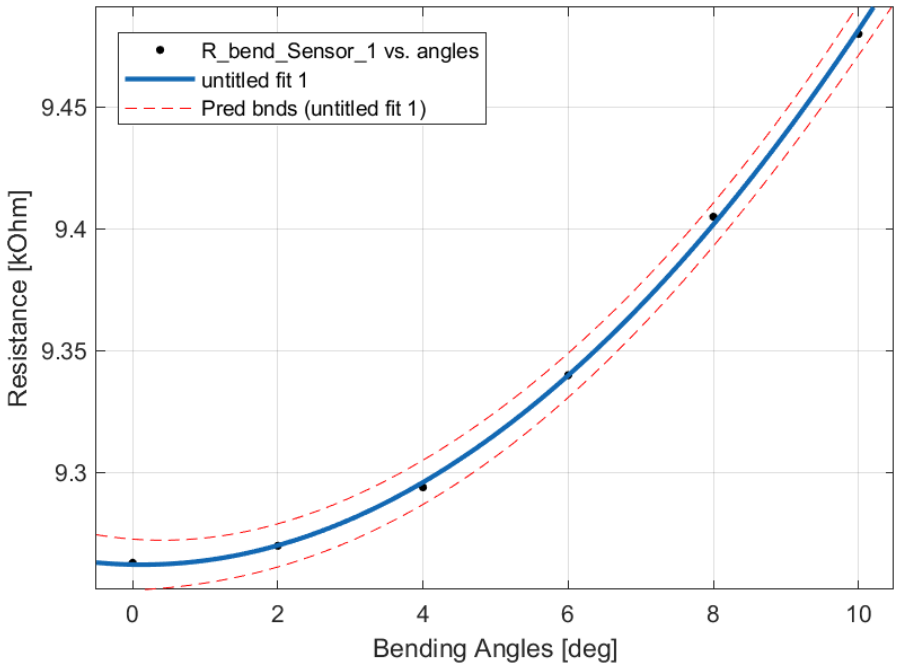

Figure 7.23. RBS Characteristic fitting

The fitting procedure has been repeated also for the characteristic taken every 0.5°. The resulting characteristic function is reported below.

$$R(x) = 0.00165x^2 - 0.00045x + 9.259 \tag{7.4}$$

SSE: 0.000235
R-square: 0.997
RMSE: 0.00364

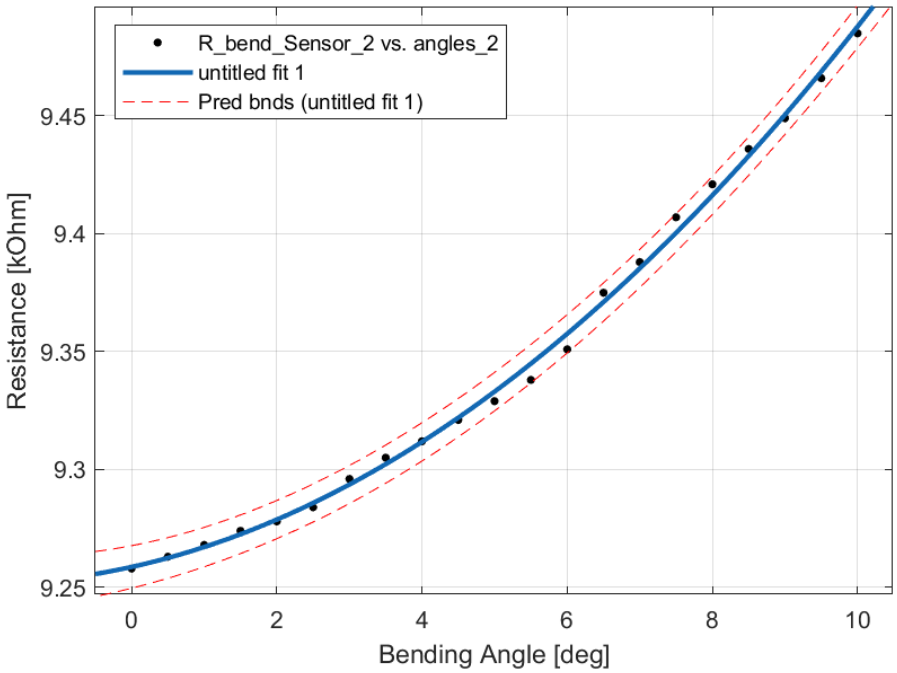

Figure 7.24. RBS Characteristic fitting

Part III

Third Part

Chapter 8

Implementation and Testing

8.1 Assembled Popup Robot

The links are manufactured using a soft PVC reinforced foil, folded over itself to obtain a pipe and fixed by using glue. The extremities are closed with two rigid caps on which the 3d printed joints can be screwed. Fig. 8.1 and Fig. 8.2 show the 3d printed joints installed on the motors and and links. In Fig. 8.1 are also clearly visible the compressed air lines (in blue) and the wiring harness protected by a plastic spiral.

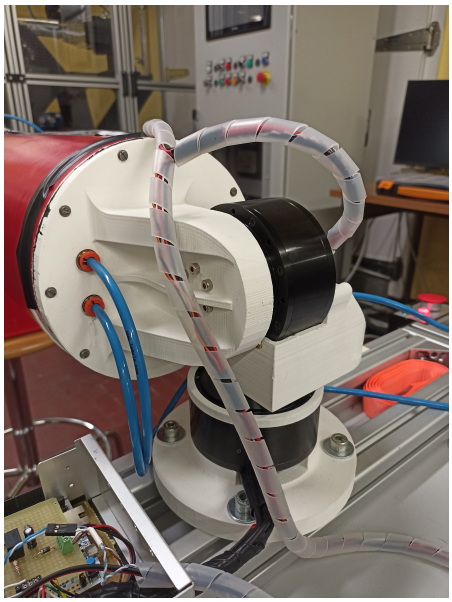

Figure 8.1. Assembled Popup Robot - Joint 1 and Joint 2.

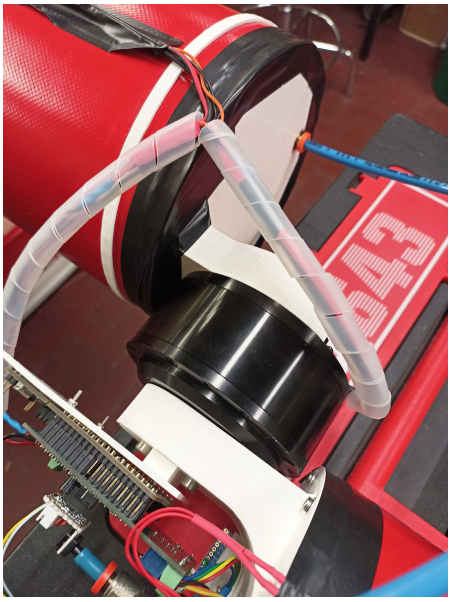

Figure 8.2. Assembled Popup Robot - Joint 3.

Fig. 8.3 to 8.5 show the manipulator in its entirety from different angles. Fig. 8.4 and 8.5 have been taken after the homing procedure.

Figure 8.3. Assembled Popup Robot.

Figure 8.4. Assembled Popup Robot.

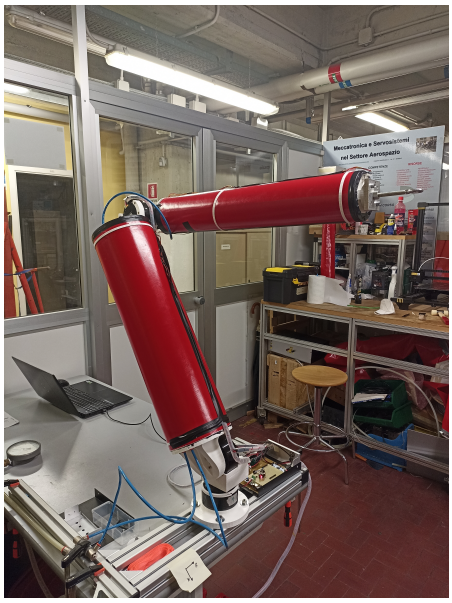

Figure 8.5. Assembled Popup Robot.

8.2 Wiring Harness

The cable harness is the assembly of electrical wires used to transmit power and signal. Harnessing provide many benefits as connection stability, optimized space utilization, resistance against vibration. abrasion, moisture etc. The connection diagram in figure 8.6 represents a diagram of the connections between the various components of the electronic system.The connectors used and their respective pin numbers are also listed at the bottom left.

Fig. 8,7 and Fig 8.8 show the MCB installed in a metallic housing and the LDB of the link 2 installed just near the link cap.

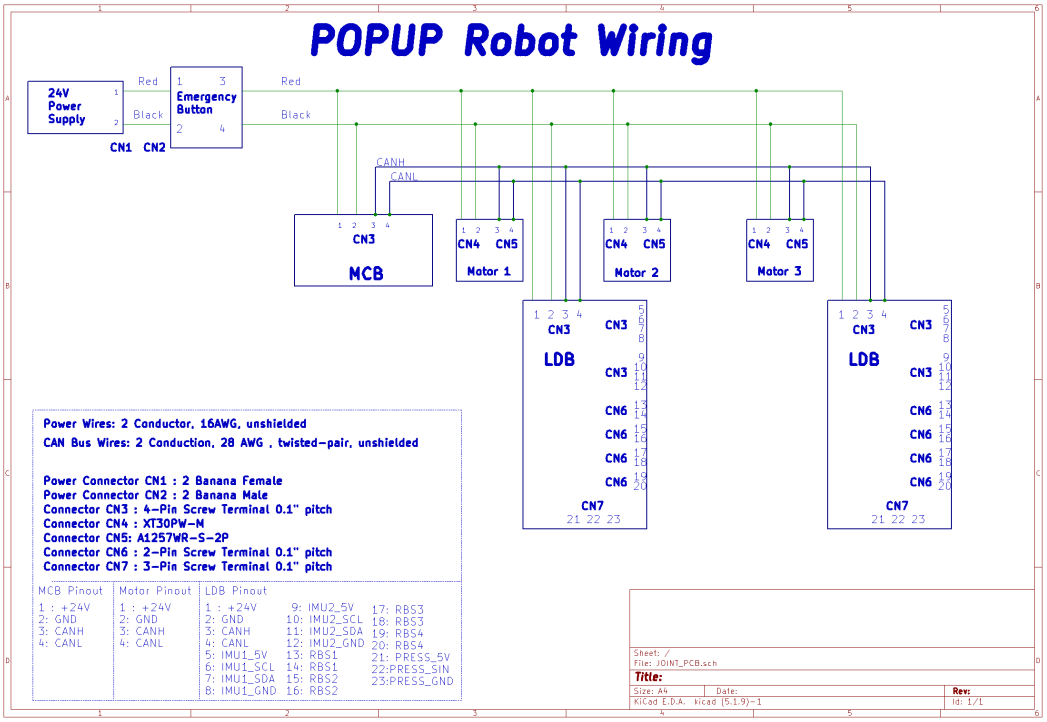

Figure 8.6. Wiring harness scheme.

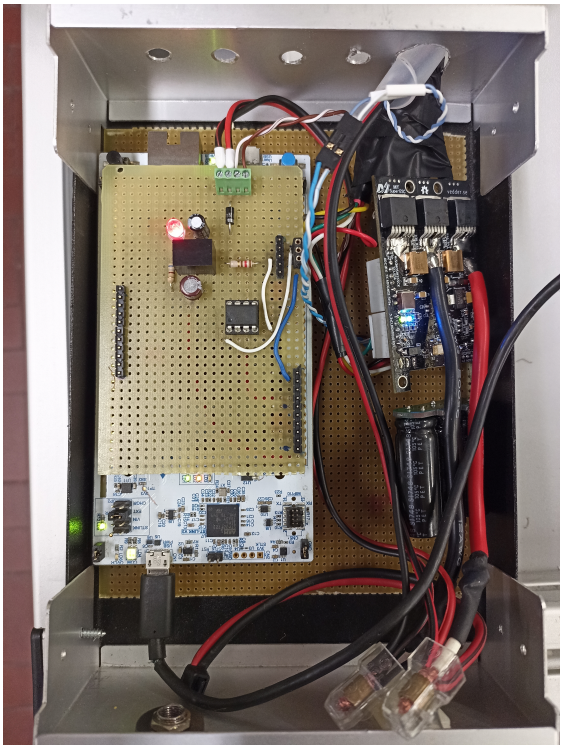

Figure 8.7. MCB installed on the basis of the manipulator.

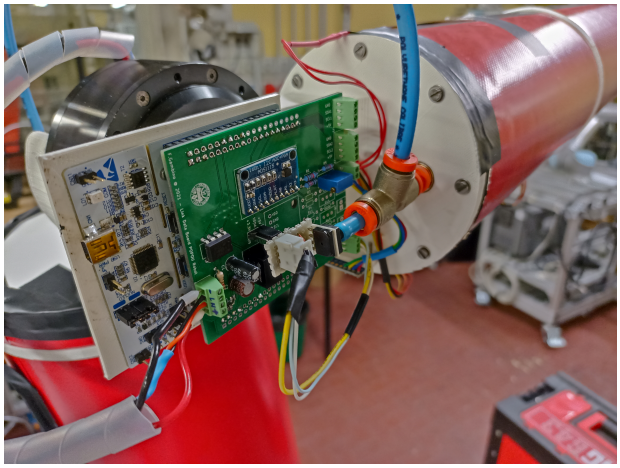

Figure 8.8. LDB installed on link 2.

8.3 Main Controller Testing

The system test procedures mainly focused on verifying the communication between the MCB and the LDB, the impact on the manipulator's dynamics as the controller parameters changed and verifying the acquisition of data from the LDB. Fig 8.9 is the screenshot of the serial console used to communicate with the MCB. Looking at it is possible to reconnaise all the step previously discussed about the initialization procedure.

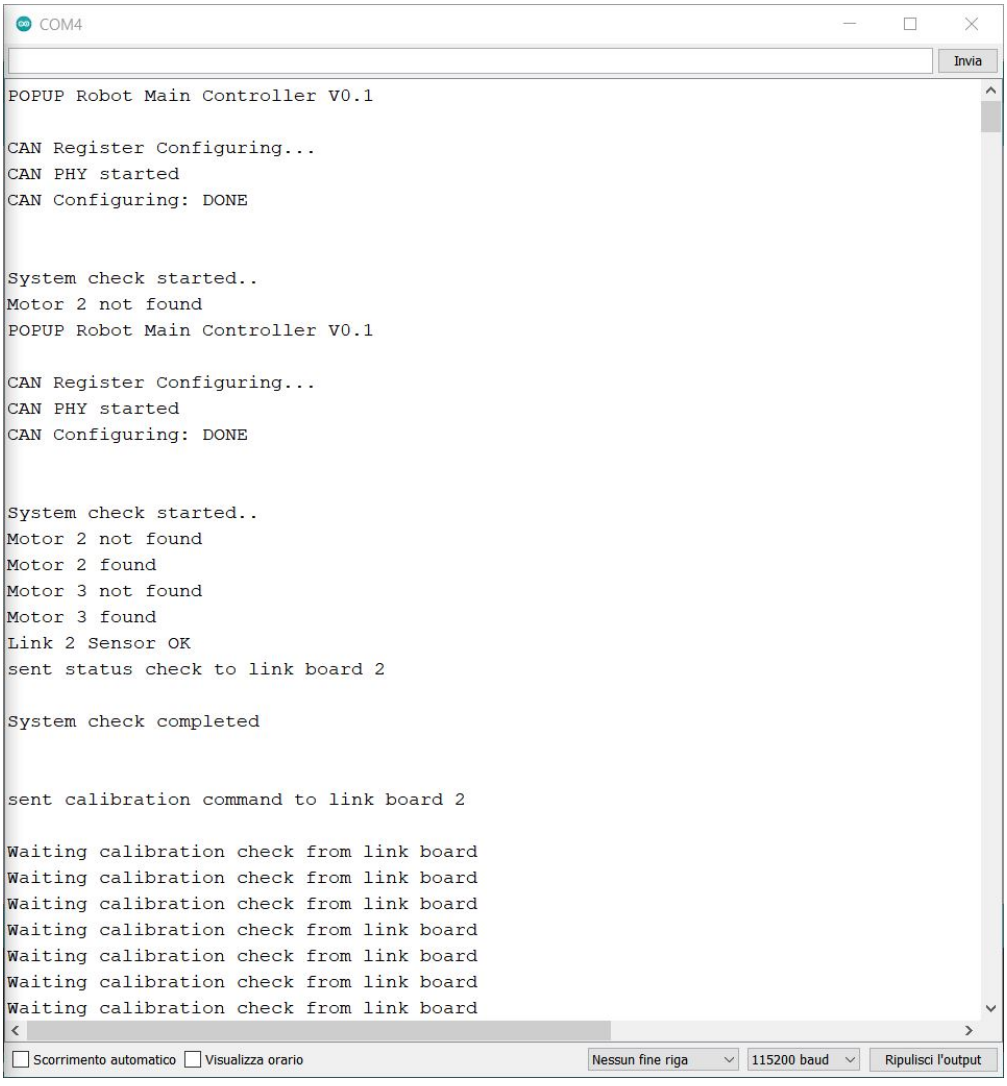

Figure 8.9. MCB serial console output.

Fig. 8.10 has been taken using a Logic Analyzer to verify the CAN packet on the CAN Bus. Starting the acqusition just before powering up the system allowed to acquire the complete initialization procedure where the MCB and the LDB exchange some CAN message prior the execution of the homing function.

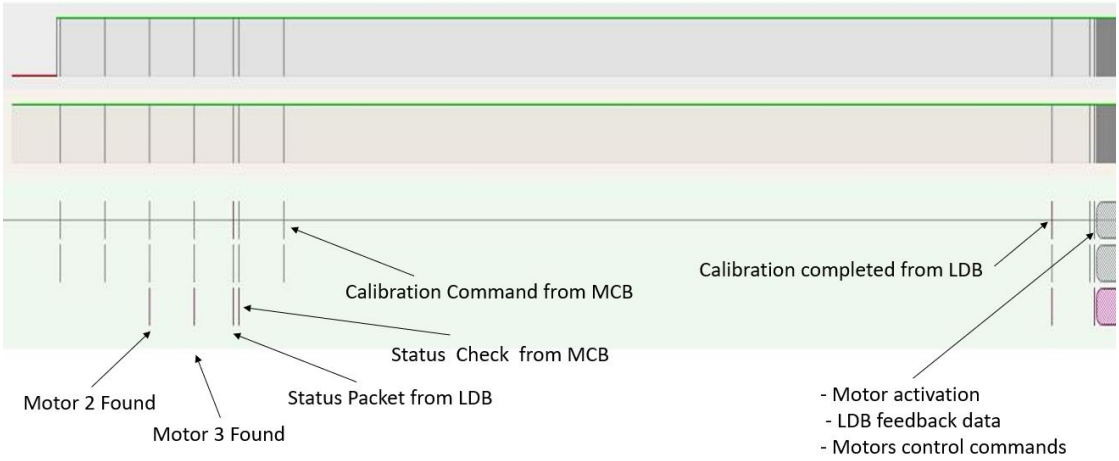

Figure 8.10. Initialization procedure observed using a Logic Analyzer.

Once the experimental verification of the communication between LDB and MCB was completed, we moved on to the analysis of the telemetry data. Fig. 8.11 to 8.14 show the system response to a planned trajectory with the following parameter imposed for the controller. For the position controllers the parameters are

Param	KP	KI	KD
Q1	0	0	0
Q2	40	0.005	0
Q3	40	0.005	0

and for the speed controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	5	0	0
Q3	5	0	0

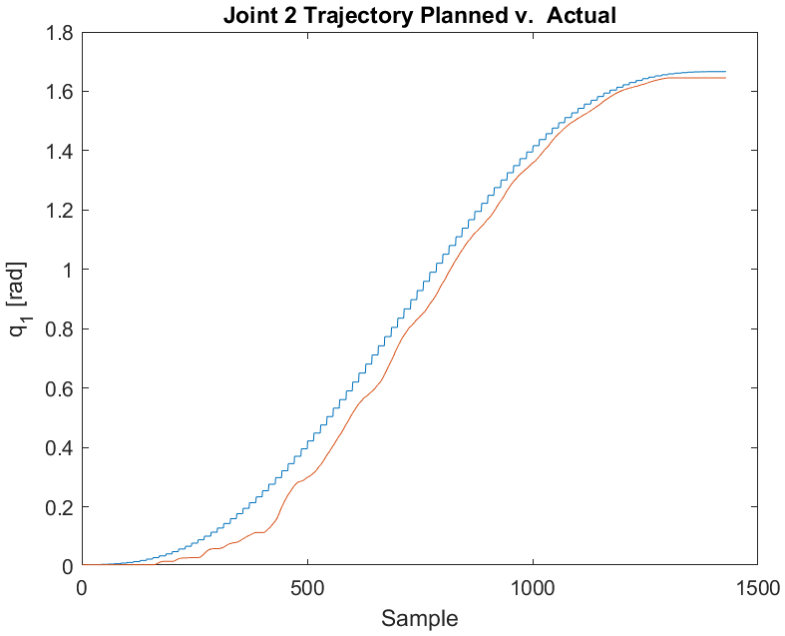

Figure 8.11. Position Set and feedback data from Joint 2.

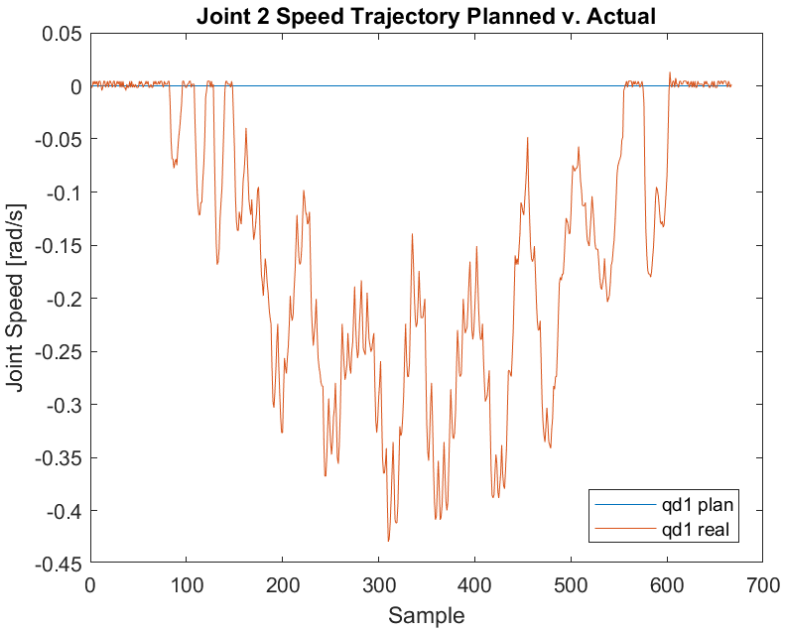

Figure 8.12. Speed Set and feedback data from Joint 2.

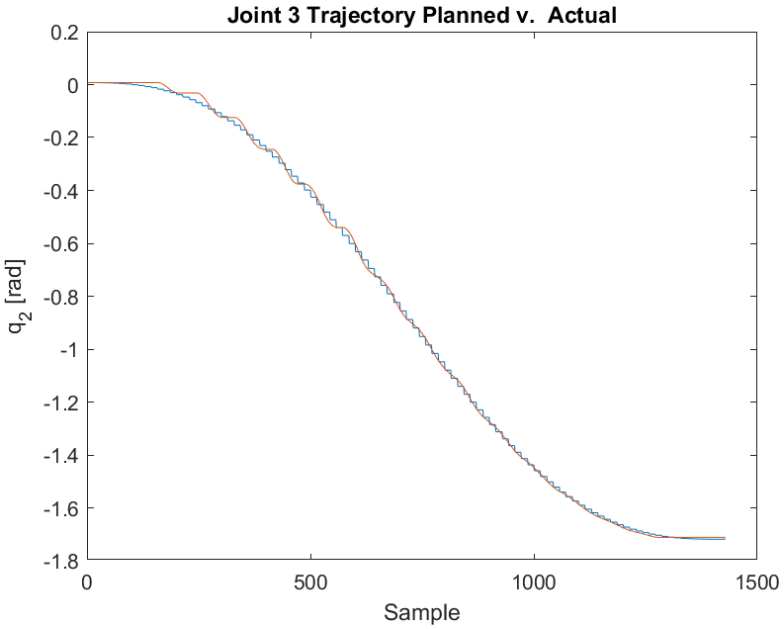

Figure 8.13. Position Set and feedback data from Joint 3.

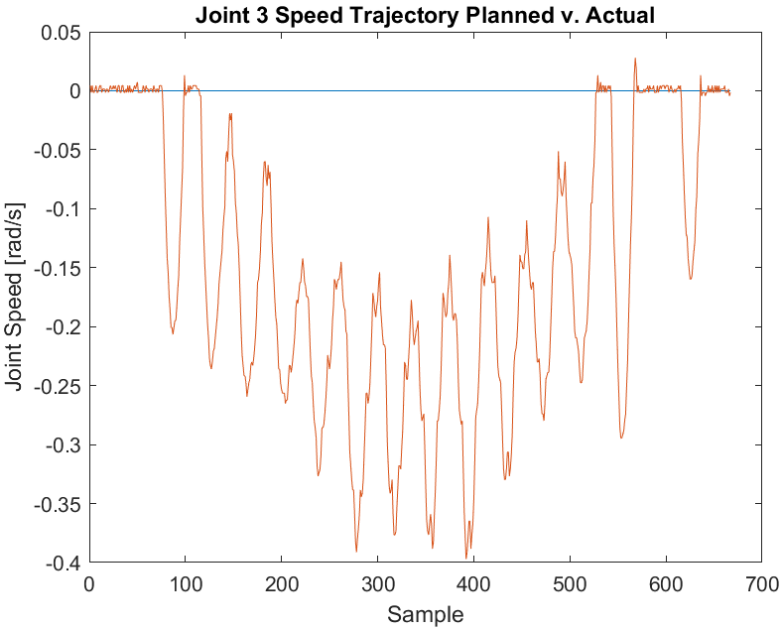

Figure 8.14. Speed Set and feedback data from Joint 3.

Its cleary visible how those parameters are far to be optimal. Even if the final positioning is acceptable, during the movemente the manipulator vibrates, as evident from the speed feedback. Different set of parameters have been tested to find, with trial and error, a set of parameter giving the wanted performance. For the position controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	5	0	0
Q3	5	0	0

and for the speed controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	10	0	0
Q3	10	0	0

gives as response the following.

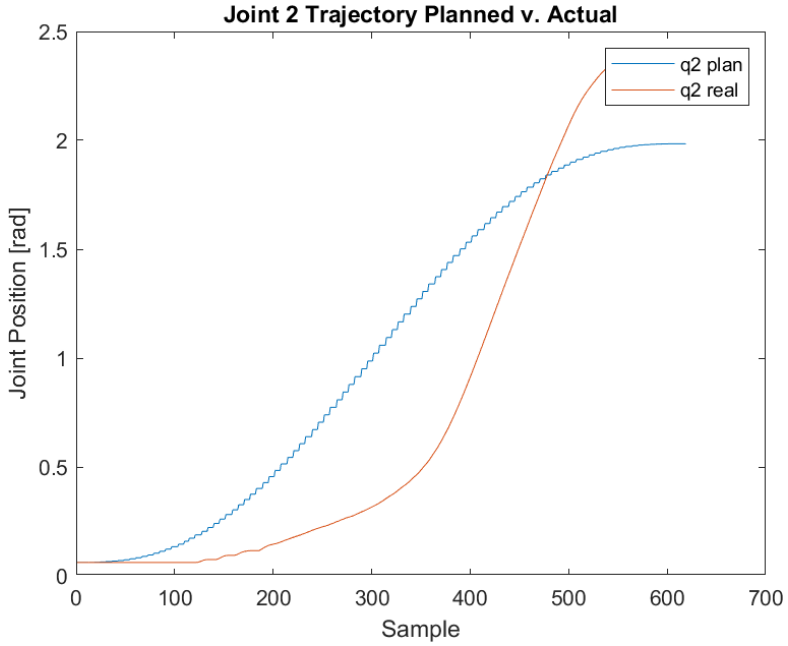

Figure 8.15. Position Set and feedback data from Joint 2.

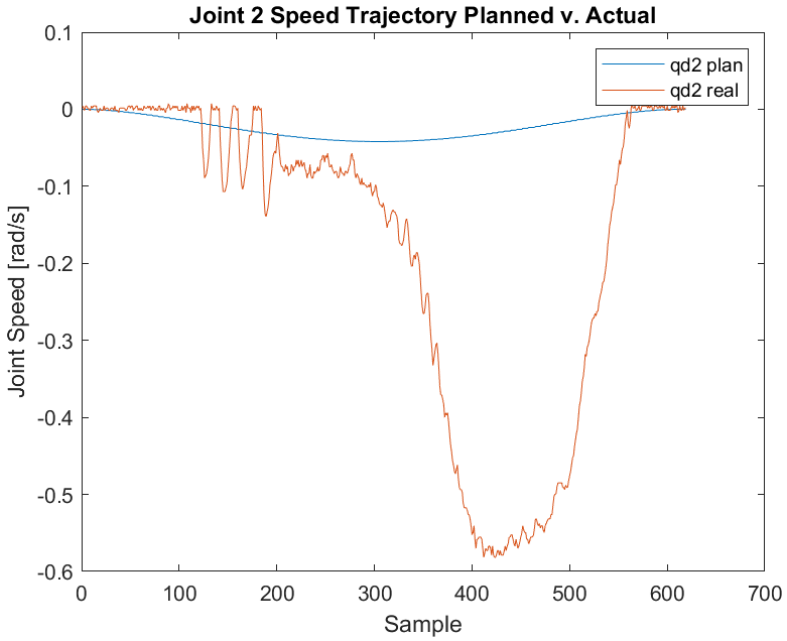

Figure 8.16. Speed Set and feedback data from Joint 3.

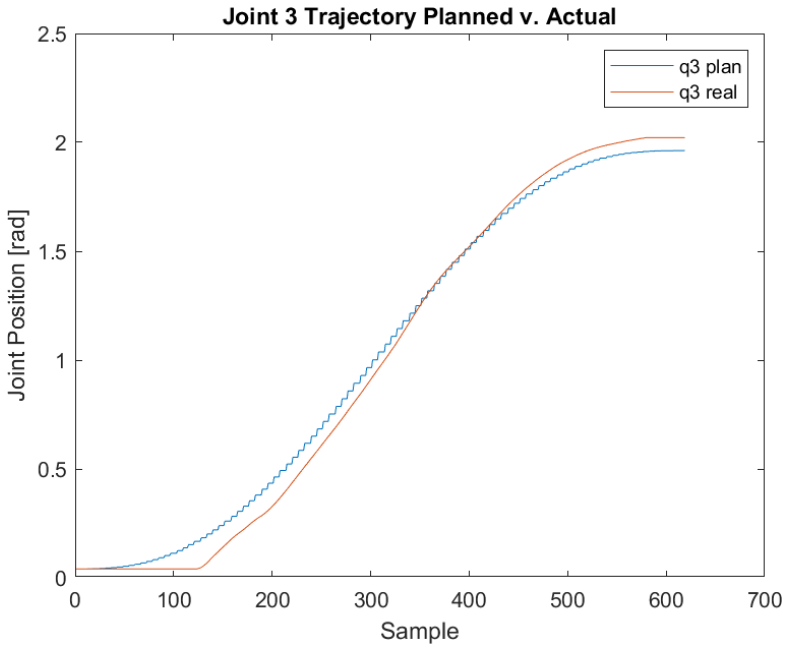

Figure 8.17. Position Set and feedback data from Joint 3.

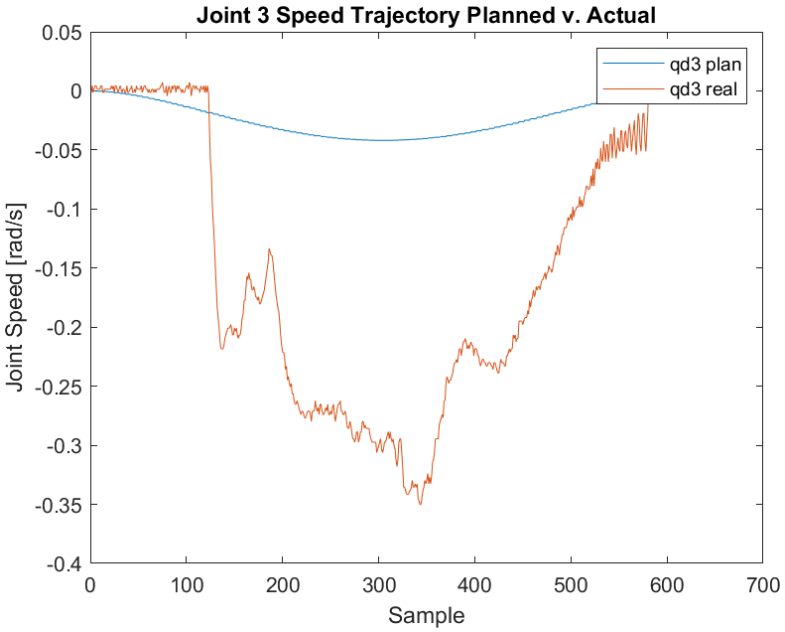

Figure 8.18. Speed Set and feedback data from Joint 3.

The behaviour is a bit better because there are no evident oscillation caused by the controllers. However a strong error is now present at steady-state. To solve, another set of parameters has been tested. For the position controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	10	0	0
Q3	10	0	0

and for the speed controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	10	0	0
Q3	10	0	0

The results are illustrated from Fig. 8.19 to Fig. 8.22.

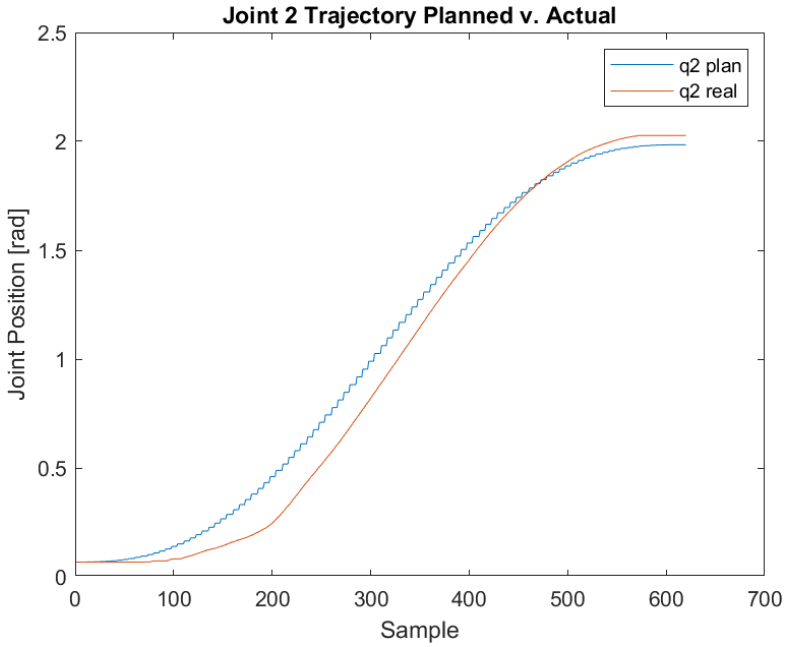

Figure 8.19. Position Set and feedback data from Joint 2

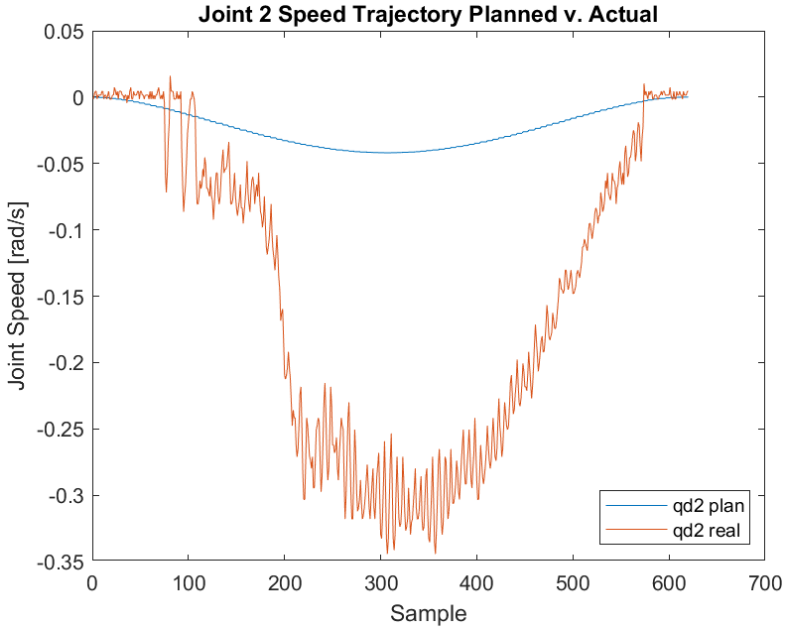

Figure 8.20. Speed Set and feedback data from Joint 3

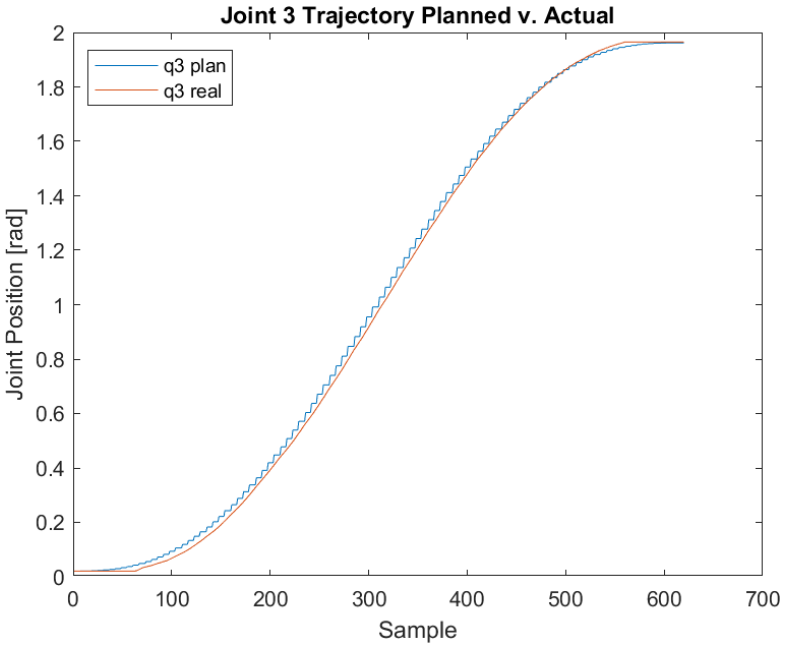

Figure 8.21. Position Set and feedback data from Joint 3

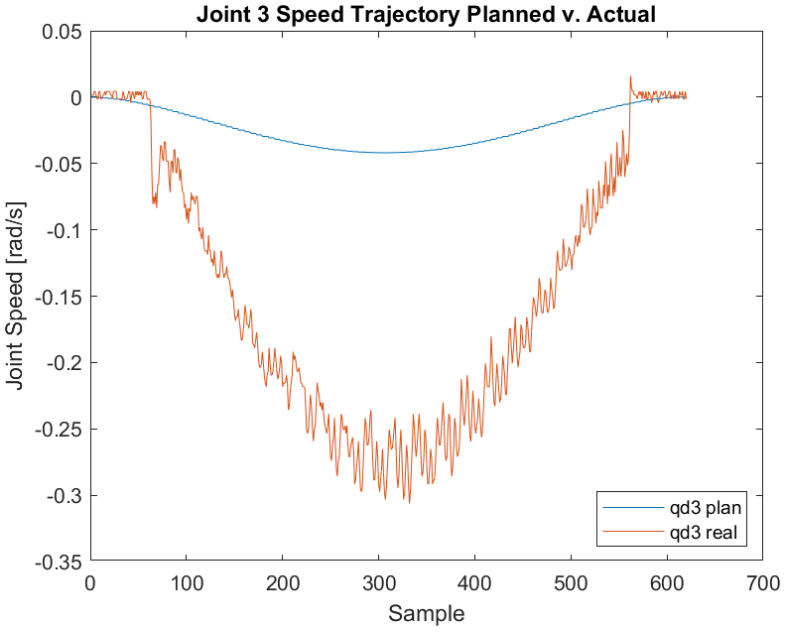

Figure 8.22. Speed Set and feedback data from Joint 3

With respect the previous response the steady state error is lower and no oscillation occurs. So the direction on which the parameters have been modified is correct. Trying with some fine tuning leads to the figure below. The used parameter have been, for the position controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	5	0.005	0
Q3	5	0.005	0

and for the speed controllers

Param	KP	KI	KD
Q1	0	0	0
Q2	10	0.02	0
Q3	10	0.02	0

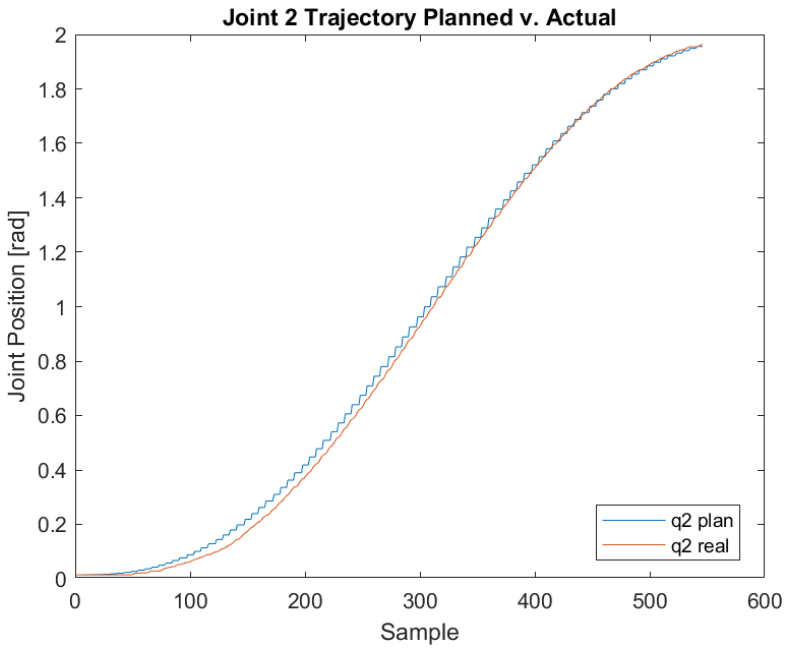

Figure 8.23. Position Set and feedback data from Joint 2.

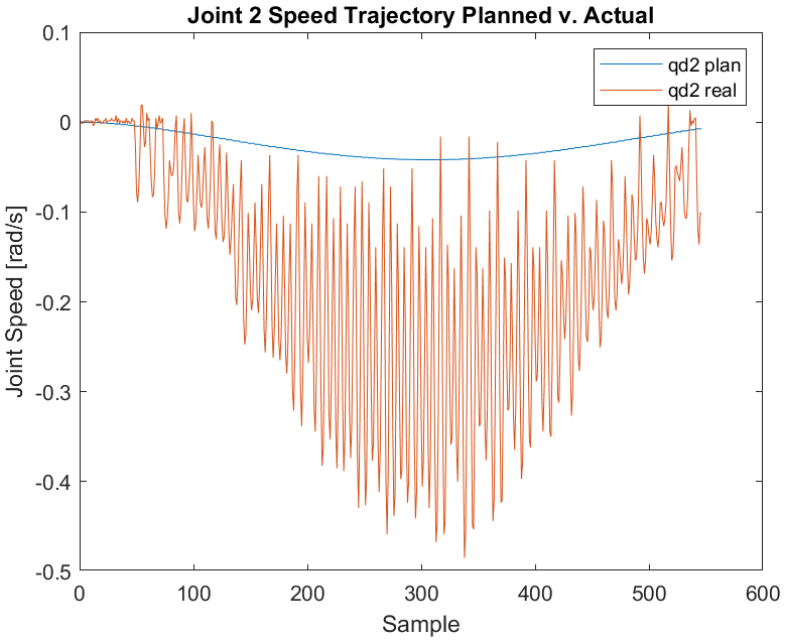

Figure 8.24. Speed Set and feedback data from Joint 2.

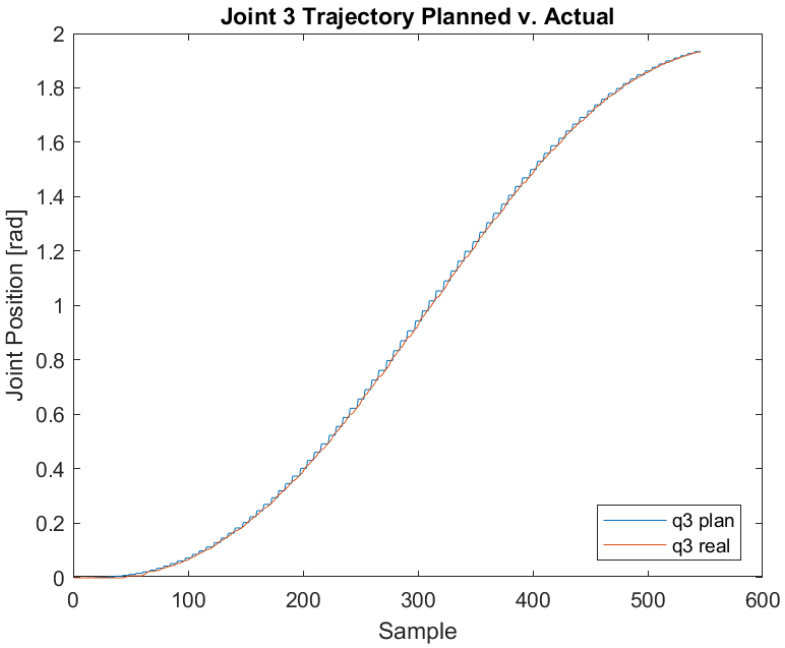

Figure 8.25. Position Set and feedback data from Joint 3.

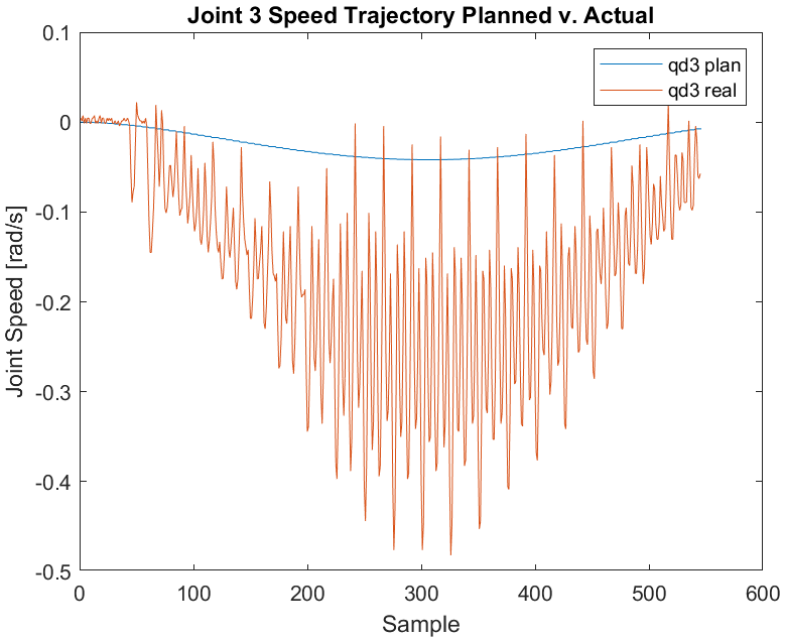

Figure 8.26. Speed Set and feedback data from Joint 3

Before coming to a conclusion it is appropriate to observe and discuss the data acquired from LDBs. Fig 8.27 shows the correlation between the inflation pressure and the capability of the link to sustain a load. Looking at the graph below, around sample 5000 can be noticed the moment on which a 1kg payload has been attached to the EE. With a pressure of 0.300 bar the link maintain its shape without deforming. Around Sample 8000 the pressure starts to be decreased. When the pressure reaches around 0.18 bar, the link collapsed with a deformation of almost 30 degree. Further decreasing the pressure leads to a complete deflation of the link up to almost 60 degree of bending. From around sample 18000 the pressure was increased again. The link inflates and the EE returned to its initial position.

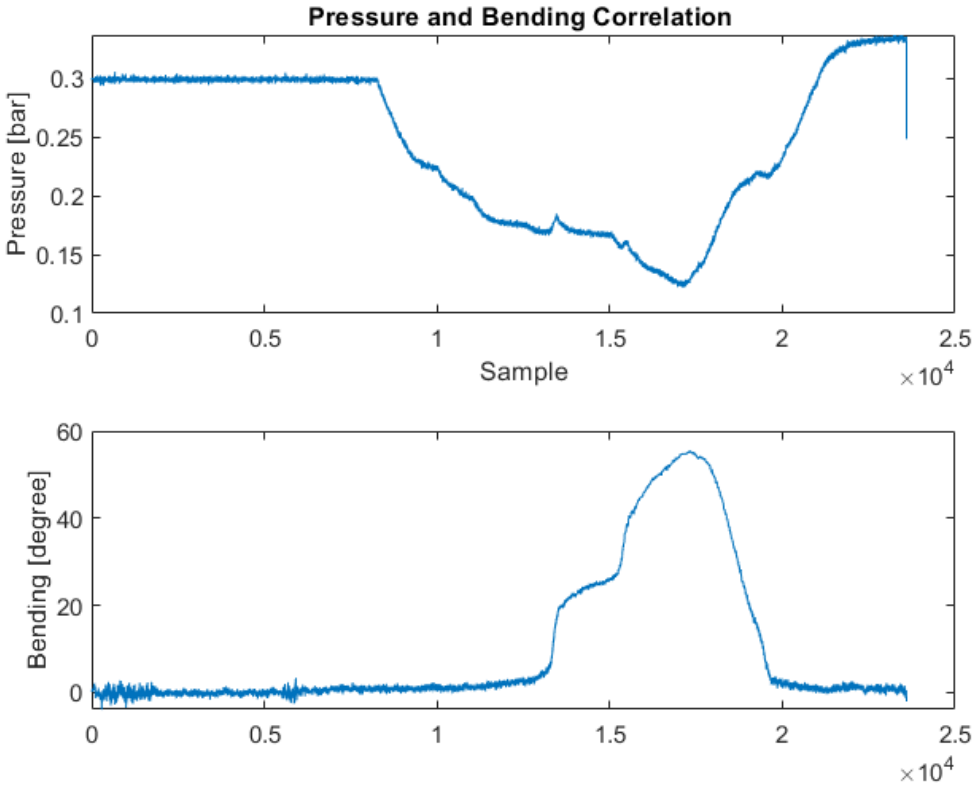

Figure 8.27. Correlation between bending angle and inflation pressure for 1kg payload applied.

Chapter 9

Conclusion

The aim of this thesis was to develop a system for the control and acquisition of link deformation data. The work was carried out as a first part of theoretical design using MATLAB, Simulink and Simscape to design and test the chosen control scheme. The possibility of correcting the link deformation in real time by acting on the joint variable related to the deformed link was also evaluated. The entire electronic system was then designed, starting from the choice of actuators and sensors to the hardware and software development of the two subsystems that make up the designed electronic system, the Main Controller Board (MCB) and the Link Data Board (LDB). Once the design phase was completed, we moved on to the implementation phase. The components were soldered onto the printed boards and debugging procedures were carried out. Finally, the developed system was used to run various trajectories in order to evaluate the impact of changing controller parameters. A decent repeatability could be appreciated but could not be quantified due to the lack of a measurement system. Finally, the analysis of the data coming from the LDBs was briefly shown, which proved to be absolutely interesting, in particular the correlation between the link inflation pressure and the link's ability to support a given payload. Possible future work includes the implementation of adaptive control systems, the transfer of the MCB firmware to FreeRTOS to exploit the multitasking capability or the use of pressure sensor data as a way to identify collisions.

Appendix A

Listing A.1. Inverse kinematic function MATLAB

```
function [Q] = POPUP_IK_funct(ee_target,mode)

Q = zeros(3,1);

% geometric variable
L1 = 0.745;
L2 = 0.685;
OFF = 0.07; %offset

%input
X = ee_target(1);
Y= ee_target(2);
Z = ee_target(3);
%mode
Mode = mode;

if Z < 0
    %display('Z cannot be lower than 0');
    return;
end
%joint 3
c3 = (X^2 + Y^2 + Z^2 - L1^2 - L2^2 - OFF^2)/(2*L1*L2);

%c3 must be comprise between 1 and -1, if not the target is out of
%workspace
if(c3 < 1 && c3 > -1)

    if(X > 0 && Y > 0 && Mode == 1)
        %solution 3
         s3_p = sqrt(1-c3^2);

         Q(3) = atan2(s3_p,c3);
```

```
33          Q(2) = atan2((L1 + L2*c3)*Z + L2*s3_p*sqrt(X^2 + Y^2 -OFF^2)
34          , -(L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_p*Z);
35
36          K = sqrt(X^2 + Y^2 - OFF^2);
37          c1 = (X/K + (Y*OFF)/K^2)/(1+(OFF^2)/K^2);
38          s1= sqrt(1-c1^2);
39
40
41          Q(1) = atan2(-s1,-c1);
42
43      end
44
45       if(X > 0 && Y > 0 && Mode == 0)
46         %solution 4
47          s3_m = -sqrt(1-c3^2);
48
49          Q(3) = atan2(s3_m,c3);
50
51          Q(2) = atan2((L1 + L2*c3)*Z + L2*s3_m*sqrt(X^2 + Y^2 - OFF^2) ,
52          -(L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_m*Z);
53
54          K = sqrt(X^2 + Y^2 - OFF^2);
55          c1 = (X/K + (Y*OFF)/K^2)/(1+(OFF^2)/K^2);
56          s1= sqrt(1-c1^2);
57
58          Q(1) = atan2(-s1,-c1);
59
60       end
61
62      if(X < 0 && Y > 0 && Mode == 1)
63                  %solution 3
64          s3_p = sqrt(1-c3^2);
65
66          Q(3) = atan2(s3_p,c3);
67
68          Q(2) = atan2((L1 + L2*c3)*Z + L2*s3_p*sqrt(X^2 + Y^2 -OFF^2) ,
69          -(L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_p*Z);
70
71          K = sqrt(X^2 + Y^2 - OFF^2);
72          c1 = (X/K + (Y*OFF)/K^2)/(1+(OFF^2)/K^2);
73          s1= sqrt(1-c1^2);
74
75          Q(1) = atan2(-s1,-c1);
76      end
77
```

```
78  if(X < 0 && Y > 0 && Mode == 0)
79
80      s3_m = -sqrt(1-c3^2);
81
82      Q(3) = atan2(s3_m,c3);
83
84      Q(2) = atan2((L1 + L2*c3)*Z + L2*s3_m*sqrt(X^2 + Y^2 - OFF^2) ,
85      -(L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_m*Z);
86
87      K = sqrt(X^2 + Y^2 - OFF^2);
88      c1 = (X/K + (Y*OFF)/K^2)/(1+(OFF^2)/K^2);
89      s1= sqrt(1-c1^2);
90
91      Q(1) = atan2(-s1,-c1);
92
93
94  end
95
96  if(X > 0 && Y < 0 && Mode == 0)
97       %solution 1
98       s3_p = sqrt(1-c3^2);
99
100      Q(3) = atan2(s3_p,c3);
101
102      Q(2) = atan2((L1 + L2*c3)*Z - L2*s3_p*sqrt(X^2 + Y^2 - OFF^2),
103      (L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_p*Z);
104
105     K_m = -sqrt(X^2 + Y^2 - OFF^2);
106
107     c1_m = (X/K_m + (Y*OFF)/K_m^2)/(1+(OFF^2)/K_m^2);
108     s1_m= sqrt(1-c1_m^2);
109
110     Q(1) = atan2(s1_m,c1_m) - pi
111
112
113 end
114
115      if(X > 0 && Y < 0 && Mode == 1)
116     %solution 2
117        s3_m = -sqrt(1-c3^2);
118       Q(3) = atan2(s3_m,c3);
119
120       Q(2) = atan2((L1 + L2*c3)*Z - L2*s3_m*sqrt(X^2 + Y^2 -OFF^2) ,
121       (L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_m*Z);
122
```

```
123             K = sqrt(X^2 + Y^2 - OFF^2);
124              K_m = -sqrt(X^2 + Y^2 - OFF^2);
125
126              c1 = (X/K + (Y*OFF)/K^2)/(1+(OFF^2)/K^2);
127              c1_m = (X/K_m + (Y*OFF)/K_m^2)/(1+(OFF^2)/K_m^2);
128
129              s1= sqrt(1-c1^2);
130              s1_m= sqrt(1-c1_m^2);
131
132              Q(1) = atan2(s1_m,c1_m) - pi;
133
134
135
136           end
137
138
139       if(X < 0 && Y < 0 && Mode == 0)
140           %solution 1
141           s3_p = sqrt(1-c3^2);
142
143           Q(3) = atan2(s3_p,c3);
144
145           Q(2) = atan2((L1 + L2*c3)*Z - L2*s3_p*sqrt(X^2 + Y^2 - OFF^2),
146           (L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_p*Z);
147
148   K_m = -sqrt(X^2 + Y^2 - OFF^2);
149
150   c1_m = (X/K_m + (Y*OFF)/K_m^2)/(1+(OFF^2)/K_m^2);
151
152   s1_m= sqrt(1-c1_m^2);
153
154   Q(1) = atan2(s1_m,c1_m) + pi;
155
156
157
158       end
159
160            if(X < 0 && Y < 0 && Mode == 1)
161           %solution 2
162           s3_m = -sqrt(1-c3^2);
163
164           Q(3) = atan2(s3_m,c3);
165
166           Q(2) = atan2((L1 + L2*c3)*Z - L2*s3_m*sqrt(X^2 + Y^2 -OFF^2) ,
167           (L1 + L2*c3)*sqrt(X^2 + Y^2 - OFF^2) + L2*s3_m*Z);
```

```
168
169 K_m = -sqrt(X^2 + Y^2 - OFF^2);
170
171 c1_m = (X/K_m + (Y*OFF)/K_m^2)/(1+(OFF^2)/K_m^2);
172
173 s1_m= sqrt(1-c1_m^2);
174
175 Q(1) = atan2(s1_m,c1_m) + pi;
176
177       end
178 else
179     %display('Target is out of WS');
180     return;
181 end
182
183 end
```

Appendix B

Listing B.1. Trajectory planner function MATLAB

```
function [Q_planned, QD_planned,QDD_planned, t]= TrajPlanner(q0,qf,tmax,Ts)
%Ts = 0.01;
t = [0:Ts:tmax];

qd0 = 0;
qdf = 0;
qdd0 = 0;
qddf = 0;

tf = tmax;

a0 = q0;
a1 = qd0;
a2 = 0.5*qdd0;
a3 = (1/(2*tf^3))*(20*(qf - q0) - (8*qdf + 12*qd0)*tf - (3*qddf - qdd0)*tf^2);
a4 = (1/(2*tf^4))*(30*(q0 - qf) + (14*qdf + 16*qd0)*tf +(3*qddf - 2*qdd0)*tf^2);
a5 = (1/(2*tf^5))*(12*(qf - q0) - 6*(qdf + qd0)*tf - (qddf - qdd0)*tf^2);

q_planned = transpose(a0 + a1*t + a2*t.^2 + a3*t.^3 + a4*t.^4 + a5*t.^5);

qd_planned = transpose(a1 + 2*a2*t + 3*a3*t.^2 + 4*a4*t.^3 + 5*a5*t.^4);

qdd_planned = transpose(2*a2 + 6*a3*t + 12*a4*t.^2 + 20*a5*t.^3);

end
```

Appendix C

Listing C.1. Main Controller Firmware

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file : main.c
  * @brief : Main program body
  ******************************************************************************
  * POPUP Robot Main Controller
  * @Author: Francesco Gambino
  * @Year: 2021
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "string.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <math.h>
#include <retarget.h> // Fprintf sulla seriale
#include <stdio.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

```

```
33 // KINEMATIC VARIABLES
34 #define L1 0.745
35 #define L2 0.685
36 #define OFF 0.07
37 #define L1square 0.555025
38 #define L2square 0.469225
39 #define OFFsquare 0.0049
40
41 //ACTUATOR LIMITS
42 #define P_MAX 12.5 // posizione
43 #define V_MAX 6 // velocita
44 #define T_MAX 48 // coppia
45 #define Kp_MAX 500 // non vengono utilizzati, sono posti a zero. Sono relativi al controllo del firmv
46 #define Kd_MAX 1000
47 // CAN ID
48 #define MOTOR1 1
49 #define MOTOR2 2
50 #define MOTOR3 3
51 #define MOTOR_FB 0 // Nel pacchetto di feedback, ID unico per tutti i motori, e presente una stri
52
53 #define LINK_BOARD_1 10
54 #define LINK_BOARD_2 11
55 #define LINK_BOARD_FB_MSG_ID 0x6 // ID definito in esadecimale
56 #define LINK_BOARD_STATUS_MSG_ID 0x14 // ID definito in esadecimale
57 #define LINK_BOARD_CALIBRATION_CHECK_MSG_ID 0x28 // ID definito in esadecimale
58 // Gli status, feedback, calibration hanno un unico ID poiche all'interno una stringa mi indica a quale
59
60
61
62
63 #define MOTOR_FB_DEBUG 0
64 #define LINK_FB_DEBUG 0
65 #define TRAJ_PLAN_DEBUG 0
66 #define POS_CONTROLLER_DEBUG 1
67 #define SPEED_CONTROLLER_DEBUG 0
68
69
70 //CONTROLLER DEFINEs
71 #define MAX_INTEGRAL_ERROR 1000
72 /* USER CODE END PD */
73
74 /* Private macro ------------------------------------------------------------
75 /* USER CODE BEGIN PM */
76
77 /* USER CODE END PM */
```

```
78
79  /* Private variables ---------------------------------------------------------
80  #if defined ( __ICCARM__ ) /*!< IAR Compiler */
81
82  #pragma location=0x30040000
83  ETH_DMADescTypeDef DMARxDscrTab[ETH_RX_DESC_CNT]; /* Ethernet Rx DMA Descriptors */
84  #pragma location=0x30040060
85  ETH_DMADescTypeDef DMATxDscrTab[ETH_TX_DESC_CNT]; /* Ethernet Tx DMA Descriptors */
86  #pragma location=0x30040200
87  uint8_t Rx_Buff[ETH_RX_DESC_CNT][ETH_MAX_PACKET_SIZE]; /* Ethernet Receive Buffers */
88
89  #elif defined ( __CC_ARM ) /* MDK ARM Compiler */
90
91  __attribute__((at(0x30040000))) ETH_DMADescTypeDef DMARxDscrTab[ETH_RX_DESC_CNT]; /* Etherne
92  __attribute__((at(0x30040060))) ETH_DMADescTypeDef DMATxDscrTab[ETH_TX_DESC_CNT]; /* Ethernet
93  __attribute__((at(0x30040200))) uint8_t Rx_Buff[ETH_RX_DESC_CNT][ETH_MAX_PACKET_SIZE]; /* Ether
94
95  #elif defined ( __GNUC__ ) /* GNU Compiler */
96
97  ETH_DMADescTypeDef DMARxDscrTab[ETH_RX_DESC_CNT] __attribute__((section(".RxDecripSection")))
98  ETH_DMADescTypeDef DMATxDscrTab[ETH_TX_DESC_CNT] __attribute__((section(".TxDecripSection")));
    /* Ethernet Tx DMA Descriptors */
99  uint8_t Rx_Buff[ETH_RX_DESC_CNT][ETH_MAX_PACKET_SIZE] __attribute__((section(".RxArraySection"))
100
101 #endif
102
103 ETH_TxPacketConfig TxConfig;
104
105 ETH_HandleTypeDef heth;
106
107 FDCAN_HandleTypeDef hfdcan1;
108
109 TIM_HandleTypeDef htim2;
110 TIM_HandleTypeDef htim3;
111 TIM_HandleTypeDef htim4;
112
113 UART_HandleTypeDef huart3;
114
115 PCD_HandleTypeDef hpcd_USB_OTG_FS;
116
117 /* USER CODE BEGIN PV */
118 // CAN PARAMETERS AND BUFFERS
119 FDCAN_TxHeaderTypeDef pTxHeader;
120 FDCAN_FilterTypeDef sFilterConfig;
121 FDCAN_RxHeaderTypeDef pRxHeader;
```

```
122
123 uint8_t CAN_tx_buffer[8];
124 uint8_t CAN_rx_buffer[8];
125
126
127 //TARGET VARIABLES
128 float Cartesian_target[3];
129 float Joint_target[3];
130 float Joint_speed_target[3];
131
132 // ACTUATORS FEEDBACK VARIABLES
133 float actualPos[3];
134 float actualSpeed[3];
135 float actualCurr[3];
136
137 // TRAJECTORY PLANNER BUFFER
138 float Joint_target_plan[2000][3]; //matrix of Ns rows and 3 column
139 float Joint_speed_target_plan[2000][3];
140 int plan_counter = 0;
141
142
143 // POSITION CONTROLLER VARIABLES
144 float Joint_target_planned[3]; //here the trajector planner will update the ref point for the position co
145 float pos_KP[3];
146 float pos_KI[3];
147 float pos_KD[3];
148 float pos_error[3];
149 float pos_integral[3];
150 float pos_derivative[3];
151 float pos_previous_error[3];
152 float pos_command[3];
153 volatile int flag_pos_controller = 0;
154
155 // SPEED CONTROLLER VARIABLES
156 float speed_KP[3];
157 float speed_KI[3];
158 float speed_KD[3];
159 float speed_error[3];
160 float speed_integral[3];
161 float speed_derivative[3];
162 float speed_previous_error[3];
163 float speed_command[3];
164 volatile int flag_speed_controller = 0;
165
166 // LINK BENDING FEEDBACK VARAIBLES
```

```
167 float actual_horiz_bend = 0;
168 float actual_vert_bend = 0;
169
170 // STATUS VARIABLES
171 int motor_status_flag[3];
172 int link_status_flag[2];
173 int link_cal_check[2];
174
175 // VESC CONTROL VARIABLE
176 float Joint_speed_target_planned[3]; //here the trajector planner will update the ref point for the position con
177
178
179
180 /* USER CODE END PV */
181
182 /* Private function prototypes -----------------------------------------------
183 void SystemClock_Config(void);
184 static void MX_GPIO_Init(void);
185 static void MX_ETH_Init(void);
186 static void MX_FDCAN1_Init(void);
187 static void MX_USART3_UART_Init(void);
188 static void MX_USB_OTG_FS_PCD_Init(void);
189 static void MX_TIM2_Init(void);
190 static void MX_TIM3_Init(void);
191 static void MX_TIM4_Init(void);
192 /* USER CODE BEGIN PFP */
193 int float_to_uint(float x, float x_min , float x_max, unsigned int bits);
194 float uint_to_float( int x_int , float x_min , float x_max , int bits);
195
196 void CAN_RxFilter_Config();
197 void CAN_TxHeader_Config();
198
199 void ActivateMotor(int id);
200 void SendTorque(int id, float u); // Esplorata con Francesco
201
202 void unpack_motor_FB();
203 void unpack_link_FB();
204 void unpack_link_STATUS();
205 void unpack_link_CAL_CHECK();
206
207 void InverseKinematic(float EE_target[3], int Mode);
208 void TrajectorPlanner(float q0[3], float qf[3], float t);
209 void BendingCorrection();
210
211 void PositionController();
```

```
212 void SpeedController();
213
214
215 //veryfy system connectivity (link + sensors)
216 void POPUP_system_check();
217 //verify link status (pressure reading)
218 void POPUP_link_status_check();
219 //activate motors
220 void POPUP_activate_motors();
221 //start controllers
222 void POPUP_start_controllers();
223
224 void POPUP_start_plan();
225
226 //homing
227 void POPUP_homing();
228 //send calibration command to link boards
229 void POPUP_calibrate_link_sensors();
230
231 void CAN_TX_vesc_speed(float speed);
232 /* USER CODE END PFP */
233
234 /* Private user code ---------------------------------------------------------
235 /* USER CODE BEGIN 0 */
236
237 /* USER CODE END 0 */
238
239 /**
240   * @brief The application entry point.
241   * @retval int
242   */
243 int main(void)
244 {
245   /* USER CODE BEGIN 1 */
246
247   /* USER CODE END 1 */
248
249   /* MCU Configuration--------------------------------------------------------
250
251   /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
252   HAL_Init(); //Inizializza le sue funzioni interne con cui richiama le periferiche a basso livello
253
254   /* USER CODE BEGIN Init */
255
256   /* USER CODE END Init */
```

```
257
258   /* Configure the system clock */
259   SystemClock_Config(); //COnfigura il clock come e stato impostato su CubeMXclolcconfigurator
260
261   /* USER CODE BEGIN SysInit */
262
263   /* USER CODE END SysInit */
264
265   /* Initialize all configured peripherals */
266   MX_GPIO_Init();
267   MX_ETH_Init();
268   MX_FDCAN1_Init();
269   MX_USART3_UART_Init();
270   MX_USB_OTG_FS_PCD_Init();
271   MX_TIM2_Init();
272   MX_TIM3_Init();
273   MX_TIM4_Init();
274   /* USER CODE BEGIN 2 */
275   RetargetInit(&huart3); // Mi serve per usare printf
276
277   printf("POPUP Robot Main Controller V0.1\n");
278
279
280   printf("CAN Register Configuring... \n");
281
282    CAN_RxFilter_Config(); //Voglio che una periferica non veda tutti i pacchetti passanti sul CAN
283    CAN_TxHeader_Config(); //Parametri iniziali che aggiorniamo
284
285    if ( (HAL_FDCAN_Start(& hfdcan1)) == HAL_OK) // l'ultimo argomento non ci interessa se usiamo la F
286              //FIFO = memoria di tipo first in first out
287              printf("CAN PHY started\n");
288    else {
289              printf("CAN PHY initialization error\n");
290              //while(1);
291
292    }
293    //HAL_FDCAN_ConfigInterruptLines(&hfdcan1, FDCAN_IT_RX_FIFO0_NEW_MESSAGE, FDCAN_INTI
294    if ( HAL_FDCAN_ActivateNotification(&hfdcan1, FDCAN_IT_RX_FIFO0_NEW_MESSAGE, 0) == HAL_
// l'ultimo argomento non ci interessa se usiamo la FIFO
295              printf("CAN Configuring: DONE\n\n\n"); //Attiva la comunicazione e permette al CAN di genera
296
297
298    HAL_Delay(2000);
299    ////////////////////////////////////////////////
300    //assign controllers parameter
```

```
301
302     //joint 1 position PID controller gain
303             pos_KP[0] = 5;
304             pos_KI[0] = 0;
305             pos_KD[0] = 0;
306
307             //joint 2 position PID controller gain
308             pos_KP[1] = 12;
309             pos_KI[1] = 0.05;
310             pos_KD[1] = 0;
311
312             //joint 3 position PID controller gain
313             pos_KP[2] = 8;
314             pos_KI[2] = 0.05;
315             pos_KD[2] = 0;
316
317             //joint 1 speed PID controller gain
318             speed_KP[0] = 10;
319             speed_KI[0] = 0;
320             speed_KD[0] = 0;
321
322             //joint 2 speed PID controller gain
323             speed_KP[1] = 5;
324             speed_KI[1] = 0;
325             speed_KD[1] = 0;
326
327             //joint 3 speed PID controller gain
328             speed_KP[2] = 10;
329             speed_KI[2] = 0;
330             speed_KD[2] = 0;
331
332             ////////////////////////////////////
333     //veryfy system status (link + sensors)
334             POPUP_system_check();
335     //activate motors
336             POPUP_activate_motors(); //Il motore 1 e abilitato grazie alla "Vesc"
337     //start controllers
338     //homing
339              //POPUP_
340        //POPUP_homing();
341     //send calibration command to link boards
342              //POPUP_calibrate_link_sensors(10); //Da sbloccare dopo aver sistemato il sistema fisico
343     //start loop
344
345             printf("\nSystem_Initialized\n");
```

```
346
347
348     Cartesian_target[0] = 0.7; //X
349     Cartesian_target[1] = 0; //Y
350     Cartesian_target[2] = 0.1; //Z
351
352     printf("Target = [ X: %f   Y: %f   Z: %f ]\n",Cartesian_target[0],Cartesian_target[1],Cartesian_target[2]);
353
354     InverseKinematic(Cartesian_target, 1); //Target[3] , Mode (elbow up,down)
355
356     printf("Joint Target = [ q1: %f   q2: %f   q3: %f ]\n",Joint_target[0],Joint_target[1],Joint_target[2]);
357     printf("Actual Joint position = [ q1: %f   q2: %f  q3: %f ]\n",actualPos[0],actualPos[1],actualPos[2]);
358     Joint_target[2] = -Joint_target[2]; // I motori sono invertiti rispetto i parametri D-H
359     printf("\nStarting trajectory planner...\n");
360
361     TrajectorPlanner(actualPos, Joint_target, 10);
362
363     printf("\nStarting trajectory planning completed.\n");
364
365
366     if(TRAJ_PLAN_DEBUG) {
367             //Dichiarata all'inizio
368             int i,j;
369             for(i = 0; i < 3; i++) {
370                     for(j = 0; j < 101; j++) {
371                             printf("T: %f q%d: %f  qd%d: %f\n",(j*0.1),i,Joint_target_plan[j][i],i,Joint_speed_t
372                             }
373                     printf("\n\n");
374                     }
375     }
376
377     //assign the first planned variable to the position controller input
378     for(int m = 0; m<3; m++) {
379             Joint_target_planned[m] = Joint_target_plan[0][m]; //Rimane fermo, il primo punto corrisponde al p
380     }
381
382
383     //start controllers
384     POPUP_start_controllers();
385
386     HAL_Delay(1000); // Da valutare di rimuoverlo
387
388     //start planned movement
389     POPUP_start_plan();
390
```

```
391
392     /* USER CODE END 2 */
393
394     /* Infinite loop */
395     /* USER CODE BEGIN WHILE */
396     while (1)
397     {
398             if(flag_pos_controller == 1)
399                     PositionController();
400
401             if(flag_speed_controller == 1) {
402                     SpeedController();
403                     CAN_TX_vesc_speed(-Joint_speed_target_planned[0]);
404             }
405
406               //printf("VESC target speed: %f\n",Joint_speed_target_planned[0]);
407             //APPUNTI MARIO-PIER
408             //I motori mandano pacchetti CAN solo quando invio loro un pacchetto
409             //Le board mandano pacchetti CAN quando le attivo
410
411       /* USER CODE END WHILE */
412
413       /* USER CODE BEGIN 3 */
414
415
416
417     }
418     /* USER CODE END 3 */
419 }
420
421 /**
422     * @brief System Clock Configuration
423     * @retval None
424     */
425 void SystemClock_Config(void)
426 {
427     RCC_OscInitTypeDef RCC_OscInitStruct = {0};
428     RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
429
430     /** Supply configuration update enable
431     */
432     HAL_PWREx_ConfigSupply(PWR_LDO_SUPPLY);
433     /** Configure the main internal regulator output voltage
434     */
435     __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);
```

```
436
437   while(!__HAL_PWR_GET_FLAG(PWR_FLAG_VOSRDY)) {}
438   /** Initializes the RCC Oscillators according to the specified parameters
439   * in the RCC_OscInitTypeDef structure.
440   */
441   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
442   RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
443   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
444   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
445   RCC_OscInitStruct.PLL.PLLM = 1;
446   RCC_OscInitStruct.PLL.PLLN = 60;
447   RCC_OscInitStruct.PLL.PLLP = 2;
448   RCC_OscInitStruct.PLL.PLLQ = 8;
449   RCC_OscInitStruct.PLL.PLLR = 2;
450   RCC_OscInitStruct.PLL.PLLRGE = RCC_PLL1VCIRANGE_3;
451   RCC_OscInitStruct.PLL.PLLVCOSEL = RCC_PLL1VCOWIDE;
452   RCC_OscInitStruct.PLL.PLLFRACN = 0;
453   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
454   {
455     Error_Handler();
456   }
457   /** Initializes the CPU, AHB and APB buses clocks
458   */
459   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
460                                 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2
461                                 |RCC_CLOCKTYPE_D3PCLK1|RCC_CLOCKTYPE_D1PCLK1;
462   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
463   RCC_ClkInitStruct.SYSCLKDivider = RCC_SYSCLK_DIV1;
464   RCC_ClkInitStruct.AHBCLKDivider = RCC_HCLK_DIV2;
465   RCC_ClkInitStruct.APB3CLKDivider = RCC_APB3_DIV1;
466   RCC_ClkInitStruct.APB1CLKDivider = RCC_APB1_DIV2;
467   RCC_ClkInitStruct.APB2CLKDivider = RCC_APB2_DIV1;
468   RCC_ClkInitStruct.APB4CLKDivider = RCC_APB4_DIV1;
469
470   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
471   {
472     Error_Handler();
473   }
474 }
475
476 /**
477   * @brief ETH Initialization Function
478   * @param None
479   * @retval None
480   */
```

```
481 static void MX_ETH_Init(void)
482 {
483
484   /* USER CODE BEGIN ETH_Init 0 */
485
486   /* USER CODE END ETH_Init 0 */
487
488    static uint8_t MACAddr[6];
489
490   /* USER CODE BEGIN ETH_Init 1 */
491
492   /* USER CODE END ETH_Init 1 */
493   heth.Instance = ETH;
494   MACAddr[0] = 0x00;
495   MACAddr[1] = 0x80;
496   MACAddr[2] = 0xE1;
497   MACAddr[3] = 0x00;
498   MACAddr[4] = 0x00;
499   MACAddr[5] = 0x00;
500   heth.Init.MACAddr = &MACAddr[0];
501   heth.Init.MediaInterface = HAL_ETH_RMII_MODE;
502   heth.Init.TxDesc = DMATxDscrTab;
503   heth.Init.RxDesc = DMARxDscrTab;
504   heth.Init.RxBuffLen = 1524;
505
506   /* USER CODE BEGIN MACADDRESS */
507
508   /* USER CODE END MACADDRESS */
509
510   if (HAL_ETH_Init(&heth) != HAL_OK)
511   {
512     Error_Handler();
513   }
514
515   memset(&TxConfig, 0 , sizeof(ETH_TxPacketConfig));
516   TxConfig.Attributes = ETH_TX_PACKETS_FEATURES_CSUM | ETH_TX_PACKETS_FEATURE
517   TxConfig.ChecksumCtrl = ETH_CHECKSUM_IPHDR_PAYLOAD_INSERT_PHDR_CALC;
518   TxConfig.CRCPadCtrl = ETH_CRC_PAD_INSERT;
519   /* USER CODE BEGIN ETH_Init 2 */
520
521   /* USER CODE END ETH_Init 2 */
522
523 }
524
525 /**
```

```
526   * @brief FDCAN1 Initialization Function
527   * @param None
528   * @retval None
529   */
530 static void MX_FDCAN1_Init(void)
531 {
532
533   /* USER CODE BEGIN FDCAN1_Init 0 */
534
535   /* USER CODE END FDCAN1_Init 0 */
536
537   /* USER CODE BEGIN FDCAN1_Init 1 */
538
539   /* USER CODE END FDCAN1_Init 1 */
540   hfdcan1.Instance = FDCAN1;
541   hfdcan1.Init.FrameFormat = FDCAN_FRAME_CLASSIC;
542   hfdcan1.Init.Mode = FDCAN_MODE_NORMAL;
543   hfdcan1.Init.AutoRetransmission = DISABLE;
544   hfdcan1.Init.TransmitPause = DISABLE;
545   hfdcan1.Init.ProtocolException = DISABLE;
546   hfdcan1.Init.NominalPrescaler = 4;
547   hfdcan1.Init.NominalSyncJumpWidth = 1;
548   hfdcan1.Init.NominalTimeSeg1 = 12;
549   hfdcan1.Init.NominalTimeSeg2 = 2;
550   hfdcan1.Init.DataPrescaler = 1;
551   hfdcan1.Init.DataSyncJumpWidth = 1;
552   hfdcan1.Init.DataTimeSeg1 = 1;
553   hfdcan1.Init.DataTimeSeg2 = 1;
554   hfdcan1.Init.MessageRAMOffset = 0;
555   hfdcan1.Init.StdFiltersNbr = 0;
556   hfdcan1.Init.ExtFiltersNbr = 0;
557   hfdcan1.Init.RxFifo0ElmtsNbr = 1;
558   hfdcan1.Init.RxFifo0ElmtSize = FDCAN_DATA_BYTES_8;
559   hfdcan1.Init.RxFifo1ElmtsNbr = 1;
560   hfdcan1.Init.RxFifo1ElmtSize = FDCAN_DATA_BYTES_8;
561   hfdcan1.Init.RxBuffersNbr = 1;
562   hfdcan1.Init.RxBufferSize = FDCAN_DATA_BYTES_8;
563   hfdcan1.Init.TxEventsNbr = 0;
564   hfdcan1.Init.TxBuffersNbr = 0;
565   hfdcan1.Init.TxFifoQueueElmtsNbr = 1;
566   hfdcan1.Init.TxFifoQueueMode = FDCAN_TX_FIFO_OPERATION;
567   hfdcan1.Init.TxElmtSize = FDCAN_DATA_BYTES_8;
568   if (HAL_FDCAN_Init(&hfdcan1) != HAL_OK)
569   {
570     Error_Handler();
```

```
571   }
572   /* USER CODE BEGIN FDCAN1_Init 2 */
573
574   /* USER CODE END FDCAN1_Init 2 */
575
576 }
577
578 /**
579   * @brief TIM2 Initialization Function
580   * @param None
581   * @retval None
582   */
583 static void MX_TIM2_Init(void)
584 {
585
586   /* USER CODE BEGIN TIM2_Init 0 */
587
588   /* USER CODE END TIM2_Init 0 */
589
590   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
591   TIM_MasterConfigTypeDef sMasterConfig = {0};
592
593   /* USER CODE BEGIN TIM2_Init 1 */
594
595   /* USER CODE END TIM2_Init 1 */
596   htim2.Instance = TIM2;
597   htim2.Init.Prescaler = 120;
598   htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
599   htim2.Init.Period = 1000;
600   htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
601   htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
602   if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
603   {
604     Error_Handler();
605   }
606   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
607   if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
608   {
609     Error_Handler();
610   }
611   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
612   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
613   if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
614   {
615     Error_Handler();
```

```
616   }
617   /* USER CODE BEGIN TIM2_Init 2 */
618
619   /* USER CODE END TIM2_Init 2 */
620
621 }
622
623 /**
624   * @brief TIM3 Initialization Function
625   * @param None
626   * @retval None
627   */
628 static void MX_TIM3_Init(void)
629 {
630
631   /* USER CODE BEGIN TIM3_Init 0 */
632
633   /* USER CODE END TIM3_Init 0 */
634
635   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
636   TIM_MasterConfigTypeDef sMasterConfig = {0};
637
638   /* USER CODE BEGIN TIM3_Init 1 */
639
640   /* USER CODE END TIM3_Init 1 */
641   htim3.Instance = TIM3;
642   htim3.Init.Prescaler = 120;
643   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
644   htim3.Init.Period = 10000;
645   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
646   htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
647   if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
648   {
649     Error_Handler();
650   }
651   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
652   if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
653   {
654     Error_Handler();
655   }
656   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
657   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
658   if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
659   {
660     Error_Handler();
```

```
661   }
662   /* USER CODE BEGIN TIM3_Init 2 */
663
664   /* USER CODE END TIM3_Init 2 */
665
666 }
667
668 /**
669   * @brief TIM4 Initialization Function
670   * @param None
671   * @retval None
672   */
673 static void MX_TIM4_Init(void)
674 {
675
676   /* USER CODE BEGIN TIM4_Init 0 */
677
678   /* USER CODE END TIM4_Init 0 */
679
680   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
681   TIM_MasterConfigTypeDef sMasterConfig = {0};
682
683   /* USER CODE BEGIN TIM4_Init 1 */
684
685   /* USER CODE END TIM4_Init 1 */
686   htim4.Instance = TIM4;
687   htim4.Init.Prescaler = 1200 - 1;
688   htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
689   htim4.Init.Period = 10000 - 1;
690   htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
691   htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
692   if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
693   {
694     Error_Handler();
695   }
696   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
697   if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
698   {
699     Error_Handler();
700   }
701   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
702   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
703   if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
704   {
705     Error_Handler();
```

```
706     }
707     /* USER CODE BEGIN TIM4_Init 2 */
708
709     /* USER CODE END TIM4_Init 2 */
710
711 }
712
713 /**
714   * @brief USART3 Initialization Function
715   * @param None
716   * @retval None
717   */
718 static void MX_USART3_UART_Init(void)
719 {
720
721   /* USER CODE BEGIN USART3_Init 0 */
722
723   /* USER CODE END USART3_Init 0 */
724
725   /* USER CODE BEGIN USART3_Init 1 */
726
727   /* USER CODE END USART3_Init 1 */
728   huart3.Instance = USART3;
729   huart3.Init.BaudRate = 115200;
730   huart3.Init.WordLength = UART_WORDLENGTH_8B;
731   huart3.Init.StopBits = UART_STOPBITS_1;
732   huart3.Init.Parity = UART_PARITY_NONE;
733   huart3.Init.Mode = UART_MODE_TX_RX;
734   huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
735   huart3.Init.OverSampling = UART_OVERSAMPLING_16;
736   huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
737   huart3.Init.ClockPrescaler = UART_PRESCALER_DIV1;
738   huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
739   if (HAL_UART_Init(&huart3) != HAL_OK)
740   {
741     Error_Handler();
742   }
743   if (HAL_UARTEx_SetTxFifoThreshold(&huart3, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
744   {
745     Error_Handler();
746   }
747   if (HAL_UARTEx_SetRxFifoThreshold(&huart3, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
748   {
749     Error_Handler();
750   }
```

```
751   if (HAL_UARTEx_DisableFifoMode(&huart3) != HAL_OK)
752   {
753     Error_Handler();
754   }
755   /* USER CODE BEGIN USART3_Init 2 */
756
757   /* USER CODE END USART3_Init 2 */
758
759 }
760
761 /**
762   * @brief USB_OTG_FS Initialization Function
763   * @param None
764   * @retval None
765   */
766 static void MX_USB_OTG_FS_PCD_Init(void)
767 {
768
769   /* USER CODE BEGIN USB_OTG_FS_Init 0 */
770
771   /* USER CODE END USB_OTG_FS_Init 0 */
772
773   /* USER CODE BEGIN USB_OTG_FS_Init 1 */
774
775   /* USER CODE END USB_OTG_FS_Init 1 */
776   hpcd_USB_OTG_FS.Instance = USB_OTG_FS;
777   hpcd_USB_OTG_FS.Init.dev_endpoints = 9;
778   hpcd_USB_OTG_FS.Init.speed = PCD_SPEED_FULL;
779   hpcd_USB_OTG_FS.Init.dma_enable = DISABLE;
780   hpcd_USB_OTG_FS.Init.phy_itface = PCD_PHY_EMBEDDED;
781   hpcd_USB_OTG_FS.Init.Sof_enable = ENABLE;
782   hpcd_USB_OTG_FS.Init.low_power_enable = DISABLE;
783   hpcd_USB_OTG_FS.Init.lpm_enable = DISABLE;
784   hpcd_USB_OTG_FS.Init.battery_charging_enable = ENABLE;
785   hpcd_USB_OTG_FS.Init.vbus_sensing_enable = ENABLE;
786   hpcd_USB_OTG_FS.Init.use_dedicated_ep1 = DISABLE;
787   if (HAL_PCD_Init(&hpcd_USB_OTG_FS) != HAL_OK)
788   {
789     Error_Handler();
790   }
791   /* USER CODE BEGIN USB_OTG_FS_Init 2 */
792
793   /* USER CODE END USB_OTG_FS_Init 2 */
794
795 }
```

```
796
797 /**
798   * @brief GPIO Initialization Function
799   * @param None
800   * @retval None
801   */
802 static void MX_GPIO_Init(void)
803 {
804   GPIO_InitTypeDef GPIO_InitStruct = {0};
805
806   /* GPIO Ports Clock Enable */
807   __HAL_RCC_GPIOC_CLK_ENABLE();
808   __HAL_RCC_GPIOH_CLK_ENABLE();
809   __HAL_RCC_GPIOA_CLK_ENABLE();
810   __HAL_RCC_GPIOB_CLK_ENABLE();
811   __HAL_RCC_GPIOD_CLK_ENABLE();
812   __HAL_RCC_GPIOG_CLK_ENABLE();
813   __HAL_RCC_GPIOE_CLK_ENABLE();
814
815   /*Configure GPIO pin Output Level */
816   HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin, GPIO_PIN_RESET);
817
818   /*Configure GPIO pin Output Level */
819   HAL_GPIO_WritePin(USB_OTG_FS_PWR_EN_GPIO_Port, USB_OTG_FS_PWR_EN_Pin, GPIO_PIN_RESI
820
821   /*Configure GPIO pin Output Level */
822   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
823
824   /*Configure GPIO pin : B1_Pin */
825   GPIO_InitStruct.Pin = B1_Pin;
826   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
827   GPIO_InitStruct.Pull = GPIO_NOPULL;
828   HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
829
830   /*Configure GPIO pins : LD1_Pin LD3_Pin */
831   GPIO_InitStruct.Pin = LD1_Pin|LD3_Pin;
832   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
833   GPIO_InitStruct.Pull = GPIO_NOPULL;
834   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
835   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
836
837   /*Configure GPIO pin : USB_OTG_FS_PWR_EN_Pin */
838   GPIO_InitStruct.Pin = USB_OTG_FS_PWR_EN_Pin;
839   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
840   GPIO_InitStruct.Pull = GPIO_NOPULL;
```

```
841   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
842   HAL_GPIO_Init(USB_OTG_FS_PWR_EN_GPIO_Port, &GPIO_InitStruct);
843
844   /*Configure GPIO pin : USB_OTG_FS_OVCR_Pin */
845   GPIO_InitStruct.Pin = USB_OTG_FS_OVCR_Pin;
846   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
847   GPIO_InitStruct.Pull = GPIO_NOPULL;
848   HAL_GPIO_Init(USB_OTG_FS_OVCR_GPIO_Port, &GPIO_InitStruct);
849
850   /*Configure GPIO pin : LD2_Pin */
851   GPIO_InitStruct.Pin = LD2_Pin;
852   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
853   GPIO_InitStruct.Pull = GPIO_NOPULL;
854   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
855   HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
856
857 }
858
859 /* USER CODE BEGIN 4 */
860 void CAN_RxFilter_Config(void)
861 {
862   sFilterConfig.FilterIndex = 0;
863   sFilterConfig.IdType = FDCAN_EXTENDED_ID;
864   sFilterConfig.FilterType = FDCAN_FILTER_MASK;
865   sFilterConfig.FilterConfig = FDCAN_FILTER_TO_RXFIFO0 ;
866   sFilterConfig.FilterID1 = 0x00000000;
867   sFilterConfig.FilterID2 = 0x00000000; // mask => allow id from 0x00000150 to 0x0000015F
868
869   HAL_FDCAN_ConfigFilter(&hfdcan1, &sFilterConfig);
870
871   //HAL_FDCAN_ConfigGlobalFilter(&hfdcan1, FDCAN_REJECT, FDCAN_REJECT, FDCAN_RE
872
873 }
874
875 void CAN_TxHeader_Config(void)
876 {
877  pTxHeader.Identifier = 0x00000140;
878  pTxHeader.IdType = FDCAN_EXTENDED_ID; // specifies extended id
879  pTxHeader.TxFrameType = FDCAN_DATA_FRAME ; // frame type
880  pTxHeader.DataLength = FDCAN_DLC_BYTES_4; // specifies frame length
881  pTxHeader.ErrorStateIndicator = FDCAN_ESI_ACTIVE;
882  pTxHeader.BitRateSwitch = FDCAN_BRS_OFF;
883  pTxHeader.FDFormat = FDCAN_CLASSIC_CAN;
884  pTxHeader.TxEventFifoControl = FDCAN_NO_TX_EVENTS;
885  pTxHeader.MessageMarker = 0;
```

```
886 }
887
888
889 void HAL_FDCAN_RxFifo0Callback(FDCAN_HandleTypeDef *hfdcan, uint32_t RxFifo0ITs)
890 {
891
892         if((RxFifo0ITs & FDCAN_IT_RX_FIFO0_NEW_MESSAGE) != RESET)
893         {
894                 if (HAL_FDCAN_GetRxMessage(&hfdcan1,FDCAN_RX_FIFO0, &pRxHeader, CAN_rx_buffe
895                         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin); // yellow led
896                 }
897
898         if((pRxHeader.Identifier == 0)) //feedback from motor (position, speed)
899                         {
900                 unpack_motor_FB();
901                         }
902
903         if((pRxHeader.Identifier == LINK_BOARD_FB_MSG_ID)) //feedback from link data board (bending
904                         {
905                 unpack_link_FB();
906                         }
907         if((pRxHeader.Identifier == LINK_BOARD_STATUS_MSG_ID)) //status message with id 20 (DEC)
908                         {
909                 unpack_link_STATUS();
910                         }
911         if((pRxHeader.Identifier == LINK_BOARD_CALIBRATION_CHECK_MSG_ID)) //status message w
912                         {
913                 unpack_link_CAL_CHECK();
914                 HAL_GPIO_TogglePin(GPIOB, LD1_Pin); // green led
915                         }
916         if((pRxHeader.Identifier == 2305)) //status message from vesc
917                         {
918                 HAL_GPIO_TogglePin(GPIOB, LD1_Pin); // green led
919                 unpack_vesc_FB();
920                         }
921         // for(int i = 0; i < pRxHeader.DataLength; i++)
922         // printf("CAN_BUFF_RX[%d]: %d\n",i,CAN_rx_buffer[i]);
923
924         }
925 }
926
927 void unpack_vesc_FB() {
928
929           uint32_t rpm = ((CAN_rx_buffer[0] << 24) +(CAN_rx_buffer[1] << 16) + (CAN_rx_buffer[2] << 8
930
```

```
931             float mot1_speed = (float)(rpm/1120.0); //1120 = 14(pole-pair motor)*80 (gearbox reduc
932
933             //printf("Speed : %f rad/s \n",mot1_speed*0.10472 );
934
935             //printf("rpm : %d \n",rpm);
936
937 }
938
939 void unpack_motor_FB() {
940   //if it is the first time execution for each motor i must set the status flag
941
942   int motor_id = CAN_rx_buffer[0];
943
944   int pos_int = CAN_rx_buffer[1] << 8 | CAN_rx_buffer[2];
945
946   int vel_int = CAN_rx_buffer[3] << 4 | CAN_rx_buffer[4] >> 4;
947
948   int current_int = (CAN_rx_buffer[4] & 0xF) << 8 | CAN_rx_buffer[5];
949
950
951   actualPos[motor_id -1] = uint_to_float(pos_int, -P_MAX, P_MAX, 16);
952
953   actualSpeed[motor_id -1] = uint_to_float(vel_int, -V_MAX, V_MAX, 12);
954
955   actualCurr[motor_id -1] = uint_to_float(current_int, -T_MAX , T_MAX, 12);
956
957
958   if(motor_id == MOTOR1 && motor_status_flag[0] == 0)
959             motor_status_flag[0] = 1;
960
961   if(motor_id == MOTOR2 && motor_status_flag[1] == 0)
962             motor_status_flag[1] = 1;
963
964   if(motor_id == MOTOR3 && motor_status_flag[2] == 0)
965             motor_status_flag[2] = 1;
966
967
968   if (MOTOR_FB_DEBUG) {
969     printf("CAN_ID:␣%2d␣-␣Pos[rad]:␣%8f␣-␣Vel[rad/s]:␣%10f␣-␣Torque[Nm]:␣%8f␣\n",motor_id,actu
970             }
971 }
972
973 void unpack_link_FB() {
974
975   int link_id = CAN_rx_buffer[0];
```

```
976
977   int bend_horiz_int = CAN_rx_buffer[1] << 8 | CAN_rx_buffer[2]; //16 bit
978
979   int bend_vert_int = CAN_rx_buffer[3] << 8 | CAN_rx_buffer[4];
980
981 actual_horiz_bend = uint_to_float(bend_horiz_int, -90, 90, 16);
982
983 actual_vert_bend = uint_to_float(bend_vert_int, -90, 90, 16);
984
985 link_status_flag[link_id - 10] = CAN_rx_buffer[5]; //
986
987   if (LINK_FB_DEBUG) {
988     printf("LINK_ID: %d - Bend_Horiz: %2.1f - Bend_Vert: %2.1f \n",link_id,actual_horiz_bend,actual_ver
989
990   }
991 }
992
993 void unpack_link_STATUS() {
994
995             int link_id = CAN_rx_buffer[0];
996
997             int imu_status = CAN_rx_buffer[1];
998
999             int adc_status = CAN_rx_buffer[2];
1000
1001            int pressure_int = CAN_rx_buffer[3] << 8 | CAN_rx_buffer[4]; // Converto il dato da 8 bit a 16 bit
1002
1003            float pressure = uint_to_float(pressure_int, 0, 2, 16); // Valore in 16bit - Valore min - Valore max
1004
1005          link_status_flag[link_id - 10] = imu_status*adc_status; //
1006
1007            if (LINK_FB_DEBUG) {
1008              printf("LINK_ID: %d - Link_Pressure: %1.2f - IMU_Status: %d - ADC_Status: %d - Syster
1009            }
1010
1011 }
1012
1013 void unpack_link_CAL_CHECK() {
1014
1015
1016            int link_id = CAN_rx_buffer[0];
1017
1018            int cal_check = CAN_rx_buffer[1];
1019
1020          link_cal_check[link_id - 10] = cal_check; //
```

```
1021
1022            if (LINK_FB_DEBUG) {
1023              printf("LINK_ID: %d - Calibration_Status: &d \n",link_id,link_cal_check[link_id - 10]);
1024            }
1025
1026 }
1027
1028 void CAN_TX_link_board_status_check(int id) {
1029                 pTxHeader.Identifier = 0x30;       //status check message id
1030                     pTxHeader.IdType = FDCAN_STANDARD_ID; // specifies extended id
1031                     pTxHeader.TxFrameType = FDCAN_DATA_FRAME ; // frame type
1032                     pTxHeader.DataLength = FDCAN_DLC_BYTES_2;
1033
1034                     CAN_tx_buffer[0] = id;
1035                     CAN_tx_buffer[1] = 1;
1036
1037                      if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader, CAN_tx_buffer)
1038                             //HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
1039                      }
1040
1041 }
1042
1043 void CAN_TX_link_board_calibration(int id) {
1044                 pTxHeader.Identifier = 0x32;       //status check message id
1045                     pTxHeader.IdType = FDCAN_STANDARD_ID; // specifies extended id
1046                     pTxHeader.TxFrameType = FDCAN_DATA_FRAME ; // frame type
1047                     pTxHeader.DataLength = FDCAN_DLC_BYTES_2;
1048
1049                     CAN_tx_buffer[0] = id;
1050                     CAN_tx_buffer[1] = 1;
1051
1052                      if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader, CAN_tx_buffer)
1053                             //HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
1054                      }
1055
1056 }
1057
1058 void CAN_TX_vesc_speed(float speed) {
1059         //vesc id is 1
1060
1061         float s = speed;
1062
1063          pTxHeader.IdType = FDCAN_EXTENDED_ID; // specifies extended id
1064
1065     // 000003 e il messaggio che la vesc
```

```
1066        // si aspetta per gli rpm, 01 e l' id della scheda
1067        // status check message id
1068
1069              pTxHeader.TxFrameType = FDCAN_DATA_FRAME ;// frame type
1070              pTxHeader.DataLength = FDCAN_DLC_BYTES_4;
1071
1072              //take speed argument (rad/s)
1073              //convert to ERPM
1074              int erpm =(int)((s*9.5463*21.0*80.0)); //(rad/s)*(RPM_con)*(pole-pair)*(reduction);
1075         // il primo bit fornisce il segno al numero, occorre quindi mandare alla vesc tutti i bit
1076              // Alla vesc diamo la velocita e non la posizione
1077
1078                        CAN_tx_buffer[0] = erpm >> 24; // Converto da 32 a 8 bit
1079                        CAN_tx_buffer[1] = erpm >> 16;
1080                        CAN_tx_buffer[2] = erpm >> 8;
1081                        CAN_tx_buffer[3] = erpm;
1082
1083
1084              if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader, CAN_tx_buffer) == HAL_OK) {
1085                        //HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
1086               }
1087   }
1088
1089   void ActivateMotor(int id) {
1090
1091            pTxHeader.Identifier = id;
1092            pTxHeader.IdType = FDCAN_STANDARD_ID; // specifies extended id
1093             pTxHeader.TxFrameType = FDCAN_DATA_FRAME ; // frame type
1094             pTxHeader.DataLength = FDCAN_DLC_BYTES_8;
1095
1096                                    CAN_tx_buffer[0] = 0XFF;
1097                                    CAN_tx_buffer[1] = 0XFF;
1098                                    CAN_tx_buffer[2] = 0XFF;
1099                                    CAN_tx_buffer[3] = 0XFF;
1100                                    CAN_tx_buffer[4] = 0XFF;
1101                                    CAN_tx_buffer[5] = 0XFF;
1102                                    CAN_tx_buffer[6] = 0XFF;
1103                                    CAN_tx_buffer[7] = 0XFC; //Vedere datasheet
1104
1105                                     if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader, CAN_tx
1106                                              //HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
1107                                             // printf("Motor %lu Activated \n",pTxHeader.Identifier);
1108                                     }
1109   }
1110
```

```
1111 void DeactivateMotor(int id) {
1112
1113         pTxHeader.Identifier = id;
1114         pTxHeader.IdType = FDCAN_STANDARD_ID; // specifies extended id
1115          pTxHeader.TxFrameType = FDCAN_DATA_FRAME ; // frame type
1116          pTxHeader.DataLength = FDCAN_DLC_BYTES_8;
1117
1118                                   CAN_tx_buffer[0] = 0XFF;
1119                                   CAN_tx_buffer[1] = 0XFF;
1120                                   CAN_tx_buffer[2] = 0XFF;
1121                                   CAN_tx_buffer[3] = 0XFF;
1122                                   CAN_tx_buffer[4] = 0XFF;
1123                                   CAN_tx_buffer[5] = 0XFF;
1124                                   CAN_tx_buffer[6] = 0XFF;
1125                                   CAN_tx_buffer[7] = 0XFD;
1126
1127                                     if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader,
1128                                             //HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
1129                                             //printf("Motor %lu Activated \n",pTxHeader.Identifier);
1130                                     }
1131 }
1132
1133
1134 void SendTorque(int id, float u) {
1135             float p_des = 0;
1136             float v_des = 0;
1137             float t_ff = u;
1138
1139             float KP = 0;
1140             float KD = 0;
1141
1142             int p_int = float_to_uint(p_des, -P_MAX, P_MAX, 16);
1143             int v_int = float_to_uint(v_des, -V_MAX, V_MAX, 12);
1144
1145             int kp_int = float_to_uint(KP, 0, Kp_MAX, 12);
1146             int kd_int = float_to_uint(KD, 0, Kd_MAX, 12);
1147
1148             int t_int = float_to_uint(t_ff, -T_MAX, T_MAX, 12);
1149
1150             pTxHeader.Identifier = id;
1151             pTxHeader.IdType = FDCAN_STANDARD_ID; // specifies extended id
1152             pTxHeader.TxFrameType = FDCAN_DATA_FRAME ; // frame type
1153             pTxHeader.DataLength = FDCAN_DLC_BYTES_8;
1154
1155             CAN_tx_buffer[0] = p_int >> 8; //pos 8H
```

```
1156            CAN_tx_buffer[1] = p_int & 0xFF; //pos 8L
1157
1158            CAN_tx_buffer[2] = v_int >> 4; // speed 8H
1159            CAN_tx_buffer[3] = ((v_int & 0xF) << 4) | (kp_int >> 8) ; //speed 4L KP 8H
1160
1161            CAN_tx_buffer[4] = kp_int & 0xFF; // KP 8L
1162
1163            CAN_tx_buffer[5] = kd_int >> 4; // kd 8H
1164
1165            CAN_tx_buffer[6] = ((kd_int & 0xF) << 4) | (t_int >> 8) ; // KP 4L Torque $H
1166
1167            CAN_tx_buffer[7] = t_int & 0xFF; // torque 8L
1168
1169
1170            if (HAL_FDCAN_AddMessageToTxFifoQ(&hfdcan1, &pTxHeader, CAN_tx_buffer) == HAL_OK) {
1171
1172                    }
1173            else
1174            printf("Error sending command to Motor %d\n",id);
1175 }
1176
1177 void InverseKinematic(float EE_target[3], int Mode)
1178 {
1179
1180   //LOCAL VARIABLE
1181   float X = EE_target[0];
1182   float Y = EE_target[1];
1183   float Z = EE_target[2];
1184
1185   float Xsquare = pow(X, 2);
1186   float Ysquare = pow(Y, 2);
1187   float Zsquare = pow(Z, 2);
1188
1189   float K = sqrt(Xsquare + Ysquare - OFFsquare);
1190   float Ksquare = pow(K, 2);
1191   float c3, s3_p, c1, s1,s3_m;
1192
1193   if (Z < 0)
1194   {
1195     printf("Z must be > 0\n");
1196         return;
1197   }
1198
1199   //compute cosine of joint 3
1200   c3 = (Xsquare + Ysquare + Zsquare - L1square - L2square - OFFsquare) / (2 * L1 * L2);
```

```
1201
1202  if (c3 < 1 && c3 > −1)
1203          { // if requested point is part of WS
1204
1205          if (X >= 0 && Y >= 0 && Mode == 1)
1206                  {
1207      s3_p = sqrt(1 − pow(c3, 2));
1208
1209      Joint_target[2] = atan2(s3_p, c3);; //joint 3
1210      Joint_target[1] = atan2((L1 + L2 * c3) * Z + L2 * s3_p * K, −(L1 + L2 * c3) * K + L2 * s3_p
1211
1212      c1 = (X / K + (Y * OFF) / Ksquare) / (1 + (OFFsquare) / Ksquare);
1213      s1 = sqrt(1 − pow(c1, 2));
1214      Joint_target[0] = atan2(−s1, −c1);
1215                  }
1216
1217          if (X >= 0 && Y >= 0 && Mode == 0)
1218                  {
1219      s3_p = sqrt(1 − pow(c3, 2));
1220
1221      Joint_target[2] = atan2(−s3_p, c3); //joint 3
1222      Joint_target[1] = atan2((L1 + L2 * c3) * Z + L2 * −s3_p * K, −(L1 + L2 * c3) * K + L2 * −s
1223
1224      c1 = (X / K + (Y * OFF) / Ksquare) / (1 + (OFFsquare) / Ksquare);
1225      s1 = sqrt(1 − pow(c1, 2));
1226      Joint_target[0] = atan2(−s1, −c1);
1227                  }
1228
1229          if (X <= 0 && Y >= 0 && Mode == 1)
1230                  {
1231
1232
1233      s3_p = sqrt(1 − pow(c3, 2));
1234
1235      Joint_target[2] = atan2(s3_p, c3); //joint 3
1236
1237      Joint_target[1] = atan2((L1 + L2 * c3) * Z + L2 * s3_p * K, −(L1 + L2 * c3) * K + L2 * s3_p
1238
1239      c1 = (X / K + (Y * OFF) / Ksquare) / (1 + (OFFsquare) / Ksquare);
1240      s1 = sqrt(1 − pow(c1, 2));
1241
1242      Joint_target[0] = atan2(−s1, −c1);
1243                  }
1244
1245          if(X <= 0 && Y >= 0 && Mode == 0) {
```

```
1246
1247        s3_p = sqrt(1 − pow(c3, 2));
1248
1249        Joint_target[2] = atan2(−s3_p, c3);
1250
1251        Joint_target[1] = atan2((L1 + L2 ∗ c3) ∗ Z + L2 ∗ −s3_p ∗ K, −(L1 + L2 ∗ c3) ∗ K + L2 ∗ −s3_p ∗ Z)
1252
1253        c1 = (X / K + (Y ∗ OFF) / Ksquare) / (1 + (OFFsquare) / Ksquare);
1254        s1 = sqrt(1 − pow(c1, 2));
1255
1256        Joint_target[0] = atan2(−s1, −c1);
1257      }
1258
1259            if(X >= 0 && Y <= 0 && Mode == 1)
1260            {
1261                    s3_m = −sqrt(1−(c3∗c3));
1262
1263            Joint_target[2]= atan2(s3_m,c3);
1264
1265            Joint_target[1] = atan2((L1 + L2∗c3)∗Z − L2∗s3_m∗K , (L1 + L2∗c3)∗K + L2∗s3_m∗Z);
1266
1267                    float K_m = −Ksquare;
1268
1269                   c1 = (X/K_m + (Y∗OFF)/(K_m∗K_m))/(1+(OFFsquare)/(K_m∗K_m));
1270
1271                    s1= sqrt(1−pow(c1,2));
1272
1273            Joint_target[0] = atan2(s1,c1) − 3.1415;
1274            }
1275    }
1276    else
1277    {
1278            printf("Requested_target_is_not_part_of_reachable_workspace\n");
1279
1280    }
1281 }
1282 // joint space trjector planener
1283 void TrajectorPlanner(float q0[3], float qf[3], float t)
1284 {
1285    // q0 initial pose
1286    // qf final pose
1287    // t trajector time (steps)
1288    float Ts = 0.1; //time division step
1289
1290    float t_step = 0; //used as counter
```

```
1291  int counter = 0;
1292
1293  float Ns = t/Ts; //Number of step;
1294
1295  if(Ns > 2000) //max 20s planning at 0.1 Ts
1296          {
1297          printf("Ns cannot be higher than 2000");
1298          return;
1299          }
1300
1301
1302  float qd0[3];
1303  float qdf[3];
1304  float qdd0[3];
1305  float qddf[3];
1306
1307  //define poly coefficent
1308
1309  float a0[3];
1310  float a1[3];
1311  float a2[3];
1312  float a3[3];
1313  float a4[3];
1314  float a5[3];
1315
1316  //from t i must create an array with n elements
1317
1318  // fifth order polynomial interpolation function
1319  for (int i = 0; i < 3; i++) // execute for all the joint
1320  {
1321    qd0[i] = 0;
1322    qdd0[i] = 0;
1323
1324    qdf[i] = 0;
1325    qddf[i] = 0;
1326
1327    a0[i] = q0[i];
1328    a1[i] = qd0[i];
1329    a2[i] = 0.5 * qdd0[i];
1330
1331    //execute FOR to compute time value of joint
1332    a3[i] = (1 / (2 * pow(t, 3))) * (20 * (qf[i] - q0[i]) - (8 * qdf[i] + 12 * qd0[i]) * t - (3 * qddf[i] -
1333
1334    a4[i] = (1 / (2 * pow(t, 4))) * (30 * (q0[i] - qf[i]) + (14 * qdf[i] + 16 * qd0[i]) * t + (3 * qddf[i] -
1335
```

```
1336      a5[i] = (1 / (2 * pow(t, 5))) * (12 * (qf[i] − q0[i]) − 6 * (qdf[i] + qd0[i]) * t − (qddf[i] − qdd0[i]) * (t * t
1337
1338      for(counter = 0; counter < Ns+1; counter++ ) {
1339      t_step = counter*Ts;
1340
1341      Joint_target_plan[counter][i] = a0[i] + a1[i] * t_step + a2[i] * t_step * t_step + a3[i] * pow(t_step, 3) + a4
1342
1343      Joint_speed_target_plan[counter][i] = a1[i] + 2 * a2[i] * t_step + 3 * a3[i] * t_step * t_step + 4 * a4[i] * po
1344      }
1345      //float qd_quintic = a1[i] + 2 * a2[i] * t + 3 * a3[i] * t * t + 4 * a4[i] * pow(t, 3) + 5 * a5[i] * pow(t, 4);
1346
1347      //float qdd_quintic = 2 * a2[i] + 6 * a3[i] * t + 12 * a4[i] * t * t + 20 * a5[i] * pow(t, 3);
1348
1349
1350
1351    }
1352  }
1353
1354  void plan_step() {
1355          //every time is called, increment
1356          plan_counter++;
1357          if(plan_counter < 101) {
1358            for(int m = 0; m<3; m++) {
1359                    Joint_target_planned[m] = Joint_target_plan[plan_counter][m];
1360                    //printf("%d −− %f\n",m,Joint_target_planned[m]);
1361                    Joint_speed_target_planned[0] = Joint_speed_target_plan[plan_counter][0];
1362                    }
1363          }
1364  }
1365
1366  void PositionController() {
1367  //executed at 100 Hz
1368          for(int i = 1; i< 3; i++) {
1369
1370            pos_previous_error[i] = pos_error[i];
1371
1372            pos_error[i] = Joint_target_planned[i] − actualPos[i];
1373
1374            pos_integral[i] = pos_integral[i] + pos_error[i];
1375
1376            pos_derivative[i] = pos_error[i] − pos_previous_error[i];
1377
1378            //anti windup
1379            if (pos_integral[i] > MAX_INTEGRAL_ERROR || pos_integral[i] < −MAX_INTEGRAL_ERROR)
1380
```

```
1381            pos_command[i] = pos_KP[i] * pos_error[i] + pos_KI[i] * pos_integral[i] + pos_KD[i] * pos_de
1382            //apply limit
1383            if (pos_command[i] < - V_MAX) pos_command[i] = -V_MAX;
1384
1385            if (pos_command[i] > V_MAX) pos_command[i] = V_MAX;
1386
1387            if(POS_CONTROLLER_DEBUG) {
1388                    //printf("MOT_ID: %d - Error: %f - Command: %f\n",i+1,pos_error[i],pos_comm
1389                    printf("POS\\ MOT_ID: %d - Target: %10f - Actual: %10f - Error: %10f - Co
1390
1391            }
1392
1393        }
1394
1395        flag_pos_controller = 0;
1396 }
1397
1398 void SpeedController() {
1399 //executed at 1000 Hz
1400        for(int i = 1; i< 3; i++) {
1401                Joint_speed_target[i] = pos_command[i];
1402
1403            speed_previous_error[i] = speed_error[i];
1404
1405            speed_error[i] = Joint_speed_target[i] - actualSpeed[i];
1406
1407            speed_integral[i] = speed_integral[i] + speed_error[i];
1408
1409            speed_derivative[i] = speed_error[i] - speed_previous_error[i];
1410
1411            //anti windup
1412            if (speed_integral[i] > MAX_INTEGRAL_ERROR || speed_integral[i] < -MAX_INTEGRAL
1413
1414            speed_command[i] = speed_KP[i] * speed_error[i] + speed_KI[i] * speed_integral[i] + speed_K
1415            //apply limit
1416            if (speed_command[i] < - T_MAX) speed_command[i] = -T_MAX;
1417
1418            if (speed_command[i] > T_MAX) speed_command[i] = T_MAX;
1419
1420            if(SPEED_CONTROLLER_DEBUG) {
1421                    //printf("SPEED\\ MOT_ID: %d - Target: %f - Actual: %f - Error: %f - Comma
1422
1423            }
1424
1425            SendTorque(i+1, speed_command[i]); //i+1 perche i motori sono 1,2,3
```

```
1426
1427            HAL_Delay(1);
1428        }
1429        flag_speed_controller = 0;
1430
1431 }
1432
1433 /*
1434  * veryfy system connectivity (motor,link,sensors)
1435  * 1 - send packet to motor and check response (3)
1436  * -- motor answer with its id in frame 0
1437  * 2 - send system_status request to link board (x2)
1438  * -- link board answer with its id in frame 0 and the status of sensors in other frame
1439  *
1440  *
1441  */
1442 void POPUP_system_check() {
1443         //start procedure
1444         //moto
1445         printf("System check started..\n");
1446
1447
1448         HAL_GPIO_WritePin(GPIOB,LD3_Pin, SET); // red led on
1449
1450         int i = 0;
1451         for(i = 1; i<3; i++) {
1452                 DeactivateMotor(i+1); //Mandiamo un pacchetto CAN per verificare la presenza dei motori
1453                 while(motor_status_flag[i] == 0) //Il cambiamento di stato e' stato generato grazie all'interr
1454                         {
1455                                 printf("Motor %d not found\n",i+1);
1456                                 HAL_Delay(1000);
1457                                 DeactivateMotor(i+1);
1458                         }
1459                 printf("Motor %d found\n",i+1);
1460         }
1461
1462
1463         //link board status check
1464         HAL_Delay(1000);
1465         // 2 link board should answer to this packet
1466         for(i = 0; i < 1; i++) {
1467         while(link_status_flag[i] == 0) {
1468                 printf("Link Board %d not found\n",i+1);
1469                 HAL_Delay(1000);
1470         }
```

```
1471            HAL_GPIO_WritePin(GPIOB, LD3_Pin, RESET);
1472
1473            if (link_status_flag[i] == 1) printf("Link %d Sensor OK\n",i+1);
1474        else if (link_status_flag[i] == 4) printf("Link Board %d IMUs error\n",i+1); //IMU status = 4 v
1475        else if (link_status_flag[i] == 2) printf("Link Board %d RBS_ADC error\n",i+1);
1476            }
1477            //sent status check command
1478            if(link_status_flag[0] == 1) {
1479            CAN_TX_link_board_status_check(10); //send status check to link board 1
1480            printf("sent status check to link board 1\n");
1481            }
1482
1483            printf("\nSystem check completed\n\n\n");
1484
1485    }
1486
1487    void POPUP_activate_motors() {
1488            int i = 0;
1489                    for(i = 0; i<3; i++) {
1490                    ActivateMotor(i+1);
1491                    printf("Motor %d activated\n",i+1);
1492                    HAL_Delay(100);
1493
1494            }
1495    }
1496
1497    void POPUP_start_controllers() {
1498            //start timers base
1499            HAL_TIM_Base_Start_IT(&htim2);
1500            HAL_TIM_Base_Start_IT(&htim3);
1501
1502    }
1503
1504    void POPUP_start_plan() {
1505            //start timers base
1506            HAL_TIM_Base_Start_IT(&htim4); // Interrupt che ha frequenza pari a quella dei punti della
1507
1508    }
1509
1510    void POPUP_homing() {
1511
1512    }
1513
1514    void POPUP_calibrate_link_sensors(int id) {
1515            HAL_Delay(2000);
```

```
1516            CAN_TX_link_board_calibration(id); //send status check to link board 1
1517            printf("sent_calibration_command_to_link_board_%d\n",(id - 9));
1518            //wait calibration check from link board
1519        while(link_cal_check[id - 10] == 0) {
1520            printf("Waiting_calibration_check_from_link_board\n");
1521            HAL_Delay(1000);
1522        }
1523        printf("Calibration_check_received_from_link_board_%d",(id - 10));
1524    }
1525
1526    void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
1527            // Check which version of the timer triggered this callback and toggle LED
1528            if (htim == &htim2) {
1529                    flag_speed_controller = 1;
1530            }
1531            if (htim == &htim3) {
1532                    flag_pos_controller = 1;
1533            }
1534
1535            if (htim == &htim4) {
1536                    plan_step();
1537            }
1538
1539    }
1540
1541    float uint_to_float( int x_int , float x_min , float x_max , int bits) {
1542
1543      float span = x_max - x_min;
1544      float offset = x_min;
1545
1546      return ((float)x_int) * span / ((float)((1 << bits) - 1)) + offset;
1547
1548    }
1549
1550    int float_to_uint(float x, float x_min , float x_max, unsigned int bits) {
1551
1552      float span = x_max - x_min;
1553      if (x < x_min) x = x_min;
1554      else if (x > x_max ) x = x_max;
1555
1556      return (int)((x - x_min) * ((float)((1 << bits) - 1) / span));
1557
1558    }
1559    /* USER CODE END 4 */
1560
```

```
1561 /**
1562   * @brief This function is executed in case of error occurrence.
1563   * @retval None
1564   */
1565 void Error_Handler(void)
1566 {
1567   /* USER CODE BEGIN Error_Handler_Debug */
1568   /* User can add his own implementation to report the HAL error return state */
1569   __disable_irq();
1570   while (1)
1571   {
1572   }
1573   /* USER CODE END Error_Handler_Debug */
1574 }
1575
1576 #ifdef USE_FULL_ASSERT
1577 /**
1578   * @brief Reports the name of the source file and the source line number
1579   * where the assert_param error has occurred.
1580   * @param file: pointer to the source file name
1581   * @param line: assert_param error line source number
1582   * @retval None
1583   */
1584 void assert_failed(uint8_t *file, uint32_t line)
1585 {
1586   /* USER CODE BEGIN 6 */
1587   /* User can add his own implementation to report the file name and line number,
1588      ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
1589   /* USER CODE END 6 */
1590 }
1591 #endif /* USE_FULL_ASSERT */
1592
1593 /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Appendix D

Listing D.1. Link Data Board Fimware

```
1  /// LIBRARIES
2
3  #include "MYCAN.h"
4  #include "MPU9250.h"
5  #include <Wire.h> //i2c for imu1 and ads1115s
6  #include <Adafruit_ADS1115-Wire3.h>
7
8  // DEFINE
9  #define NSAMPLE 1 //ADC AVERAGING SAMPLE
10
11 #define CAN_LINK_FB_MESSAGE_ID 0x6 // ID MESSAGE FB TO MAIN CONTROLLER
12 #define CAN_LINK_ID 11 //LINK 1 ID 10 - LINK 2 ID 11
13 #define CAN_LINK_STATUS_MESSAGE_ID 0x14
14 #define CAN_LINK_STATUS_CHECK_MESSAGE_ID 0x30
15 #define CAN_LINK_CALIBRATION_COMPLETED_MESSAGE_ID 0x28
16 #define CAN_LINK_CALIBRATION_REQUEST_MESSAGE_ID 0x32
17
18 #define BENDING_ESTIMATOR_DEBUG 1 //print on serial bending information
19 #define IMUs_DEBUG 1 //print IMU readings on serial
20
21 /// INSTANCES
22 TwoWire Wire2(PB3, PB10); // I2C2 pins on stm32f4 for imu2 (Wire3 is defined in Adafruit_ADS1115-Wire
23
24 Adafruit_ADS1115 ads(0x48); // I2C ADDRESS OF ADC1 FOR RBS MEASURING (HORIZONTAL)
25 Adafruit_ADS1115 ads2(0x49); //I2C ADDRESS OF ADC2 FOR RBS MEASURING (VERTICAL)
26
27 MPU9250Setting setting; //DEFINITON OF SETTING OF IMUs (VEDI FILE MPU9250.h IN LIBRARIES
28 MPU9250 mpu1; //IMU1 DEFINITON
29 MPU9250 mpu2; //IMU2 DEFINITION
30
31 CAN_msg_t CAN_RX_msg; //GLOBAL CAN RECEIVE BUFFER
32
```

```
33 /// RBS VARIABLES
34 int counter = 0;
35 //
36 float multiplier = 0.0078125F; /* ADS1115 Voltage Resolution @ +/- 256mV (16-bit results) //
   256/2^16*/
37
38 float sum = 0;
39 float offset = 0;
40 float v_offset = 0;
41
42 float rbs_bending_angle_horiz_1 = 0;
43 float rbs_bending_angle_horiz_2 = 0;
44 float rbs_bending_angle_vert_1 = 0;
45 float rbs_bending_angle_vert_2 = 0;
46
47 /// IMUs VARIABLES
48 float imu_bending_angle_horiz = 0; //store the difference between imu horizontal orientation
49 float imu_bending_angle_vert = 0; //store the difference between imu vertical orientation
50
51 /// BENDING ESTIMATOR VARIABLES
52 float bending_estimate_horiz = 0; //store the estimated bending from the fusion on IMUs and RBS m
53 float bending_estimate_vert = 0; //store the estimated bending from the fusion on IMUs and RBS me
54 /// PRESSURE SENSOR VARIABLES
55 float pressure = 0;
56
57
58 // STATUS VARIABLES
59 uint8_t imu_Status = 0;
60 uint8_t adc_Status = 0;
61 int status_check = 0; // used to wait status verification from MCB
62 int calibration_flag = 0; // used to wait calibration command from MCB
63
64
65 void setup() {
66   delay(2000);
67
68   Serial.setRx(PA3); // Redirect Serial. to Nucleo STM32F446 UART2 RX
69   Serial.setTx(PA2);// Redirect Serial. to Nucleo STM32F446 UART2 TX
70   Serial.begin(115200); // define baud rate
71   Serial.println("Link_Data_Board_V0.1_-_Francesco_Gambino_2021");
72
73   // initializate I2C peripheral on STM32 (I2C1,I2C2,I2C3)
74
75   Wire.setSDA(PB9); // Redirect Wire to NUCLEO STM32F446 I2C1 SDA
76   Wire.setSCL(PB8); // Redirect Wire to NUCLEO STM32F446 I2C1 SCL
```

```
77  Wire.begin(); //imu1 i2c
78  Wire2.begin(); //imu2 i2c
79
80  // CAN INITIALIZATION
81  bool ret = CANInit(CAN_1000KBPS, 0); // CAN_RX mapped to PA11, CAN_TX mapped to PA12 defined
82  if (!ret) while (true); // if CAN TRANSCEIVERS IS NOT FOUND, BLOCK HERE
83
84  // TIMERS INITIALIZATION
85  TIM_TypeDef *Instance_CAN_RX = TIM3; //CAN_RX_IT used to check if a message arrived
86  TIM_TypeDef *Instance_CAN_TX = TIM5; //CAN TX Interrupt used to sent bending data to mcb
87  TIM_TypeDef *Instance_BEND = TIM6; //Interrupt used to execute bending estimation function
88
89  HardwareTimer *CAN_RX_Tim = new HardwareTimer(Instance_CAN_RX);
90  HardwareTimer *CAN_TX_Tim = new HardwareTimer(Instance_CAN_TX);
91  HardwareTimer *BEND_Tim = new HardwareTimer(Instance_BEND);
92
93  CAN_RX_Tim->setOverflow(100, HERTZ_FORMAT); // 100 Hz
94  CAN_RX_Tim->attachInterrupt(CAN_RX_IT_callback);
95  CAN_RX_Tim->resume();
96
97  init_IMUs(); //verify IMUs status
98
99  init_ADCs(); //verify ADCs status
100
101 getLinkPressure(); //verify link pressure
102
103 //wait status check from main controller //CAN RX INTERRUPT
104 while (status_check == 0) { //mentre aspetto lo status check dal mcb
105   CAN_TX_status_message(); //continuo a mandare ogni secondo il pacchetto di stato
106   Serial.println("Status_message_sent_to_main_controller._Waiting_status_check_from_MCB...");
107   delay(1000);
108 }
109
110 Serial.println("\nStatus_Check_Received_from_MCB\n");
111
112 while (calibration_flag == 0) { //una volta che lo stato del link e' verificato il mcb manda prima lo status
113   Serial.println("Waiting_calibration_command_from_MCB..."); //intanto giro e stampo sulla seriale
114   delay(100);
115 }
116
117 Serial.println("\nCalibration_command_Received_from_Main_Controller._Starting...\n");
118
119 calibrate_imus();
120 RBS_Offset_Comp();
121
```

```
122   CAN_TX_calibration_check_message();
123   Serial.println("Calibration_Check_sent_to_Main_Controller\n");
124
125   delay(1000);
126
127   Serial.println("Starting_sensor_acqusition,_bending_estimation_e_data_trasmission\n");
128
129   CAN_TX_Tim->setOverflow(100, HERTZ_FORMAT); // 10 Hz
130   CAN_TX_Tim->attachInterrupt(CAN_TX_Bending_Data);
131   CAN_TX_Tim->resume();
132
133   BEND_Tim->setOverflow(1000, HERTZ_FORMAT); // 10 Hz
134   BEND_Tim->attachInterrupt(Bending_sensor_fusion);
135   BEND_Tim->resume();
136
137
138 }
139
140 void loop() {
141
142 }
143
144 ////////// CAN FUNCTION //////////
145
146 void CAN_RX_IT_callback(void)
147 {
148   if (CANMsgAvail(1)) { //put it into an interrput
149     CANReceive(1, &CAN_RX_msg);
150     //Serial.print("RX from 0x");
151     //Serial.println(CAN_RX_msg.id);
152
153     CAN_RX_status_check();
154     CAN_RX_calibration_command();
155   }
156
157 }
158
159 void CAN_TX_Bending_Data() {
160
161   CAN_msg_t CAN_TX_msg;
162   CAN_TX_msg.id = CAN_LINK_FB_MESSAGE_ID;
163   CAN_TX_msg.type = DATA_FRAME;
164   CAN_TX_msg.format = STANDARD_FORMAT;
165   CAN_TX_msg.len = 5;
166
```

```
167   int bend_horiz_int = float_to_uint(bending_estimate_horiz, −90, 90, 16);
168   int bend_vert_int = float_to_uint(bending_estimate_vert, −90, 90, 16);
169
170   CAN_TX_msg.data[0] = CAN_LINK_ID;
171   CAN_TX_msg.data[1] = bend_horiz_int >> 8;
172   CAN_TX_msg.data[2] = bend_horiz_int & 0xFF;
173   CAN_TX_msg.data[3] = bend_vert_int >> 8;;
174   CAN_TX_msg.data[4] = bend_vert_int & 0xFF;
175
176   //Serial.print("CAN Data 0:");
177   //Serial.println(CAN_TX_msg.data[0]);
178   //Serial.print("CAN Data 1:");
179   //Serial.println(CAN_TX_msg.data[1]);
180   //Serial.print("CAN Data 2:");
181   //Serial.println(CAN_TX_msg.data[2]);
182   //Serial.print("CAN Data 3:");
183   //Serial.println(CAN_TX_msg.data[3]);
184   //Serial.print("CAN Data 4:");
185   //Serial.println(CAN_TX_msg.data[4]);
186
187   CANSend(1, &CAN_TX_msg);
188
189 }
190
191 void CAN_TX_status_message() {
192   CAN_msg_t CAN_TX_msg;
193   CAN_TX_msg.id = CAN_LINK_STATUS_MESSAGE_ID;
194   CAN_TX_msg.type = DATA_FRAME;
195   CAN_TX_msg.format = STANDARD_FORMAT;
196   CAN_TX_msg.len = 5;
197
198   int pressure_int = float_to_uint(pressure, 0, 2, 16);
199
200   CAN_TX_msg.data[0] = CAN_LINK_ID;
201   CAN_TX_msg.data[1] = imu_Status;
202   CAN_TX_msg.data[2] = adc_Status;
203   CAN_TX_msg.data[3] = pressure_int >> 8;
204   CAN_TX_msg.data[4] = pressure_int & 0xFF;
205
206   CANSend(1, &CAN_TX_msg);
207
208   // Serial.println(imu_Status);
209   // Serial.println(adc_Status);
210
211 }
```

```
212
213 void CAN_TX_calibration_check_message() {
214   CAN_msg_t CAN_TX_msg;
215   CAN_TX_msg.id = CAN_LINK_CALIBRATION_COMPLETED_MESSAGE_ID; // 0x28
216   CAN_TX_msg.type = DATA_FRAME;
217   CAN_TX_msg.format = STANDARD_FORMAT;
218   CAN_TX_msg.len = 2;
219
220   CAN_TX_msg.data[0] = CAN_LINK_ID;
221   CAN_TX_msg.data[1] = 1;
222
223   CANSend(1, &CAN_TX_msg);
224
225   // Serial.println(imu_Status);
226   // Serial.println(adc_Status);
227
228 }
229
230 void CAN_RX_status_check() {
231   if (CAN_RX_msg.id == CAN_LINK_STATUS_CHECK_MESSAGE_ID) { //0x30
232     status_check = 1;
233     Serial.println("Status_check_received");
234   }
235 }
236
237 void CAN_RX_calibration_command() {
238   if (CAN_RX_msg.id == CAN_LINK_CALIBRATION_REQUEST_MESSAGE_ID) { //0x32
239     calibration_flag = 1;
240   }
241 }
242
243
244 ////////// IMU FUNCTION //////////
245 void init_IMUs() {
246   int imu1_Status = 0;
247   int imu2_Status = 0;
248
249   if (!mpu1.setup(0x69)) { // AD0 HIGH
250     Serial.println("MPU1_connection_failed.");
251     imu1_Status = 4;
252   }
253   else {
254     Serial.println("IMU1_found");
255     imu1_Status = 1;
256   }
```

```
257
258 //per utilizzare wire2 occorre dichiarere anche il secondo argomento(setting)
259   if (!mpu2.setup(0x68, setting, Wire2)) { // AD0 LOW (
260     imu2_Status = 5;
261     Serial.println("MPU2_connection_failed.");
262   }
263   else {
264     Serial.println("IMU2_found");
265     imu2_Status = 1;
266   }
267
268 //imu_status = 1 : OK
269 // = 4 : imu 1 not found
270 // = 5 : imu 2 not found
271 // = 20 : imu 1 & 2 not found
272
273   imu_Status = imu1_Status*imu2_Status;
274
275 }
276
277 void BendingAngleIMUs() {
278   mpu1.update();
279   mpu2.update();
280
281   float Yaw_Bend = mpu1.getYaw() - mpu2.getYaw();
282   float Pitch_Bend = mpu1.getPitch() + mpu2.getPitch(); //li sommo perche sono orientate nel verso oppost
283   float Roll_Bend = mpu1.getRoll() - mpu2.getRoll();
284
285   if(IMUs_DEBUG) {
286 // Serial.print(Yaw_Bend);
287 // Serial.print(",");
288     Serial.println(Pitch_Bend);
289 // Serial.print(",");
290 // Serial.println(Roll_Bend);
291
292   }
293   imu_bending_angle_horiz = Pitch_Bend;
294   imu_bending_angle_vert = Yaw_Bend;
295
296 }
297
298 void calibrate_imus() {
299   // calibrate anytime you want to
300   Serial.println("Starting_Accel/Gyro_calibration...");
301   mpu1.calibrateAccelGyro();
```

```
302   Serial.println("IMU 1 Calibrated");
303   mpu2.calibrateAccelGyro();
304   Serial.println("IMU 2 Calibrated");
305
306   init_filter();
307 }
308
309 void init_filter() {
310   Serial.println("Waiting Madgwick Filter Convergence...");
311
312   mpu1.update(); //funzione di aggiornamento dell'algoritmo di stima dell'orientamento
313   mpu2.update();
314   while (mpu2.getPitch() > 1.0 || mpu2.getPitch() < -1.0 || mpu1.getPitch() > 1.0 || mpu1.getPitch
315     mpu1.update();
316     mpu2.update();
317     delay(1);
318   }
319   Serial.println("Madgwick  Filter Initialized");
320 }
321 ////////// RBS FUNCTION //////////
322 void init_ADCs() {
323   adc_Status = ads.check_ads_status(0x48) + 1; // +1 perche la funzione ritorna 0 se trova l'adc ma i
324   ads.setGain(GAIN_SIXTEEN); // 16x gain +/- 0.256V 1 bit = 0.125mV 0.0078125mV (in ADS_11
325   ads.begin(); //inizializza il convertitore adc
326 }
327
328 void BendingAngleRBS() {
329
330   float CALIBRATION_FACTOR = 1.05; //tuning parameter (not used)
331
332   for (int i = 0; i < NSAMPLE; i++) {
333     float volt = (ads.readADC_Differential_0_1() * multiplier);
334     sum = sum + volt;
335   }
336
337   float avg = sum / NSAMPLE; // - offset;
338
339   sum = 0;
340
341   //characteristic 4th degree function
342   //float z = (avg - 10.29)/7.675;
343   //float angl e= (-0.2365*pow(z,4) + 0.3031*pow(z,3) + 0.2339*pow(z,2) + 3.003*z + 5.117)- offse
344
345   if (avg < 3.0) //se la tensione letta e' minore di 3 mV applico la caratteristica non lineare
346     rbs_bending_angle_horiz_1 = (-0.0002016 * pow(avg, 4) + 0.008702 * pow(avg, 3) + 0.1301 * pow
```

```
347     else //per grandi angoli
348       rbs_bending_angle_horiz_1 = (0.4 * avg + 1.889) - offset;
349
350     //Serial.print(angle*CALIBRATION_FACTOR); Serial.println(" deg");
351     //Serial.println(angle);
352
353   }
354
355   void RBS_Offset_Comp() {
356     Serial.println("Starting RBS offset compensation...");
357
358     for (int i = 0; i < 100; i++) {
359       float volt = (ads.readADC_Differential_0_1() * multiplier); // 0-16bit integer * resolution.
360       sum = sum + volt;
361     }
362
363     float avg = sum / 100;
364
365     //potrei calcolarmi una varianza
366     //float u = avg/100*sqrt(3);
367
368     v_offset = avg;
369
370     //apply characteristic function (fit in matlab from experimental measurements)
371     offset = -0.0002016 * pow(avg, 4) + 0.008702 * pow(avg, 3) - 0.1301 * pow(avg, 2) + 1.228 * avg + 0.350
372
373     Serial.print("Computed Angular Offset: ");
374     Serial.print(offset);
375
376     Serial.print("Computed Voltage Offset: ");
377     Serial.print(v_offset);
378     Serial.println("mV");
379     sum = 0;
380
381   }
382
383   ////////// BENDING FUNCTION //////////
384   void Bending_sensor_fusion() {
385
386     BendingAngleIMUs();
387     BendingAngleRBS();
388
389     //simple average
390     bending_estimate_horiz = (imu_bending_angle_horiz);//+ rbs_bending_angle_horiz_1) / 2.0F;
391     bending_estimate_vert = (imu_bending_angle_vert);
```

```
392     //complementary filter
393
394
395 }
396
397 ////////// MISC //////////
398 void getLinkPressure() {
399     //read analog
400     //apply characteristic function
401     float v_pressure = analogRead(A5)*(3.3F/4096.0F);
402
403     pressure = (v_pressure - 0.25)/4.5; //Bar
404
405 }
406
407 int float_to_uint(float x, float x_min , float x_max, unsigned int bits) {
408
409     float span = x_max - x_min;
410     if (x < x_min) x = x_min;
411     else if (x > x_max ) x = x_max;
412
413     return (int)((x - x_min) * ((float)((1 << bits) - 1) / span));
414
415 }
```

Bibliography

[1] B.Siciliano. *Robotics Modelling, Planning and Control*. Springer, 2009.

[2] M. Y. Wang. C. Feifei. "Design Optimization of Soft Robots: A Review of the State of the Art."". In: *IEEE Robotics Automation Magazine, 2020, vol. 27(4), pp. 27-43.* ().

[3] P. Corke. *Robotics, vision and control fundamental algorithms in MATLAB*. Springer, 2011.

[4] Steve Corrigan... *"Introduction to the Controller Area Network (CAN)"*.

[5] G. Orengo G.Saggio. "Flex sensor characterization against shape and curvature changes". In: *Sensors and Actuators A 273 221–231* (2018).

[6] Honeywell. *"HSCSAND001BGAA5 Datasheet"*.

[7] Invesense. *"MPU9250 Datasheet"*.

[8] Jared Becker Jonathan Valdez. *"Understanding the I 2C Bus"*.

[9] Sebastian O.H. Madgwick. "An effcient orientation flter for inertial and inertial/magnetic sensor arrays". In: (2010).

[10] P. Palmeri. "A deployable and inflatable robotic arm concept for aerospace applications". In: *2021 IEEE 8th International Workshop on Metrology for AeroSpace* ().

[11] Spectra Symbol. *"Flex Sensor Datasheet"*.

[12] T-motor. *"AK8080 Datasheet"*.

[13] M. Troise. "Preliminary Analysis of a Lightweight and Deployable Soft Robot for Space Applications". In: *Appl. Sci. 2021, 11, 2558* ().