POLITECNICO DI TORINO

Master's Degree in

Mechatronic Engineering

Master's Thesis

Design and development of control system for TASI Docking mechanism prototype

---

Supervisor
Marcello Chiaberge
Co-Supervisor
Andrea Merlo

Candidate
Paolo Bocchiardi

Academic Year 2019/2020

To my family and Snoopy

Abstract

This thesis presents the design procedure of a control system for a docking mechanism, developed for on-orbit servicing of cooperative satellites. The mechanism prototype is part of the SAPERE-STRONG mission, with the aim of performing an active docking procedure between low-masses satellites, recovering the residual misalignments during the last approach and dissipating the energy associated with the relative velocities between spacecrafts upon contact. The chaser element is the active part and consist of a central docking mechanism, with a sliding probe structure. The control development started from the study of the mechanical behaviour of the mechanism and of the available electrical hardware. The technical specifications of actuators, motor drivers and sensors were examined and a suitable control architecture has been selected. The final solution consists of a central control system, where an Arduino MEGA board is the master controller and the motor drivers are the slave elements. So the control algorithm was implemented onto Arduino board, which manages the actuators motions and the sensors measurements, communicating with drivers through rs232 serial communication protocol. The docking procedure consist of several phases: the preliminary mechanism positioning, the alignment, the soft docking, the probe re-centering, the retraction and the hard docking stage. Their management is done by the use of a finite state machine. The motion of the motors and the mechanical movements are performed using the closed loop position and velocity control theory. The results of the first design iteration are tested over a dedicated test bench in no-load conditions, where it is checked if the devices work properly and if the control algorithm runs without issues. After, the electrical components are mounted over the mechanism and the laboratory tests are done. A set of mating trials between the two parts are performed and the control parameters are regulated, in order to meet the requirements. Also a PCB shield is designed, in order to integrate all the additional Arduino hardware onto a single board. The final obtained result is a working prototype of an active docking mechanism, ready to be subjected to the final mating test, which consist of a docking simulation by using a robotic manipulator. The active part will be still while the passive one will be mounted over the manipulator, that simulates the relative motion between the docking halves through apposite trajectory generation techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis work starts from the need of an automatic control system for an existing mechanical plant studied for the mating between two small sized satellites. This work is part of the SAPERE-STRONG research, a program linked to the on-orbit servicing and coordinated by Thales Alenia Space.

The target of this thesis is to design and develop a reliable control system that will be integrated onto the mechanism. It will interface with the hardware in order to execute all the docking operations in an automatic way. The union of the mechanical part and the control architecture will give a full working prototype of an original docking mechanism that will be a first solution to test.

Below the structure of the thesis and a brief description of the chapters content are exposed. The dissertation, including this introductory part, consist of 6 chapters:

- in chapter 2 (State-of-the-Art) the docking and berthing existing solutions are studied. Here the inherent definitions to docking and berthing operations are described, when it is necessary the mating of two satellites and what are the design parameters that need to take into account. Moreover it is given an overview of the evolution of the adopted solutions by main space agencies.

- chapter 3 (The STRONG docking Mechanism) explains the design process of the yet developed mechanical device, referring to the SAPERE-STRONG research work and underlining the advantages that the found solution has with respect to the existing ones. Also a detailed description of the mechanical features and of the adopted electrical devices are given.

- chapter 4 (Control System) reports the design process of the control system that led to the final control architecture. Here, the adopted design architecture and strategies are explained and a detailed description of the docking stages performed through a Finite State machine is given. Also it is reported the Arduino code, with a step by step explanation of what are the actions performed.

- chapter 5 (Experimental Tests) describes the two operative tests that are executed in laboratory, explaining the test procedure and the reached results. Also an introduction of the final test (still to be performed) with the use of a manipulator is given, where it is shown the development of the final PCB control board.
- finally the chapter 6 (Conclusions) sum up the obtained results in this thesis work and lists the remaining issues to solve, suggesting the possible solutions.

Chapter 2

State-of-the-art

This chapter gives a general overview of what is a mating between two satellites and what are the parameters to take into account in order to classify the different solutions adopted, including a brief description of time evolution and some examples coming from different space agencies. This reference would be an introduction of the scenario in which this thesis work takes place.

2.1 Definition of mating between satellites

The mating between two satellites is the union, in a solid mechanical way, of two space vehicles in on-orbit conditions. This becomes essential in the on-orbit servicing missions when, for example, there is the necessity to transfer cargo, crew or when a 'in loco' reparation is needed [13]. A clear examples of how much the on-orbit missions are important could be the assembly operations of the ISS modules. Another example is the several refurbishment missions over the Hubble telescope [19], where the measuring instruments are replaced and updated [10].

In order to mate two spacecrafts, the two parts need to be equipped with a particular mechanism, able to create a solid mechanical coupling between them. Usually, the two satellites are called chaser and target, where the first one has the active role and the target stays in its relative kinematic conditions. The mating operation can be done using two different approaches: the docking approach and the berthing approach.

- in the **DOCKING APPROACH** [11] the chaser evaluates its relative state with the target. So, after reaching the needed conditions (in terms of alignment, relative velocities, etc), it captures the target and connects the two elements.

  A docking operation is usually divided into four phases:

  1. **approach** the chaser starts to move toward the target freely and deploys the mechanism, ready for the impact

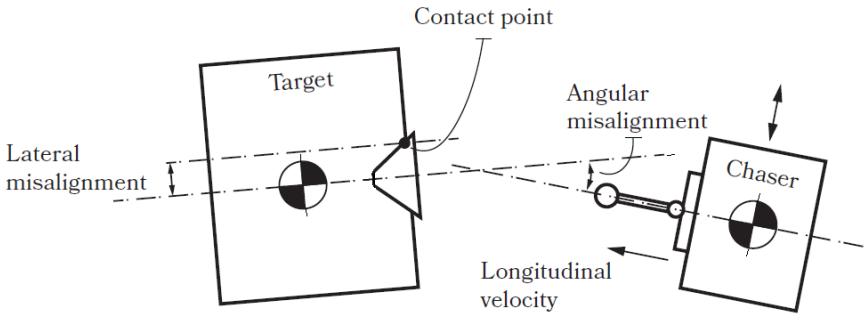

Figure 2.1: The Docking Operation

2. **alignment** thanks to a dedicated system, the poses interfaces of chaser and target are aligned
3. **soft docking** a first connection is executed, the two satellites aren't solidly connected yet, but thanks to this phase the energy related to the relative linear velocities is dissipated
4. **hard docking** a dedicated mechanism executes a more stiff connection, making chaser and target cohesive. With this phase the docking is completed.

A docking system architecture can be of two types: central or peripheral. In the first case the chaser has a male component (a probe or rod), while the target has the female part, that guides the probe toward the desired position. In the peripheral architecture all the docking mechanism is in the perimetric part, both in the chaser and in the target. The central solution has the advantage to be simple to study and develop, but it has a drawback: in an on-orbit operation where the mission is to link two spacecraft in order to create an internal passage for cargo or astronauts, the central probe obstructs the the central space. The peripheral architecture gives the solution of this problem, leaving free the central zone of the linking.

- the **BERTHING APPROACH** [**berthing˙op**] consist of the use of a manipulator for the mating [15]. The chaser reach a suitable position in space and ensure that all the relative velocities (linear and angular) with respect to spacecrafts are null. Then a manipulator, mounted either on the chaser or on the target, catch the other part and actuates the mating approach.

  Usually a berthing operation consists of seven phases:

  1. **positioning of the manipulator in ready condition** the manipulator is positioned in a portion of space such that it is possible to start the operations.

Figure 2.2: Cygnus Berthing Operation (Credit by NASA)

2. **acquisition of the berthing box** the chaser reaches a delimited space inside the workspace of the robot manipulator.
3. **initiation of capture** the chaser's thrusters are switched off and the end-effector of the manipulator starts to reach the grappling fixture over the chaser.
4. **grappling** the end-effector reach the desired pose and capture is done.
5. **transfer to the berthing port** the manipulator drags the chaser, in order to attach together the link interfaces of the two spacecrafts.
6. **insertion into the reception interfaces** the relative final pose is reached and the manipulator push the attachments one inside the other. A properly designed guide makes the fine alignment between the two interfaces.
7. **structural connection** a device fastens the connection and a solid union is done.

In this dissertation only the docking solution are taken into account.

2.2 Some examples of docking systems

The first docking operation ever was performed in 1966, during the NASA Gemini-VII mission. After a year, in 1967, also the Soviet space agency tested his first docking system during the Soyuz program. In both cases they opted for a central architecture, with a probe and a female part. But the main problem of this solution was its inconvenience when it needs to transfer cargo or passengers from the chaser to the target. So,

thanks to the collaboration between NASA and Roscosmos (on the Apollo-Soyuz Test Project, a.k.a ASTP), on 1975 the first peripheral docking mechanism was tested, becoming the basis for the development of the first Androgynous Peripheral Attachment System (APAS). The APAS technology will be used for several missions, like Buran orbiter, Shuttle-MIR and more recently in the Chinese Shenzhou missions.

In 2010 all the main space agencies collaborates, establishing a common interface to use. According to [2] they agreed to use a standard docking interface to enable on-orbit crew rescue operations and joint collaborative endeavors utilizing different spacecraft. This is also the standard used in the ISS orbiting station and the latest technologies are adaptations of this standard.

2.2.1 Gemini and Soyuz docking systems

They are the first two docking solutions implemented in the space exploration history. In all the two cases a central architecture was used. In the Gemini mission [**gemini**] the mechanism consists of a rigid male cone as probe, while a cup interface was used as drogue, linked to the target spacecraft by seven shock absorbers to dampen relative longitudinal and lateral velocities. The longitudinal shock absorbers were equipped with an orifice damper and a spring in parallel for reusability. The probe was equipped with an alignment system that had as its counterpart a v-shaped guide in the female cone. So, there was only a possible coupling configuration between the parts. The final capture was accomplished by three latches.

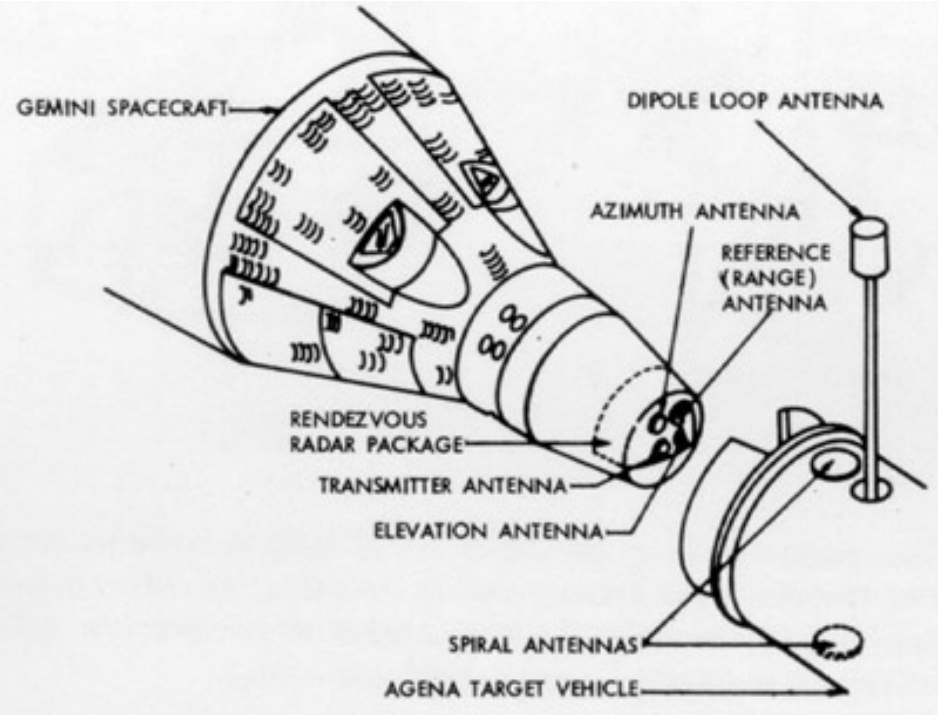

Figure 2.3: Gemini docking mechanism

Also on the Soyuz mission a probe and drogue system was used (2.4). The first one was equipped with a small ball screw and a large one that acts the shock attenuation. The retraction of this element causes the compression of a coil and a Belleville spring as well as the rotation of an electromechanical brake. The capture was done using two latches on the probe head that reached the female socket. A transducer on the head of the probe verified the end of the operation. Through the large ball screw the retraction of the probe was performed and the misalignment was deleted by female guides. This solution was used for the first time in 1967 and it's still in use today, with some reviews: in order to create a transfer tunnel between the two satellites the original design was modified, making the probe and drogue mechanism part of the hatches and giving to mechanism a more compact structure.

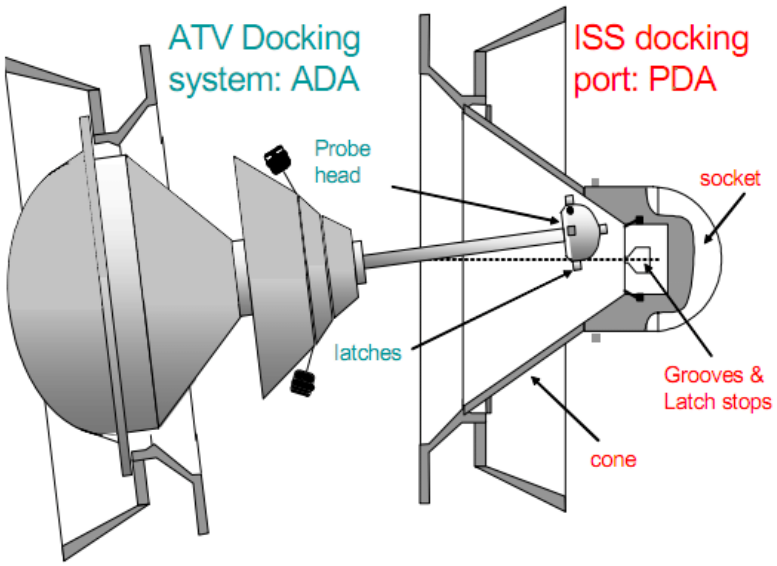

Figure 2.4: Soyuz Docking mechanism

2.2.2 The APAS solution

This kind of docking mechanism comes from the ASTP mechanism. It is the result of the cooperation between American and Russian space agencies [14], that developed the first peripheral and androgynous docking mechanism. The fact that it was androgynous means that both the two sides are designed to work actively or not, implying that if one of the two halves fails the other one could be activated, increasing the probability of success. The design concept includes a ring equipped with guides and capture latches that were located on movable rods serving as attenuators and retracting actuators, and a docking ring on which are located peripheral mating capture latches with a docking seal. Regarding the attenuation technology the two country decided to use different solutions: the American used hydraulic dampers while Russians adopted their own

electromechanical brakes.

In 1989 this technology evolved in the APAS-89 (Androgynous Peripheral Attachment System) [**apas**]. The main changes were: the adoption of the EMB attenuation technology, the mechanical latches, loaded by the use of springs for soft docking and the re-design of the guides, that from an outwards configuration adopted an inwards one. The APAS was originally studied for the Buran spaceship. It's subsequent version were used in different important missions, like Shuttle-ISS (APAS-95) and the Chinese mission (APAS-2010).

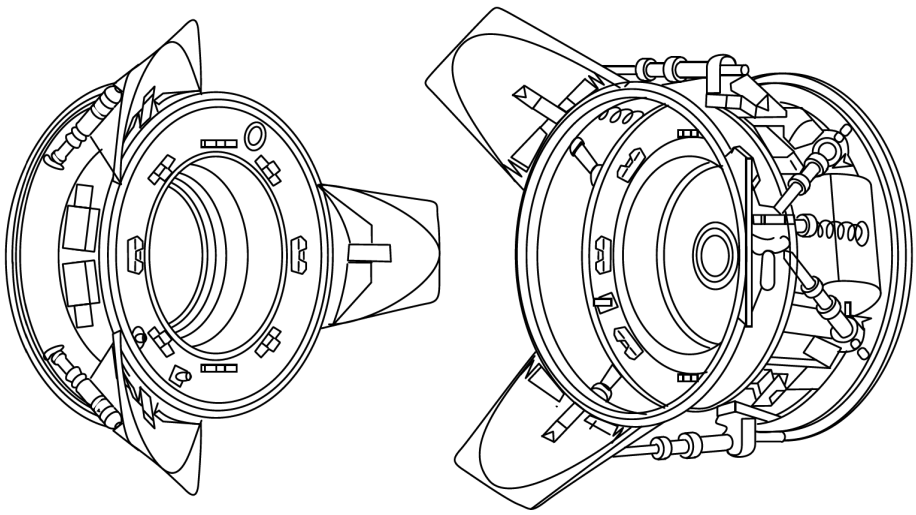

Figure 2.5: APAS Docking mechanism

2.2.3 The IDSS

The IDSS (International Docking System Standard) is the international standard mechanism for the spacecraft mating. The first project was developed in 2010 by a collaboration between the main space agencies. The purpose is to provide unique basic common design parameters that must be followed to create a mating mechanism to use with the International Space Agency (ISS) [3]. It is a peripheral androgynous system and it permits the transfer of cargo, crew, energy and data through dedicated connections. It consists of two identical elements. Every part consists of a ring that represents the mating surface, three guide petals and a set of guide pins for the correct alignment. A set of mechanical capture latches represent the soft capture system, while the hard capture is performed by a set of hooks placed along the mating surface. An example of a system compliant with the IDSS is the NDS (NASA Docking System) [2]. It is the evolution of the older APAS mechanism. The ring is equipped with electromechanical actuators, forming an active Stewart-Gough platform. Another example of IDSS compliant mechanism is the IBDM (International Berthing and Docking mechanism), developed by ESA. This mechanism has to be tested in space yet.

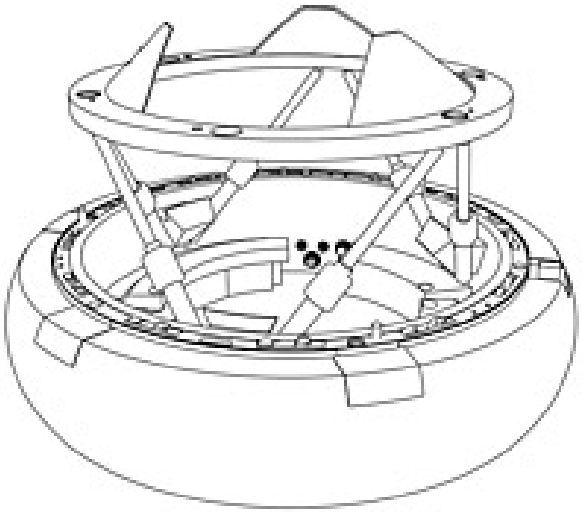

Figure 2.6: NASA Docking Mechanism in extended position, compliant with the IDSS standards. Credit by NASA

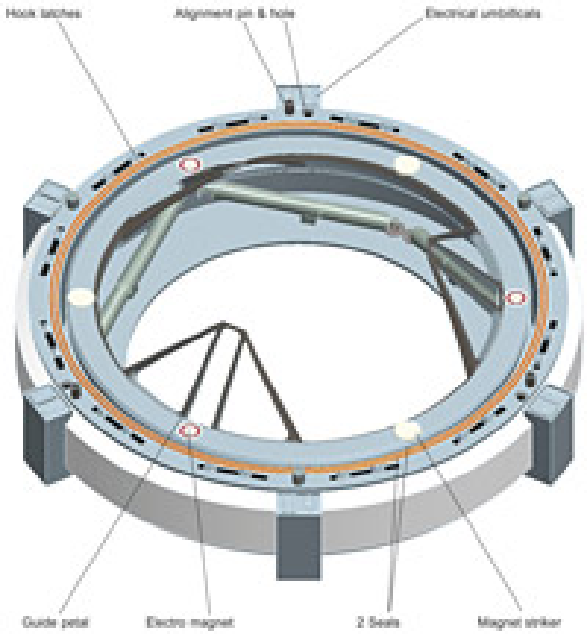

Figure 2.7: IBDM docking mechanism, compliant with the IDSS standards. Credit by ESA

Chapter 3

The STRONG docking mechanism

This chapter describes the concept and the evolution of our docking mechanism, explaining the purpose of the mission in which it take place and the found solutions.

3.1 The SAPERE-STRONG mission

The research work is part of the SAPERE-STRONG project, guided by Thales Alenia Space. It's aim is to develop the technologies for cooperative on-orbit missions inside the space exploration. In particular in the STRONG mission [20] the target is to achieve a reusable Space Tug able to reach the mating between an orbiting satellite, in order to optimize the launch's operations and transfers satellite platforms and tools from low orbits to the final ones. Also, this Space Tug must be able to perform the on-orbit refueling from an orbital tank. This permits to execute the on-orbit tasks with a reasonable reduction of fuel consumption and an optimization of the operations. In the image 3.1 is represented the behaviour of the chaser spacecraft during the mission.

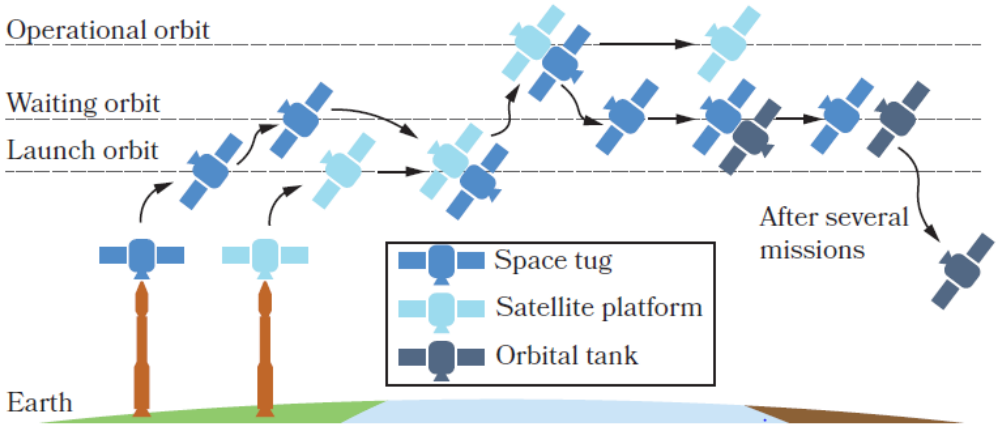

Figure 3.1: Operations during the STRONG mission

An important component of this Space Tug is the docking mechanism. It must be studied to accomplish the following operations:

- it has to recover the misalignment between the two spacecraft, both in linear and angular positions
- it has to dissipate the contact forces during the operation, avoiding excessive stresses over the mechanism and mating failures
- it must be able to create a sufficient solid connection between the two parts

An important factor to take into account is the payload: the Space Tug will dock with target spacecrafts (orbiting tank and small satellites..) that have masses that goes from 1000kg to 3000kg, so the inertial properties of the bodies are consistent. In the next paragraph will be described the found solution starting from this initial premises.

3.2 Mechanism Description

The mechanism took into consideration in this work thesis was developed on the doctoral study "Design, modeling, and testing of a space docking mechanism for cooperative on-orbit servicing" [16]. What will be done is to learn how this device was developed, what are the specifications and constraints considered and what kind of hardware was chosen.

The architecture for the mechanism was selected through a trade-off study, starting from the examination of main existing technologies. For each of these it was studied the compliance with specific criteria, that are the following:

- **Mass loads**: target plus chaser total mass, less is the mass less is the mission cost. Also is considered the capability of managing different load masses.
- **Mechatronic complexity**: the hardware complexity, it considers the mechanical complexity, the number and type of the sensors and actuators used.
- **Energy consumption**: necessary energy to drive actuators, sensors and all the hardware over the mechanism.
- **Reliability**
- **Functional confidence**

At the end of the examination, the chosen architecture as the best compromise consists of an active central docking mechanism, composed by a female passive part, to catch and linked to the target spacecraft, and a male part, the active element, linked to the chaser spacecraft. The mechanism was studied to perform a phase of approaching,

where the misalignment is avoided reaching a first catching, called soft docking, and a phase where the two elements are solidly engaged, called hard docking.

3.2.1 The male part

With reference of image (3.2) the male element is composed of two main parts: the probe (Green colored) and the base plate.

The **probe** consists of an internal part and an external one. This two parts can slide one respect to the other, giving the possibility to extend and retract the component along its longitudinal axis. This movement is activated by a ball screw, moved by an electric motor. The rotation between the two elements is constrained by apposite slotted guides. The probe is equipped with three latches (8) at its edge, normally opened by springs. Their retraction is controlled by the translational movement of the probe through the combined use of three ball plungers, used also to keep in place the probe, and three ball notches inserted over the external part (1). In this way it is possible to control the translation and the position of the latches with just a single motor. At the top of the probe there is also a spring equipped shock absorber (5). At the base of the probe there is a semi-conical element (9), connected to three compression springs (10), that can be moved along the longitudinal probe axis. it is used to store the energy necessary for the undocking operations. At the bottom of the male cone three rods are fitted (11), they will take part of the passive alignment mechanism between male part and the female one.

The **base plate** is composed by three layers: the hard docking mechanism, the active alignment mechanism and the sensors layer.

- The *hard docking mechanism* is the one that will execute the final strong connection with the target. It consist of a plain accommodation (4) and a set of three hooks (12) that can moves radially respect to the circular section. The hook's position is controlled by a slider connected to a roller follower inside a cam profile. The cams are obtained over a geared ring that can be rotated through a motor.

- The *active alignment mechanism* is accommodated below the plate of the hard docking layer. Here the probe is connected to a moving basement with an universal joint (2), that permits the tilt of the probe around the two normal axes respect to the base plane. A set of extension springs (6) permits to recover the orthogonal initial position of the probe. The base where the joint take place is a rail slide, mounted over an another slide (the red coloured parts) orthogonal to the first. So the joint, together with the probe, can translate along the hard docking basement plane. The motion of the rail slides are performed through a couple of ball nuts, bolted solidly with the slides and coupled with two ball

screws. The motion is actuated by two motors, geared to the screws. Its rotation moves the ball nuts along their length, translating the slides with them.

- Above the alignment mechanism there is the *sensors accommodation*. A basement is bolted under the last rail slide, over it the sensors (7) are positioned such that they can measure the tilting of the bottom part of the probe.

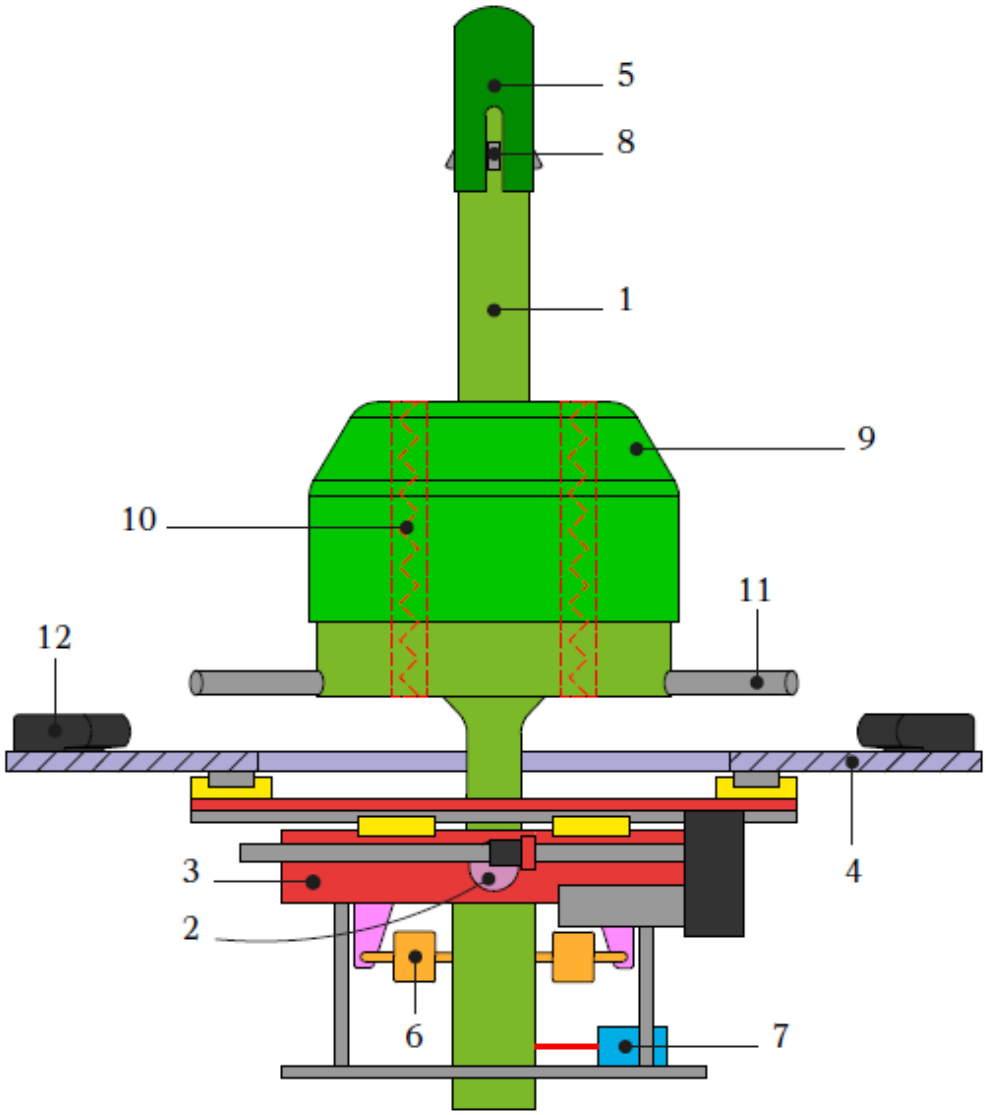

Figure 3.2: 2D Design model of the male part, side view

3.2.2 The female part

The female part (3.3) consist of a conical frustum that ends with a cylindrical element. It has the purpose of acting as a passive guide for the probe, that have to reach the bottom section. The border of the cone is designed to match correctly the hard docking mechanism of the male part. Along the cone surface three V-shaped slots are made.

They are the passive guidelines that will match the probe rods and will perform the angular alignment. At the vertex of the cylinder a socket is present. It will be reached by the probe latches performing the soft docking.

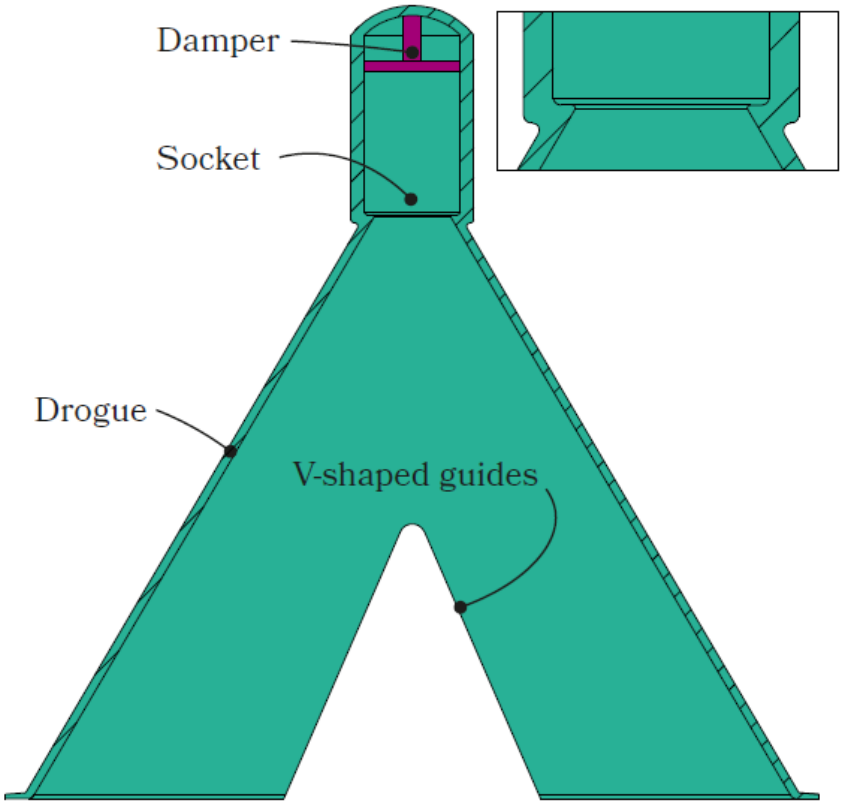

Figure 3.3: Cross section of the Female part

3.3 The docking stages

The docking manoeuvre is composed by a set of operations, called **stages**, executed sequentially. Below these stages are listed and explained:

- **Setup**: the device is switched on and checked, verifying that the mechanism is correctly positioned and ready to start.
- **Full Extension**: the probe is fully extended at its maximum position and the petals are opened. In this stage the probe is ready to perform the soft docking.
- **Alignment and soft docking**: the device is enabled to move the probe's base along the hard docking surface. In this stage the mechanism search the accommodation with the target inside the female part, avoiding the excessive contact forces over the probe. Once the petals passes across the narrowest part of the cone, the target is hooked and the soft docking is done.

- **precharge**: the probe is partially retracted pushing the target cone against the springed conical element of the chaser. This action makes the connection stronger and permits to store energy, that will be used after during a future undocking operation.

- **Homing**: the probe recovers its zero position at the centre of the mechanism. So the female cone is aligned with the hard docking mechanism with respect to the linear displacements.

- **full retraction**: the probe is fully retracted, bringing the cone margins in contact with the base plate. During the motion the angular displacement is deleted thanks to the probe's rods and the V-shaped guides over the cone.

- **hard docking**: once the female element leans correctly against the base plate, the geared ring rotates and moves the hooks, that block the cone. At this point the docking manoeuvre can be considered complete.

The behaviour of the mechanism during the manoeuvre can be described using a Finite State Machine diagram, where the states are the docking stages. In the figure (3.4) the FSM diagram is shown.

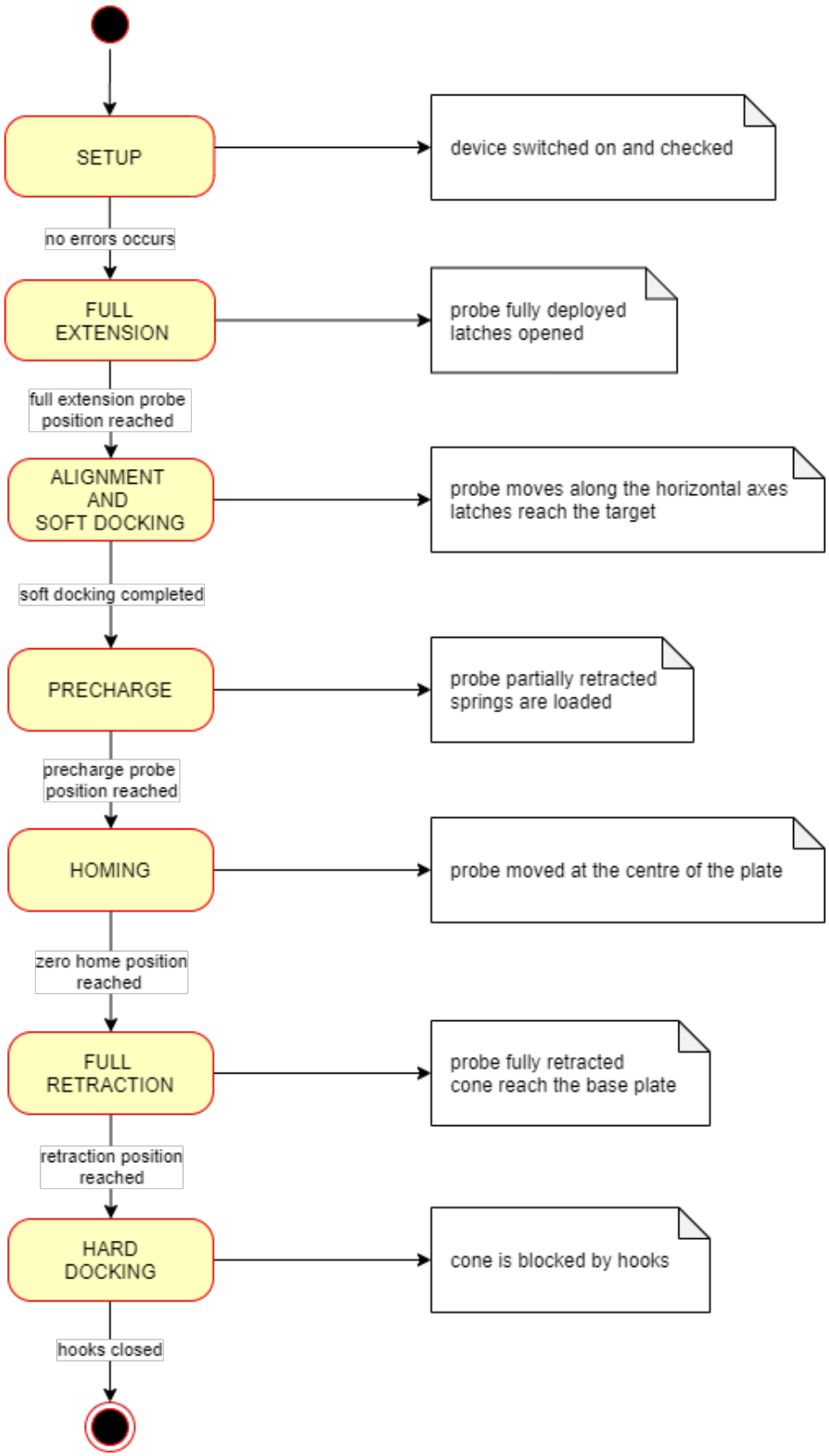

Figure 3.4: FSM of the docking manoeuvre

3.4 Electrical hardware

The active element is equipped with a set of actuators and sensors that performs the motion of the mechanism. There are a total of four motors, three of which are EC brushless motors and the remaining one is a stepper. For each EC motor an Hall sensor and an encoder are used to get information about the rotor angular positions, velocities and accelerations. Also a couple of optosensors are used to obtain information about the tilting of the probe. In table 3.1 are reported briefly the components, their tasks and the stages which are involved:

COMPONENT	**DESCRIPTION**	**TASK**	**INVOLVED STAGES**
Maxxon EC 22 100W (2X)	EC Motor with integrated Hall sensor, Encoder	movement along normal axes	Alignment Homing Full retraction
Maxxon EC-max 16 8W	EC Motor with integrated Hall sensor, Encoder	actuation of the probe mechanism	Full extension Precharge Full retraction
CTM21NLF25	Stepper motor	actuation of the geared ring	Hard docking
optoNCDT 1420 (2X)	Position laser sensor	caption of the probe's tilting	Alignment

Table 3.1: Electrical component and their functionalities

3.4.1 The Motors

The EC motors are supplied by the Maxxon Motor Company. They are brushless motors equipped with an incremental encoder and Hall sensor. According to the Maxxon Motor catalog [7] after the specifications of the used motors are listed (tables 3.2 and 3.3):

Motor Name	**Maxxon EC 22**
Nominal Power [W]	100
Nominal Voltage [V]	24
Nominal Speed [rpm]	27000
Nominal Torque [mNm]	48.5
Encoder Type	Incremental, Relative
Encoder Resolution [CTP]	128/256/512

Table 3.2: EC 22 motor specification

The **EC22** model is used to apply the motion of the probe structure along the horizontal axes. Its Nominal speed data is selected to accomplish the speed response requirement during the alignment phase. Also the nominal torque and power are evaluated to make sure that the mechanism is able to displace the inertial mass of the target spacecraft during the Homing phase.

Motor Name	**Maxxon EC-max 16**
Nominal Power [W]	8
Nominal Voltage [V]	24
Nominal Speed [rpm]	7350
Nominal Torque [mNm]	8.19
Encoder Type	Incremental, Relative
Encoder Resolution [CTP]	128/256/512

Table 3.3: EC-max 16 motor specification

The **EC-max 16** model is employed for the operations of extension and retraction of the probe. It doesn't need to operate at high speeds so its speed specification is sufficient for our purposes. A planetary Gearhead is combined with it. Its specification is reported (table 3.4):

Transmission rate	157:1
Nominal Efficiency	73%
Max. Input Speed [rpm]	12000
Max. Continuous Torque [Nm]	0.4

Table 3.4: Gearhead specification

Regarding the Stepper motor it is supplied by Danaher Motion. It will actuate the Hard docking mechanism. In table 4.1 are reported the main characteristic:

Motor name	CTM21NLF25FA
Supply DC Voltage [V]	24
Detent Torque [Nm]	0.092
Holding Torque [Nm]	1.84
Detent Torque [Nm]	0.092
Step Angle [DEG]	1.8 °
Step Accuracy	±3%

Table 3.5: Stepper motor specification

3.4.2 The motors drivers

Each motor need to have a driver unit to work properly and to control them. Below the drivers and the coupled actuators are listed:

Motor	Driver
EC 22 (X2)	Epos2 24/5 (X2)
EC-max 16	Epos2 24/2
CTM21 Stepper	P70530

Table 3.6: actuators and their coupled drivers

For the Maxxon motors the Epos2 family controller are selected [9] [6]. They are programmable controllers, able to perform several different control techniques over the motors. The programming is done through the writing and the reading of its dedicated registers.

They are supplied with a DC voltage of 24V, the maximum absorbed current is 5A in continuous conditions. The connection with the motors is done through the dedicated connectors, both for the motor supplies and the sensors outputs.

The Epos2 drivers has the option to operate with analog and digital signals, having dedicated input and output pins. They are multi-purpose inputs and outputs, configurable via software. Also several communication ports are available to program the device and monitor the operations. Below a list of the available input/output ports(3.7) (3.9) and of the communication ports (3.8) (3.10) is given:

Inputs and Outputs	
N. of digital inputs	6
N. of digital outputs	4
Input Voltage [V]	0...30
N. of Analog inputs	2
Input voltage [V]	0...5
Resolution [bit]	12
Banwith [kHz]	5

Table 3.7: Epos2 24/5 Inputs and Outputs

Communication		
Serial Port	Serial protocol	Max. bit rate
USB	USB Standard 2.0	12Mbit/s
RS232	EIA RS232 Standard	115200bit/s
CAN	CANopen DS-301	1Mbit/s

Table 3.8: Epos2 24/5 Communication Ports

Inputs and Outputs	
N. of digital inputs	6
N. of digital outputs	2 (general purpose)
Input Voltage [V]	+2.4...+24
N. of Analog inputs	2
Input voltage [V]	0...+5
Signal Resolution [bit]	12

Table 3.9: Epos2 24/2 Inputs and Outputs

Communication		
Serial Port	Serial protocol	Max. bit rate
USB	USB Standard 2.0	12Mbit/s
RS232	EIA RS232 Standard	115200bit/s
CAN	CANopen DS-301	1Mbit/s

Table 3.10: Epos2 24/2 Communication Ports

3.4.3 The sensors

The mechanism will be equipped with a couple of optical sensors, mounted under the hard docking baseplate. They are laser pulsed sensors able to measure the distance between the beam source and the surface where the laser spot lays [12]. Here in table 3.11 the main technical specification:

Supplier	Micro-Epsilon		
Name	optoNCDT 1420		
Supply Voltage [V]	11-30		
Measuring Range [mm]	Measuring Start Range	Midrange	End Range
	25	37.5	50
Measuring Rate [Hz]	250/500/1k/2k/4k		
Current signal range [mA]	4-20		
Communication Port	Serial RS422		

Table 3.11: Laser sensors specification

The measurement comes from a current analog signal. If the sensor is working in out of range conditions, the output current is set to 3mA, otherwise the output assumes a value between 4 and 20mA. In alternative a serial communication port is available. It can be used to setup the controller sensor and read the measurement through a digital data.

Chapter 4

Control system

Starting from the available hardware it needs to develop a control system able to manage the electrical components and execute all the docking stages in a correct way. In this chapter is exposed the design process and the implementation of the main controller.

4.1 The Control Architecture

The control configuration to use is selected considering the following requirement:

- it must be possible to mount the control system over the docking mechanism, in order to develop a full working prototype that doesn't need external control devices.
- the controller must be lightweight and compact, cause of the previous requirement
- the system needs to work in real-time
- low-power consumption devices is preferred, saving up precious energy coming from the spacecraft.

The chosen architecture consists of a central control system, with a Master Control Unit board and the slave elements linked to it, represented by the motor drivers. The master control unit must be able to communicate with the slave boards, giving to them the needed commands, and make the acquisition and the elaboration of the sensors measurements. In figure 4.1 the configuration is exposed:

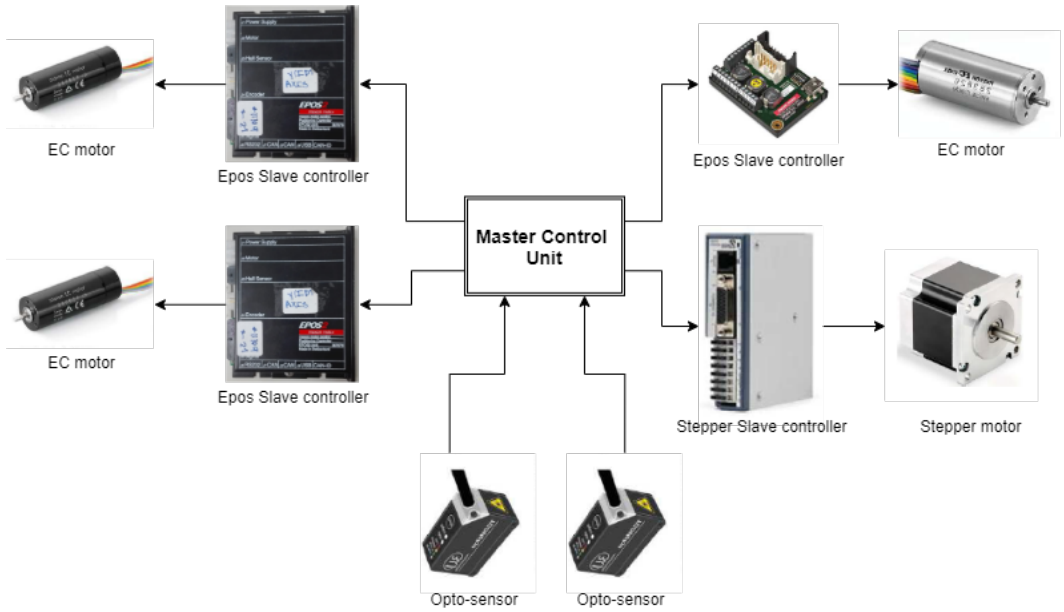

Figure 4.1: Control architecture

For this task it is opted to use an Arduino board [18]. It is a good choice for several reasons:

- it is compact and easy to integrate over the mechanism.
- it works in real-time and it is a device that works with relatively low powers
- it is easy to program and powerful in the prototyping phases, because of its flexibility of use. During the project's develop it is very simple to apply upgrades, reducing the development's time
- it is possible to acquire, manage and generate a large number of digital and analog signals, which makes it a good platform for mechatronic applications
- it is compatible with the communication protocols available on our Epos motor drivers

For our purposes it is chosen an Arduino MEGA model 4.2, because of its number of available serial communication ports (four) and its large number of programmable signal pins.

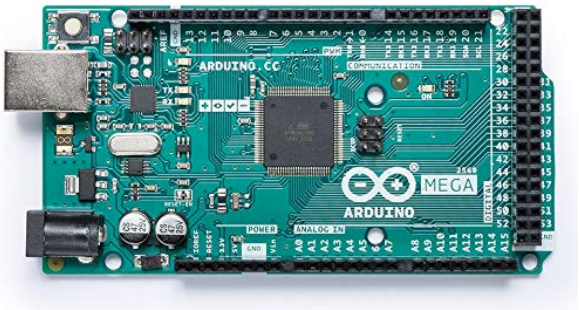

Figure 4.2: Arduino Mega board Model 2560

Below the tecnhical specifications are listed:

Arduino Model	**MEGA 2560 rev3**
Microcontroller	ATmega2560
Operating Voltage [V]	5
Input Voltage (limits) [V]	6-20
Flash Memory [KB]	256, 8 used by bootloader
Clock Speed [MHz]	16
Inputs and Outputs	
N. of Digital I/O Pins	54 (14 provide PWM output)
Operating Voltage of Dig. pins [V]	5
N. of Analog Input Pins	16
Operating Voltage of Analog Pins [V]	5 (TYP)
Input Resolution [bit]	10
DC Current for I/O Pin [mA]	40
Communication	
Supported Communications	TTL, SPI, I2C
Serial Ports Baud Rate [bps]	9600-115200

Table 4.1: Arduino MEGA technical specification

Also more hardware will be used. First of all the opto-sensor's outputs are current signals, so it is necessary to convert them into voltage outputs, making them compatible with the Arduino voltage inputs. A simple resistor will be used, connected between sensor's output and signal ground. The sizing of them is done considering the sensor specifics and following the relation:

$$V = RI.$$

From the datasheet:

$$I_{MAX} = 20mA$$

$$V = 5V$$

that are the maximum output signal current (sensor) and the nominal input voltage for analog pin (Arduino). Finally the result is:

$$R = 250\Omega.$$

In addition it is preferable to insert a voltage follower after the resistor, separating the impedance and avoiding electrical failures.

Regarding the serial communication with the motor drivers, an hardware interface is needed. For this project the rs232 communication protocol is selected, so it needs to interface the TTL port of the Arduino's microcontroller with the rs232 ports of the Epos drivers. This is done placing a MAX3232 integrated transceiver. This device provides to regulate the high and low logic levels of the two ports (0-5V for TTL, 0-15V for rs232) by using charge pumps capacitors.

After these addition the final configuration is the following:

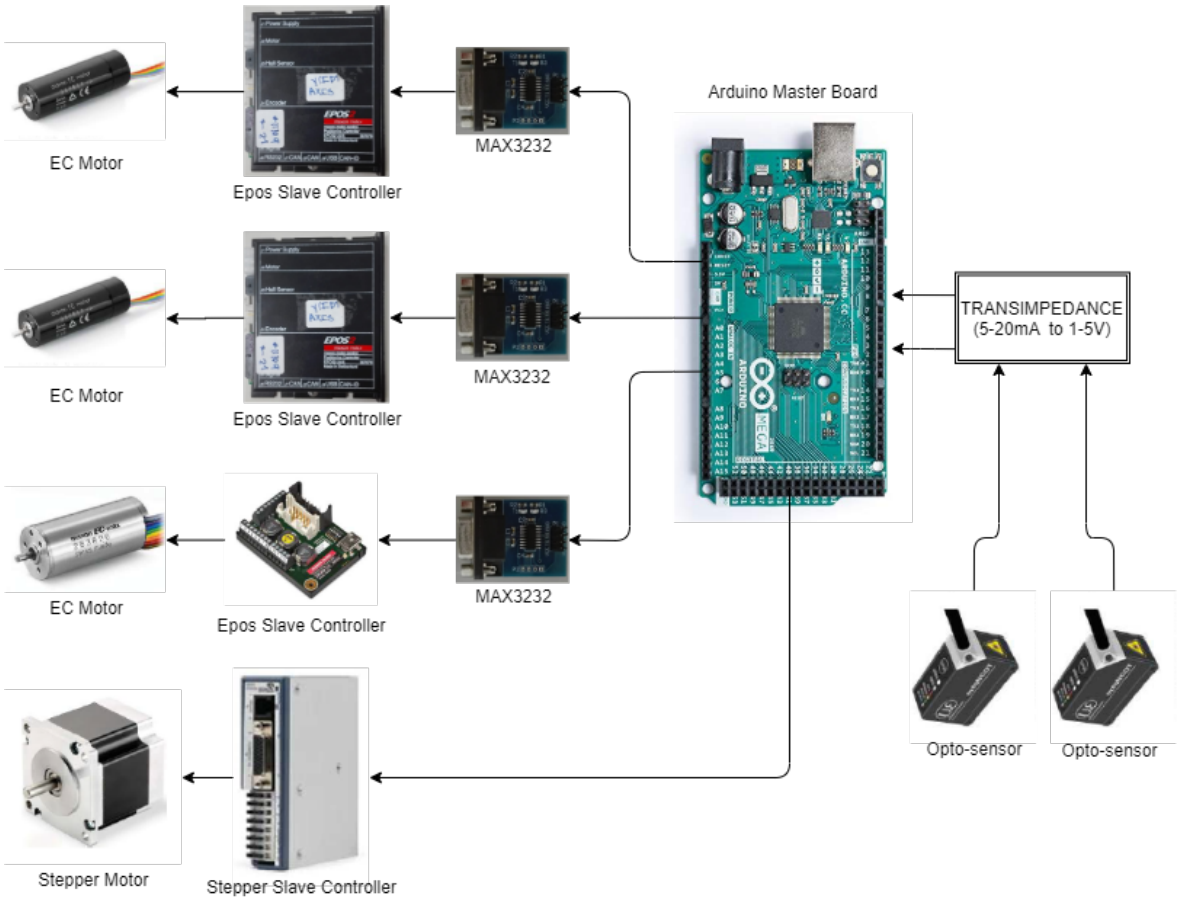

Figure 4.3: Final configuration of Control architecture

4.2 The Control Strategies

The idea is to manage the docking stages inserting it into a Finite State Machine. At each state the Master board has to activate the respective actuators and apply to them the correct control method. In figure 4.4 are shown the docking stages, the involved devices and the implemented control methods:

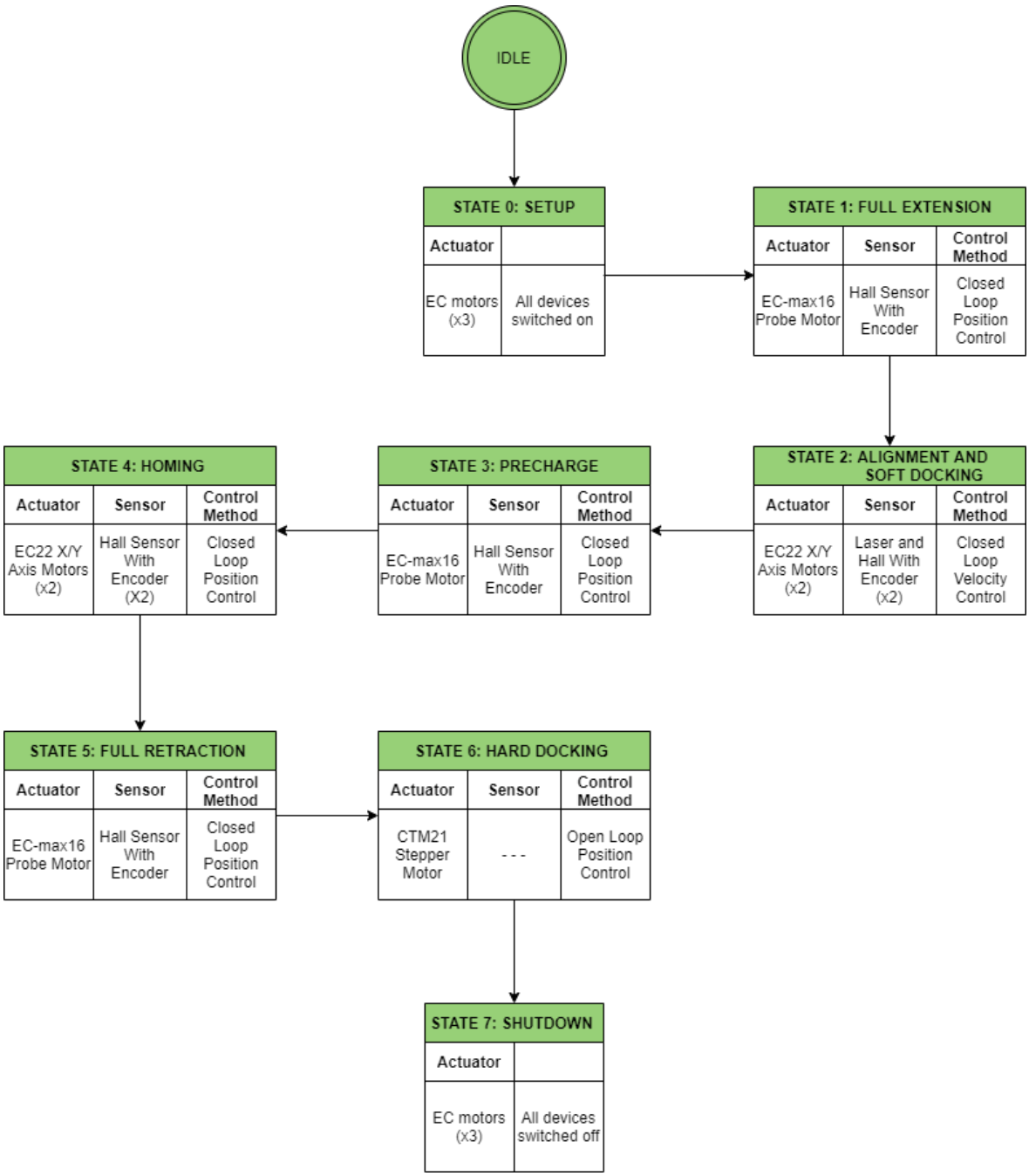

Figure 4.4: Control Flow Diagram

The execution of each state take place in this way:

1. The Arduino board calls in action the right driver and configures it, giving information about the type of control to perform, the input and output to use and

the control parameters to set [4].

2. The Epos driver perform the selected control strategy over the motor

3. After that the Arduino acknowledges that the operation is done, it sets the Epos driver in standby mode and scrolls the State Machine passing to the next state, where it restart from point 1.

Once there are no more states to execute, the Arduino board switch off the Epos drivers and return to Idle state.

In state 1, 3, 4 and 5 a closed loop position control is applied to the motors. The control scheme could be the following:

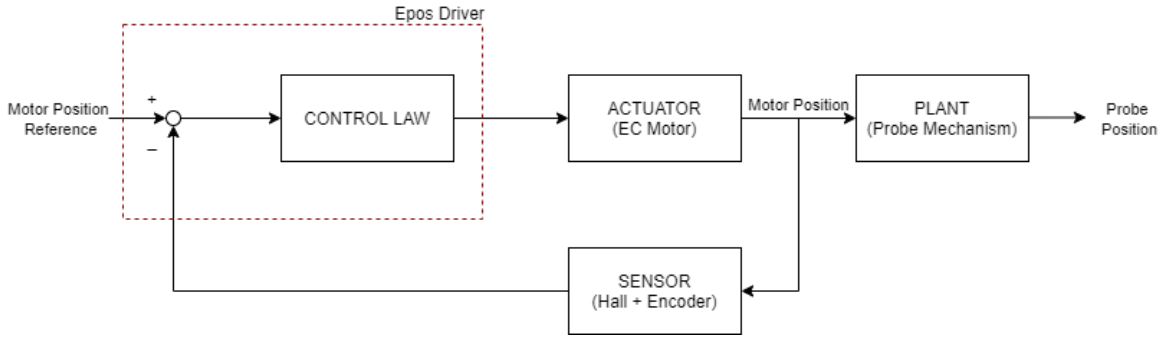

Figure 4.5: Closed Loop position control

The used control parameter is the position of the actuator's rotor. The control law to apply is generated by driver setting the "profile position mode". The position parameter is managed by Epos driver using the "quadcount" unit, defined as:

$$Quadcount = qc = 4 \times \frac{Enc.Counts}{Revolution}$$

where the *Encoder Count/Revolution* ratio is the Encoder resolution. So, remembering that the position sensors are relative and stating that the full retracted position is equal to 0 qc, it is possible to know the motor positions in qc at which the probe is fully extended, precharged and so on. The reported positions in qc are calculated starting from the mechanical dimensions, the fine regulation will be done during the test sessions.

	Full Retraction	Full Extension	Petals Opening	Precharge
Motor position (qc)	0	-5920000	-5284000	-4149000
Probe Position (mm)	145	215	215	200

Table 4.2: probe's positions and relative motor positions

In state 2 a closed loop velocity control is applied to the X and Y axis motors.

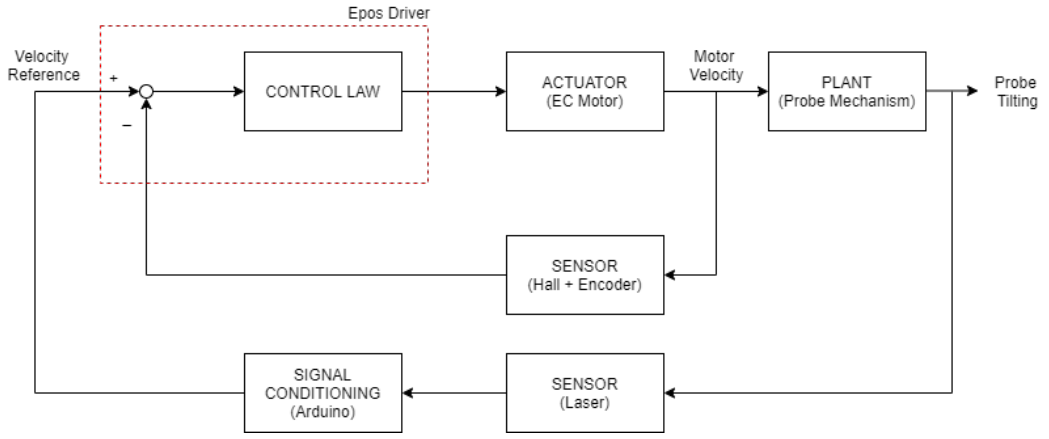

Figure 4.6: Closed Loop Velocity Control

The velocity parameters are measured in rpm (revolutions per minute). So for example, remembering the quadcount definition and considering an encoder resolution of 500 counts/rev.

$$1qc/s = \frac{60}{4 \times 500} = 0.03rpm$$

In Hard Docking Phase (State 6) an open loop is applied, because it is not possible to get feedback information about the angular position. What will be done is to find by trial and error the amount of motor displacement needed to close correctly the hooks. This displacement will be the target position sent to stepper driver.

4.3 Detailed description of the States

In this section it is given a detailed description, for each state, of what is done by Arduino board and motors drivers, and of relative actions applied over the mechanism. All commands names, internal Epos states and internal registers are referenced to *Appendix A*.

4.3.1 State 0: Setup

Once the Start command is received, the process exit to Idle state and the Setup state begins. Here the motor drivers are prepared. The Arduino board send the following commands to all the Epos devices:

- "Fault Reset" command is sent, clearing all the errors flags. Now the switching on is disabled, avoiding any unexpected motor movements
- the "actual position" register is cleaned, making the current position of the motor the zero reference position

- "Shut Down" command is sent, making the Epos device ready to switch on

Once the last command is sent correctly and the acknowledgement reply is received, the process can move forward to the next state.

4.3.2 State 1: Full Extension

In this state the mechanism is positioned at its full extended position. The motion consists of two steps. Firstly, the probe is brought from its totally retracted position to the maximum extension where the latches mechanism (ball plungers) reach the ball notches. The second step consists of a partial probe mechanism retraction, that doesn't move the external part but opens the latches. In Figure 4.7 the motion steps are shown:

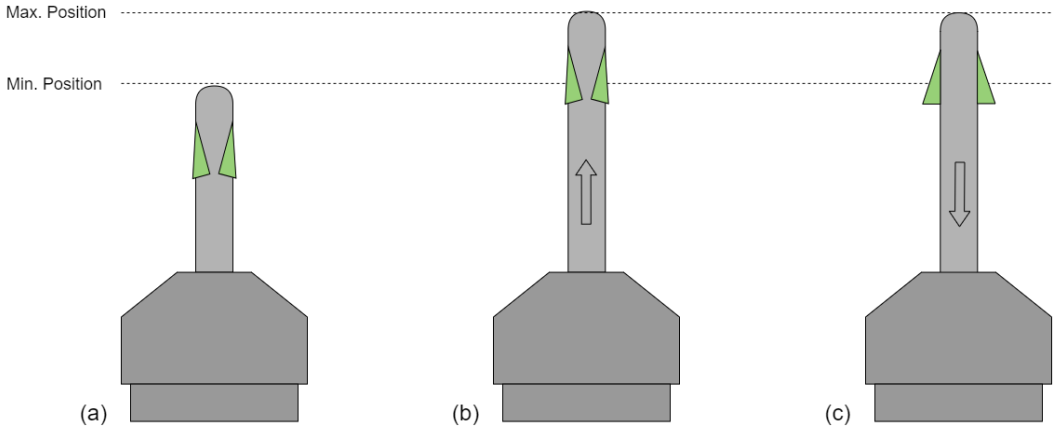

Figure 4.7: Full extension steps: from the full retracted position (a) to maximum extension (b) and the latches opening (c)

To do that the Arduino board configures the driver relative to the probe motor in order to apply a position controlled motion. That is done as follows:

- the modes of operation and the control parameters are set
- the probe's motor is armed, sending the "Switch On" command
- the target position is set, writing it onto the appropriate Epos register
- the motor is switched on sending the "Enable Operation" command
- the motion continues until the target position is reached. Then the motor is stopped by Epos controller
- once the Arduino acknowledges that the target position is reached, it sends the "Quick Stop" command, disarming the motor

This procedure is done both for full extension and for open the petals, where a counterwise motion of the motor is applied. When the operation is finished and no errors occurs, the state machine goes on.

4.3.3 State 2: Alignment and Soft Docking

Here the mechanism has to be prepared for the alignment with the soft docking female target. To do that the Arduino selects the X and Y axes motor drivers and performs a series of operations, as depicted in figure 4.8:

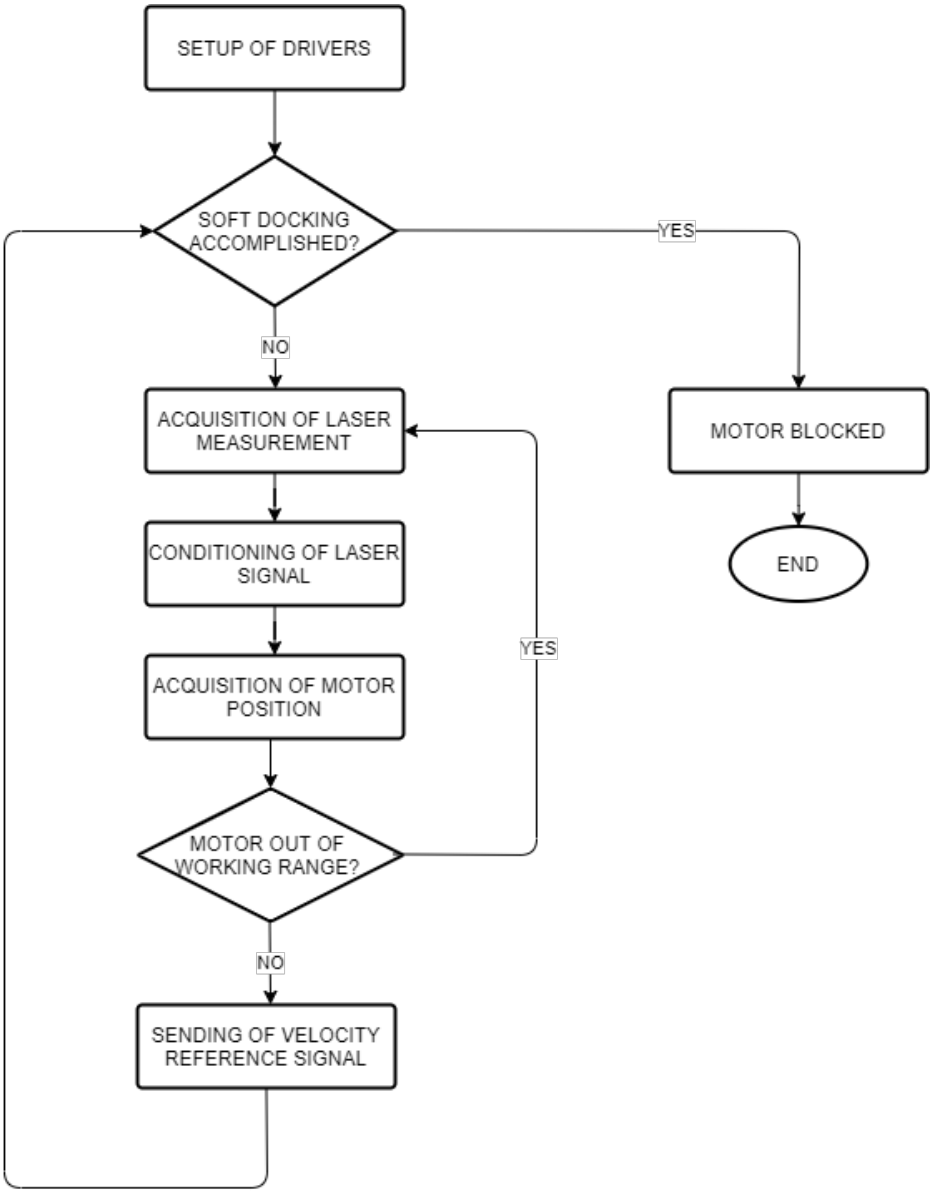

Figure 4.8: Flow Diagram of the Arduino processes during the Alignment phase

The first thing to do is the **setup of drivers**. The modes of operation is set up such that a closed loop velocity control will be applied to the motors. The control parameters are defined, regulating the maximal allowed speed and acceleration. Also the driver is programmed to receive the reference velocity for the control from a specific configurable GPIO pin.

Then the probe alignment process is started. The following operations are performed cyclically, until the soft docking is reached:

The Arduino makes the **acquisition of laser measurement**, obtaining information about how much is the tilting of the mechanism. Then the acquired signal is **conditioned**, in order to generate a reference signal compatible with the Epos input pin. Here a dead band is applied, in order to avoid noises coming from vibrations around the zero degree of tilting position, and through a DAC module a continuous voltage signal is created. In Figure 4.9 is shown how the reference signal is done, starting from the tilting measurement, and how the conversions are done.

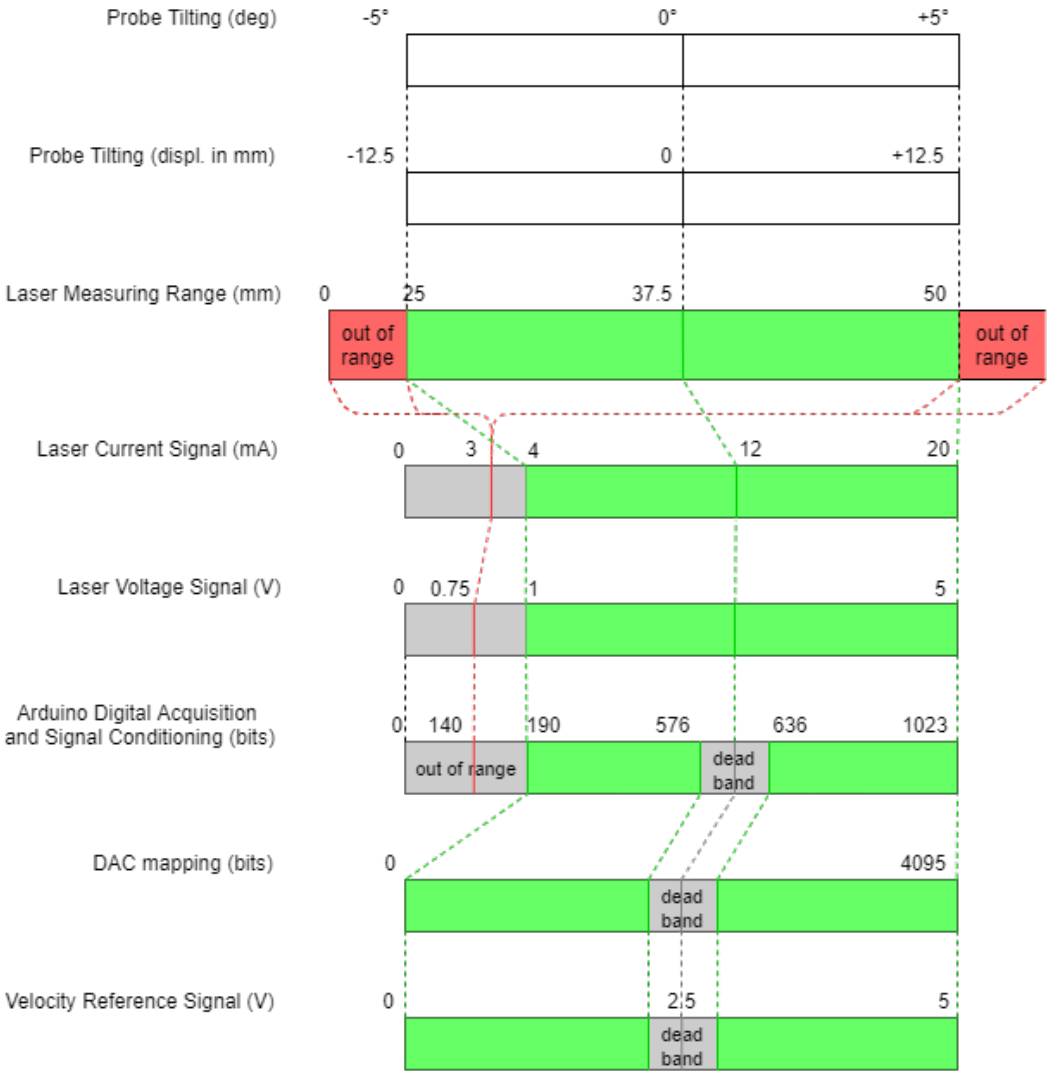

Figure 4.9: Conditioning process of the laser measurement

The motion is allowed until the ball bearing stays between the low and high endruns given by the screw edges. If the ball bearing goes out of the working range, the mechanism is forced and fault may occurs. A current protection is already implemented inside the driver, checking that the amount of demanded current doesn't exceed the fault threshold. To reduce faults occurrence a position thresholds are inserted, just before the screw mechanical endruns.

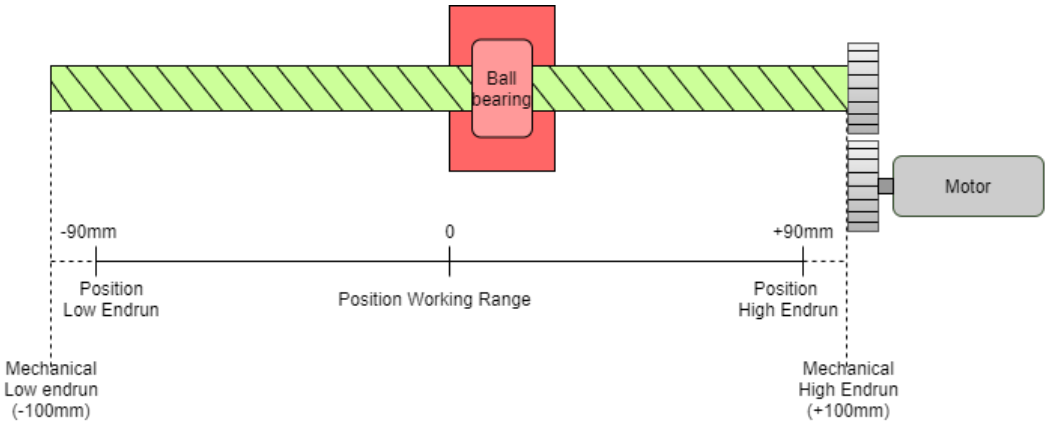

Figure 4.10: Ball screw endruns

The Arduino **reads the motor position** through Epos query. If the motor is out of the range, the velocity reference signal is not sent, the motor is blocked and the process return to acquire the laser measurement. If the actual position stays inside its range, the **velocity reference signal is sent** by Arduino to Epos.

When the "soft docking accomplished" signal is received, the motors are shut down and electrical brake are applied, avoiding undesired displacements during the next stages.

4.3.4 State 3: Precharge

In this state a partial retraction of the probe is performed. Having the female part hooked to the probe through soft docking, this operation pushes the cone element against the springed base, charging the springs. This state is useful for two reasons: firstly it permits to create a more solid connection between the two part respect to just the petals hooking, furthermore the springs compression allows to store energy to use during the undocking operation.

The motion is done configuring the probe motor applying a position control loop. The Arduino sending commands are the same of the full extension state, with the difference that the motor position reference sent is different.

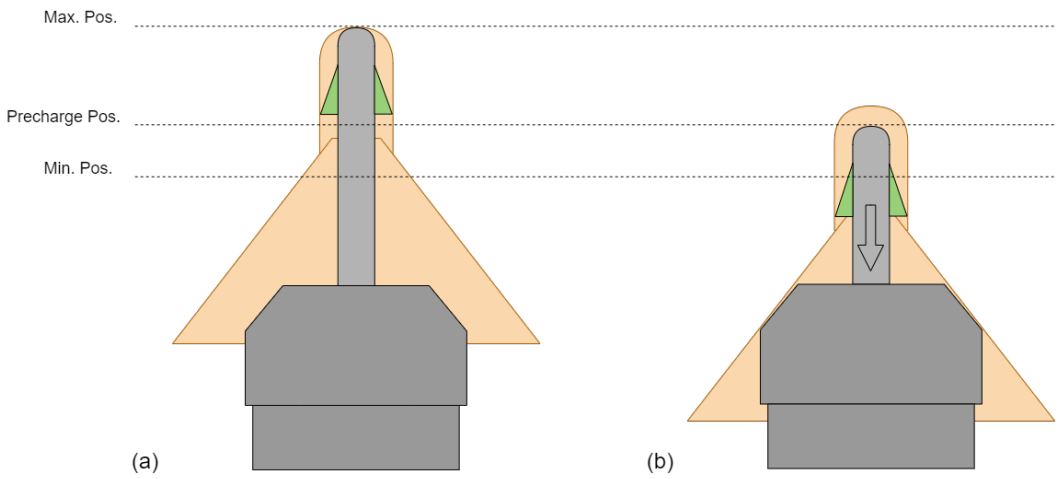

Figure 4.11: Precharge of the mechanism, from soft docking position (a) to precharge position (b)

Once the Arduino aknowledges that the desired position is reached, the state machine is ready to go ahead.

4.3.5 State 4: Homing

At the end of the alignment stages the probe is displaced from its initial central position. In State 4 the mechanism has to recover the probe at its initial central position and to drag the female cone aligning it with the hard docking mechanism. The Arduino board activates the driver of the X and Y axis motors and performs the following commands:

- the drivers are set in "profile position mode", acting a closed loop position control to the motors.
- the motion is performed sending the "Enable Device" command. Having relative encoders the starting position of the mechanism is memorized as zero position. It implies that the target position reference to send to Epos is zero.
- the Arduino asks to motor drivers if the target position is reached without errors. If the answer is positive, the electrical brakes are applied over the actuators, in order to avoid undesired displacements from zero position.

4.3.6 State 5: Full Retraction

In this state the probe is fully retracted and the female cone is placed in contact with the hard docking baseplate. Again, a position control over the probe motor is implemented, through the same Arduino commands used in stage 1 and 3. The zero position of the motor is used as reference position signal.

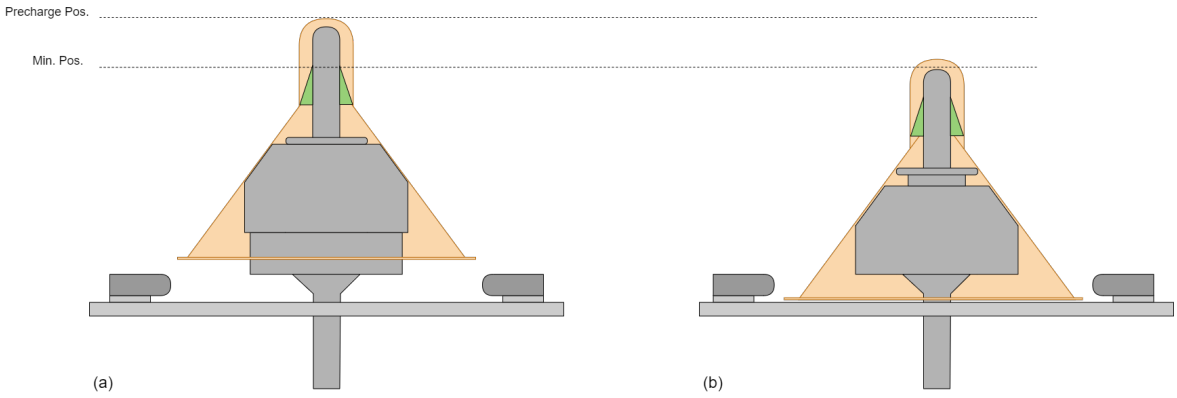

Figure 4.12: Full Retracion Stage, the probe is moved from precharge (a) to fully retracted position (b)

When the Arduino obtains as motor position actual value 0qc, the full retraction is complete and the motor is disabled.

4.3.7 State 6 and 7: Hard Docking and Shutdown

In the last docking stage the hard docking is performed. The Arduino board calls in action the stepper motor to move the hooks mechanism. The actuation's control is done by using three digital signals related to step, direction and enable motor [5]. The stepper driver was previously configured using the dedicated PC software (P700 Tools 2.11), where it was selected the motor type and the control parameters as follows:

Motor Type	DC CTM21xxx25
Step Resolution	1000 steps/rev
Enable Polarity	Low Active
Jog and Stop Parameters	
Accel/Decel	20 Revs/s^2
High Speed	2 Revs/s
Low Speed	0.5 Revs/s
Stop Rate	100 Revs/s^2

Table 4.3: stepper driver's parameters

In the following figure the digital levels and the timing of the three signals are shown:

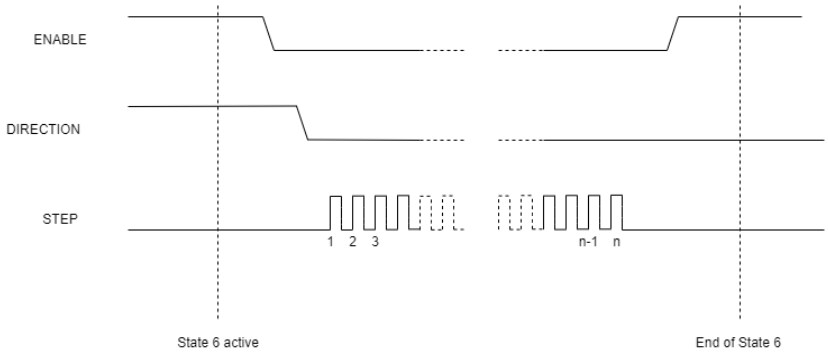

Figure 4.13: Timing of stepper digital signals

So, after entering the state the stepper is enabled and the correct direction of rotation is selected. Then the steps are sent by creating a square wave with duty cycle of 50%. The number of steps are calculated taking into account the step resolution and the fact that to have a complete closing of the hooks it needs a little more than a single revolution of the rotor. When the last step is sent the Enable signal is set to "1" and the stepper is turned off.

The last operation done by Arduino is to shutdown all the devices. in this stage the "shutdown" command is sent to each drivers through serial communication ports. After that the state machine is ended and the process returns to Idle state.

4.4 Code implementation

The Arduino code is written using the Arduino IDE [17]. The sketch is composed by four sections: the variable definitions, the setup function, the loop function and the control functions definitions.

In the **variable definitions** section are defined all the variable that must be used as global. The set data are:

- **Data Registers**, the data strings to send to Drivers in order to write or read a specific register. According to (communication appendix) they are strings of bytes that contains information about what kind of operation is done, the register's address to consider and the value to store inside it. All the strings are written in exadecimal.

- **Endrun Positions**, the relative endrun positions of the motor in qc. Here are stored the endruns of the alignment mechanism and the maximum extension of the probe. They are in exadecimal too.

- **Laser data**, the definitions necessary for DAC converters. Here the "Adafruit MCP4725" and "WIRE" libraries are included to the sketch and the DAC modules parameters are defined.

- **Pins and Internal variables**, the Arduino Input and Output pins used and the internal flags and variable used during the process.

```
//--- DATA REGISTERS ---
byte dataStartMotor[]=          {0x11,0x03,0x60,0x40,0x02,0x00,0x00,0
   x0F,0x00,0x00};
byte dataStopMotor[]=           {0x11,0x03,0x60,0x40,0x02,0x00,0x00,0
   x0B,0x00,0x00};
byte dataShutdown[]=            {0x11,0x03,0x60,0x40,0x02,0x00,0x00,0
   x06,0x00,0x00};

byte dataProfilePositionMode[]= {0x11,0x03,0x60,0x60,0x02,0x00,0x00,0
   x01,0x00,0x00};
byte dataStartMoving[]=         {0x11,0x03,0x60,0x40,0x02,0x00,0x00,0
   x3F,0x00,0x00};
byte posCompareEn[]=            {0x11,0x03,0x20,0x7A,0x02,0x01,0x00,0
   x41,0x00,0x00};

byte posActualValue_req[6]=     {0x10, 0x01, 0x60, 0x64, 0x02, 0x00};
byte actualPosition_X[4]=       {0x00, 0x00, 0x00, 0x00};
byte actualPosition_Y[4]=       {0x00, 0x00, 0x00, 0x00};
byte actualPosition_Probe[4]=   {0x00, 0x00, 0x00, 0x00};

//motors endruns positions in qc
byte LowEndRun_X[]=             {0xFF,0xFC,0xE5,0xB0};
byte HighEndRun_X[]=            {0x00,0x03,0x26,0x50};
byte LowEndRun_Y[]=             {0xFF,0xFD,0x80,0x00};
byte HighEndRun_Y[]=            {0x00,0x02,0x52,0x00};
byte MaxProbeExt[]=             {0xFF,0xAD,0x9A,0x00};

//--- DEFINITIONS FOR THE LASERS ---
#include <Adafruit_MCP4725.h>
#include <Wire.h>

Adafruit_MCP4725 dac_1;
Adafruit_MCP4725 dac_2;
int  valDac_1 =0;
int  valDac_2 =0;
int  valSensor_1=0;
int  valSensor_2=0;
int acq_time=2000;
int jj=1;
//--- ENABLE SIGNALS FOR EPOS DRIVERS ---
int EposEnable_1=10;  //X axis
int EposEnable_2=12;  //Y axis
//--- DIGITAL PINS FOR STEPPER CONTROL ---
int Enable_Pin=51;
```

```
int Step_Pin=52;
int Direction_Pin=53;
//--- INTERNAL VARIABLES AND FLAGS ---
char check='0';                //variable for reading commands from
   terminal
volatile int allignm_ok = 0;  //interrupt flag for allignment
volatile bool EM = LOW;        //interrupt flag for emergency stop
uint8_t state=0;               //state variable for state machine
```

In the **Setup Function** stays the code that must be executed once at the beginning of the process. The the serial ports are initialized, setting the baudrate, the GPIO pins, and the DAC modules. Also the interrupt signal pins are defined.

```
    void setup() {
 //serial port inizialization
 Serial.begin(115200);
 Serial1.begin(115200);
 Serial2.begin(115200);
 Serial3.begin(115200);
 pinMode(13, OUTPUT);
 //pin stepper setting
 pinMode(Enable_Pin, OUTPUT);
 pinMode(Direction_Pin, OUTPUT);
 pinMode(Step_Pin, OUTPUT);

 digitalWrite(Enable_Pin, HIGH); //stepper disabled

 pinMode(EposEnable_1, OUTPUT);
 pinMode(EposEnable_2, OUTPUT);
 //interrupt for alignment complete signal
 attachInterrupt(digitalPinToInterrupt(2), allign_status, FALLING);
 //interrupt for emergency Stop
 attachInterrupt(digitalPinToInterrupt(3), emergencyStop, FALLING);

 //inizializzation of DAC modules
  dac_1.begin(0x60);
  dac_2.begin(0x61);
  dac_1.setVoltage(1, false);
  dac_2.setVoltage(1, false);
 }

```

Inside the **Loop Function** there is the core of the process that has to run repeatedly. A serial monitor check procedure is inserted at the beginning of the loop. This serves to control the state machine flow through serial terminal and to pass from a state to the next one. If this code is uncommented the process is controlled by PC. This is usefull during testing. If the code is commented the sketch can run independently onto

Arduino board.

```
1  void loop(){
2
3   while(!Serial.available());
4   Serial.println(F("Enter any key"));
5   while (Serial.available() == 0);
6   while (Serial.available() != 0){
7          Serial.read();
8   }
```

After the state machine is implemented. A Switch-Case structure is used, where the switch variable is an integer representing the state number. Note that, before the SWITCH structure an IF statement is inserting. It verifies if the Emergency button is pressed, if yes the actual state to run is set to 8, which means the shutdown state will be executed independently at which stage the state machine is.

```
1  if(EM==HIGH)   //if emergency stop button is pressed the state
   machine is forced to
2   state=8;    //shutdown the motors ('shutdown state')
3
4 switch(state){
5
6   case 0:   //--IDLE--
7   Serial.println(F("idle state, press G to continue"));
8   while (Serial.available() == 0);      //waiting for start command
9   check=Serial.read();
10  if(check=='G')
11    state=1;
12  break;
13
14  case 1:   //-- SETUP OF THE MOTORS --
15  Serial.println(F("state 1 active"));
16  setupEpos(1);
17  setupEpos(2);           //setup di tutti i motori
18  setupEpos(3);
19  Serial.println(F("all Epos set up"));
20  state=2;
21
22  break;
23
24  case 2:   //--- FULL EXTENSION ---
25  Serial.println(F("state 2 active"));
26  moveMotor('C', 3);      //max. extension motion
27  moveMotor('D', 3);      //petals opening
28  state=3;
29  break;
30
```

```
31      case 3:   //-- ALLIGNMENT --
32      Serial.println(F("state 3 active"));
33      //driver x and y set in velocity mode
34      SetAllignment(1);
35      SetAllignment(2);
36
37      dac_1.setVoltage(2048,false);
38      dac_2.setVoltage(2025,false);
39
40      sendMessageToEPOS(dataStartMotor,sizeof(dataStartMotor), 1);
41      delay(1);
42      sendMessageToEPOS(dataStartMotor,sizeof(dataStartMotor), 2);
43      delay(1);
44      digitalWrite(EposEnable_1,HIGH);
45      digitalWrite(EposEnable_2,HIGH);
46
47      while(allignm_ok==0 && EM==LOW){
48         //acquiring laser signals
49         valSensor_1 = analogRead(A0) ;
50         valSensor_2 = analogRead(A3) ;
51         //acquiring actual position of the motors
52         readRegister(posActualValue_req, 1, actualPosition_X );
53         /*for(int i=0; i<sizeof(actualPosition_X); i++){
54           Serial.println(actualPosition_X[i], HEX);
55           }*/    //uncomment to read actual pos. from terminal
56         readRegister(posActualValue_req, 2, actualPosition_Y );
57          /*for(int i=0; i<sizeof(actualPosition_Y); i++){       //
    uncomment to read actual pos. from terminal
58           Serial.println(actualPosition_Y[i], HEX);
59           }*/
60
61           //X Motor
62
63         //check of low endrun
64         if((actualPosition_X[0] == LowEndRun_X[0] && actualPosition_X
    [1] <= LowEndRun_X[1] && actualPosition_X[2] <= LowEndRun_X[2]) &&
     valSensor_1>600){
65               digitalWrite(EposEnable_1,HIGH);
66               }
67         //check of high endrun
68         else if((actualPosition_X[0] == HighEndRun_X[0] &&
    actualPosition_X[1] >= HighEndRun_X[1] && actualPosition_X[2] >=
    HighEndRun_X[2]) && valSensor_1<600){
69               digitalWrite(EposEnable_1,HIGH);
70               }
71         //check of dead band and out of range
72         else if (valSensor_1<190 || (valSensor_1>576 && valSensor_1
```

```
<636)){
73          digitalWrite(EposEnable_1,HIGH);
74          }
75          else{
76                digitalWrite(EposEnable_1,LOW);
77                //signal mapping to send with DAC, digital input from
     0 to 4095
78                valSensor_1=map(valSensor_1, 190, 1023, 4095, 0);
79                dac_1.setVoltage(valSensor_1,false);
80               }
81
82        // Y Motor
83
84        if((actualPosition_Y[0] == LowEndRun_Y[0] && actualPosition_Y
    [1] <= LowEndRun_Y[1] && actualPosition_Y[2] <= LowEndRun_Y[2]) &&
     valSensor_2<600){
85              digitalWrite(EposEnable_2,HIGH);
86              }
87        else if((actualPosition_Y[0] == HighEndRun_Y[0] &&
    actualPosition_Y[1] >= HighEndRun_Y[1] && actualPosition_Y[2] >=
    HighEndRun_Y[2]) && valSensor_2>600){
88              digitalWrite(EposEnable_2,HIGH);
89              }
90
91        else if (valSensor_2<190 || (valSensor_2>576 && valSensor_2
    <636)) {
92            digitalWrite(EposEnable_2,HIGH);
93            }
94            else {
95                 digitalWrite(EposEnable_2,LOW);   //low means motor
    active
96                 valSensor_2=map(valSensor_2, 190, 1023, 0, 4095);
97                 dac_2.setVoltage(valSensor_2,false);
98                 }
99     }
100    digitalWrite(EposEnable_1,LOW);
101    digitalWrite(EposEnable_2,LOW);
102    sendMessageToEPOS(dataShutdown,10, 1);
103    sendMessageToEPOS(dataShutdown,10, 2);
104    Serial.println(F("soft docking completed"));
105    state=4;
106    break;
107
108    case 4:   //-- PRELOAD --
109
110    Serial.println(F("state 4 active"));
111    moveMotor('B', 3);
```

```
112     state=5;
113     break;
114 
115     case 5:   //-- HOMING --
116 
117     Serial.println(F("state 5 active"));
118     digitalWrite(EposEnable_1,LOW);
119     moveMotor('0', 1);
120     digitalWrite(EposEnable_2,LOW);
121     moveMotor('0', 2);
122     sendMessageToEPOS(dataStartMotor,sizeof(dataStartMotor), 1);  //
    arm the motore, inserting a brake
123     delay(1);
124     sendMessageToEPOS(dataStartMotor,sizeof(dataStartMotor), 2);
125     Serial.println(F("home position reached"));
126     state=6;
127     break;
128 
129     case 6:   //-- RETRACTION --
130     Serial.println(F("state 6 active"));
131     moveMotor('A', 3);
132     Serial.println(F("full retraction completed"));
133     state=7;
134     break;
135 
136     case 7:   //-- HARD DOCKING --
137     Serial.println(F("state 7 active"));
138     moveStepper();
139     Serial.println(F("hard docking completed"));
140     state=8;
141     break;
142 
143     case 8:   //--SHUTDOWN-- stato per arrestare tutti i motori
144     Serial.println(F("state 8 active"));
145     sendMessageToEPOS(dataShutdown,sizeof(dataShutdown), 1);
146     delay(1);
147     sendMessageToEPOS(dataShutdown,sizeof(dataShutdown), 2);
148     delay(1);
149     sendMessageToEPOS(dataShutdown,sizeof(dataShutdown), 3);
150     digitalWrite(EposEnable_1,HIGH);
151     digitalWrite(EposEnable_2,HIGH);
152     digitalWrite(Enable_Pin,HIGH);
153     Serial.println(F("shutdown completed"));
154     state=0;
155     break;
156   }
157     }
```

In the last code section there is the definition of the functions used during the process. They can be divided in two categories: the communication functions and the operational functions.

The **communication functions** are the ones that are used to implement the communication protocol between Arduino and Epos controllers (see Appendix A.3).

- **sendMessageToEPOS**: it is the function used to send a request to write a register of the Epos controller. It receives as input a string of bytes, containing the register's address and the data register to write, an integer that represents the length of the string in bytes, and an integer that defines the controller number to send the command.

```
1    void sendMessageToEPOS(byte data[], int lengths, int index){
2     byte opcode =      data[0];
3     uint16_t crcCode = CRC_XModem(data, lengths);
4     byte dataToSend[lengths + 2];
5     dataToSend[0] =    data[0];
6     dataToSend[1] =    data[1];
7     for (int i = 2; i < lengths; i++){    //Transmit order is
   different from oringial,a word's high byte is transmit after
   low byte
8       if (i % 2 == 0)    //Replace the position of each high
   and low byte in one word.
9         dataToSend[i] = data[i + 1];
10      else
11        dataToSend[i] = data[i - 1];
12    }
13    uint8_t highByteCrc = (crcCode >> 8) & 0x00FF;
14    uint8_t lowByteCrc =  crcCode & 0x00FF;   //Get crc code
15    dataToSend[lengths] = lowByteCrc;
16    dataToSend[lengths + 1] = highByteCrc;
17
18    #ifdef DEBUG
19    debug_println(F("Origin data:"));
20    for (int i = 0; i < lengths; i++){
21       debug_print_int(data[i]);
22       debug_println(" ");
23    }
24    debug_println();
25    debug_print(F("CRCCode:"));
26    debug_print_int(crcCode);
27   #endif
28    char datatemp;
29   flag1:;
30
31   SerialSend(opcode, index);          //Firstly,sending the
```

```
    opcode
32    debug_println(F("Write opcode ,waiting for response"));
33    long int now = millis();
34    while (SerialAvailable(index) == 0){
35         if (millis() - now > 50)      //Go out of the loop in
    case of error
36           goto flag1;
37    }   //wait for  response
38    debug_print(F("Send opcode,Recv:"));
39    while (SerialAvailable(index) != 0){
40         datatemp = SerialRead(index);
41         delay(1);
42         debug_print(F(","));
43    }
44    debug_println(F(""));
45    if (datatemp == '0'){               //Response correct, send
    other data
46      debug_println(F("Response correct,continue!\nSend data"));
47    for (int i = 1; i < lengths + 2; i++)
48      SerialSend(dataToSend[i], index);     //Sending all the
    data
49    debug_println(F("Finish sending other data now wait for
    response..."));
50
51    while (SerialAvailable(index) == 0);    //wait for another
    respnse
52         debug_print(F("Receive data:"));
53         debug_println(F("Finish sending"));
54    }
55    else{
56        debug_println(F("First response is not okay"));
57        goto flag1;
58    }
59}
60
```

- **readRegister**: it is the function used to send a request to read a register of the Epos controller. It receives as input a string of bytes, containing the address of the register to read, an integer that represents the length of the string in bytes, and an integer that defines the controller number to send the command.

```
1    void readRegister( byte Data[], int serialnum, byte pdata[])
     //Data is the command, serialnum is the motor number, pdata
    the received data you want to read
2{
3  byte dataRead[]=        {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
    x00, 0x00, 0x00, 0x00, 0x00};
```

```
4  byte inputData[] =      {Data[0], Data[1], Data[2],Data[3],
    Data[4], Data[5]};
5
6  flag2:;
7  Serial.println("trying to read actual position: ");
8
9  debug_println("sending position request: ");
10 sendMessageToEPOS(inputData, sizeof(inputData), serialnum);  //
    sending reading request
11
12  byte tx_req;
13  while (SerialAvailable(serialnum) != 0)
14  {
15   tx_req = SerialRead(serialnum);
16   debug_print("tx_req:  ");
17   debug_print_int(tx_req);
18   debug_println("");
19   delay(1);
20
21   debug_println("aspe ");
22  }
23  if(tx_req == 0x00)                    //verifica se Epos
    risponde "0x00"
24  {
25   debug_println("ricevuta risposta 0x00");
26   SerialSend(0x4F, serialnum);
27  }
28 else
29  {
30   debug_println("reading failed, try again");
31   goto flag2;
32  }
33
34 long int now = millis();
35 while (SerialAvailable(serialnum) == 0)
36 {
37   if (millis() - now > 50)    //Go out of the loop in case of
    error
38      goto flag2;
39 }   //wait for  response
40 debug_print("receiving data.. ");
41
42 int i=0;                          //receiving bytes
43 while (SerialAvailable(serialnum) != 0)
44 {
45   debug_println("byte received! ");
46   dataRead[i] = SerialRead(serialnum);
```

```
47      i++;
48      delay(1);
49    }
50    if(SerialAvailable(serialnum)== 0)
51    {
52
53    for(int i=0;i<4;i++)                    //save data to read
54    {
55    pdata[i]=dataRead[8-i];
56    }
57
58   #ifdef DEBUG
59  for(int i=0; i<sizeof(pdata);i++)      //print data to read
60    {
61     Serial.print(pdata[i], HEX);
62     Serial.print(" ");
63    }
64    #endif
65    }
66    }
67
```

- **CRC_XModem**: the function used to create the CRC code, which is used by Epos driver to check the correctness of communication.

```
1     uint16_t CRC_XModem(byte bytes[], int lengths) {
2
3         uint16_t crc = 0x00;          // initial value
4         uint16_t polynomial = 0x1021;
5         for (int index = 0 ; index < lengths; index++) {
6            byte b = bytes[index];
7            for (int i = 0; i < 8; i++) {
8                boolean bits = ((b   >> (7 - i) & 1) == 1);
9                boolean c15 = ((crc >> 15    & 1) == 1);
10               crc <<= 1;
11               if (c15 ^ bits)
12                  crc ^= polynomial;
13           }
14        }
15        crc &= 0xffff;
16
17   return crc;
18  }
19
```

The **operational functions** is the ones that performs the actions to apply over the motors.

- **setupEpos**: clean all faults and resets the selected driver. As input it received an integer indicating the driver to set up.

```
    void setupEpos(int serialnum){

 byte dataCleanExecMask[]=    {0x11,0x03,0x20,0x7D,0x02,0x00,0x00
   ,0x00,0x00,0x00};
 byte dataSetVelocity[]=        {0x11,0x03,0x60,0x81,0x02,0x00,0
   x1F,0x40,0x00,0x00};
 byte dataCleanPosition[]=    {0x11,0x03,0x20,0x62,0x02,0x00,0x00
   ,0x00,0x00,0x00};
 byte dataCleanTarget[]=      {0x11,0x03,0x60,0x7A,0x02,0x00,0x00
   ,0x00,0x00,0x00};
 byte dataClearFault[]=         {0x11,0x03,0x60,0x40,0x02,0x00,0
   x00,0x80,0x00,0x00};

  Serial.println(F("cleaning the epos"));
  sendMessageToEPOS(dataClearFault,10, serialnum);
  delay(1);
  sendMessageToEPOS(dataShutdown,10, serialnum);
  sendMessageToEPOS(dataCleanExecMask,10, serialnum);
  delay(1);
  sendMessageToEPOS(dataCleanPosition,10, serialnum);
  if(serialnum==3){
    sendMessageToEPOS(dataSetVelocity,10, serialnum);
  }
  delay(1);
  sendMessageToEPOS(dataCleanTarget,10, serialnum);
  delay(1000);
  Serial.println(F("registers cleaned"));
}

```

- **moveMotor**: function to move a motor using the profile position mode. It sends to selected device the target position, enables the motion, checks if the desired position is reached and finally disable the motion.

```
    void  moveMotor(char pos, int serialnum){

byte targetPos[10]=          {0x11,0x03,0x60,0x7A,0x02,0x00,0x00,0
   x00,0x00,0x00};

if(serialnum==3){
  if(pos=='C'){
  targetPos[6]=0xA9;    // max extension position
  targetPos[7]=0xDD;
  targetPos[8]=0xFF;
  targetPos[9]=0xA5;
```

```
  }
  else if(pos=='B'){    //partial retraction
  targetPos[6]=0xB0;
  targetPos[7]=0x00;
  targetPos[8]=0xFF;
  targetPos[9]=0xC0;
  }
  else if(pos=='A'){    //full retraction
  targetPos[6]=0x00;
  targetPos[7]=0x00;
  targetPos[8]=0x00;
  targetPos[9]=0x00;
  }
  else if(pos=='D'){
  targetPos[6]=0x5D;
  targetPos[7]=0x8D;
  targetPos[8]=0xFF;
  targetPos[9]=0xAF;    // petal open position
  }
}
else {
  targetPos[6]=0x00;
  targetPos[7]=0x00;
  targetPos[8]=0x00;
  targetPos[9]=0x00;
}

readRegister(posActualValue_req, serialnum, actualPosition_Probe
   );

    sendMessageToEPOS(dataProfilePositionMode,sizeof(
   dataProfilePositionMode), serialnum);
    delay(1);
    sendMessageToEPOS(dataStartMotor,sizeof(dataStartMotor),
   serialnum);
    delay(1);
    sendMessageToEPOS(targetPos, sizeof(targetPos), serialnum);
    delay(1);
    sendMessageToEPOS(dataStartMoving,sizeof(dataStartMoving),
   serialnum);

    while((actualPosition_Probe[2]!=targetPos[6] ||
   actualPosition_Probe[3]!=targetPos[7] || actualPosition_Probe
   [0]!=targetPos[8] || actualPosition_Probe[1]!=targetPos[9]) &&
    EM==LOW){

        readRegister(posActualValue_req, serialnum,
```

```
    actualPosition_Probe );
51          debug_println(F("leggo   "));
52          for(int i=0; i<sizeof(actualPosition_Probe); i++){
53             Serial.println(actualPosition_Probe[i], HEX);
54             }
55          }
56     sendMessageToEPOS(dataShutdown,sizeof(dataShutdown),
    serialnum);
57     Serial.println(F("position reached"));
58 }
59
```

- **moveStepper**: function to move the stepper motor. It generates the required signals and sends them to stepper controller.

```
1     void moveStepper(){
2     digitalWrite(Direction_Pin, LOW);  //direction selected
3     digitalWrite(Enable_Pin, LOW);    //stepper enabled
4
5          for (int i=0; i<=1050; i++){
6           if (EM==HIGH)
7             i=1050;
8           digitalWrite(Step_Pin, HIGH);
9           delay(10);
10          digitalWrite(Step_Pin, LOW);
11          delay(10);
12          debug_println(F("step sent"));
13         }
14    digitalWrite(Enable_Pin, HIGH);    //stepper disabled
15    Serial.println(F("hooks closed correctly"));
16 }
17
```

- **SetAlignment**: function to set the X and Y axes motors drivers to work with velocity control mode. Here the control parameters and the programmable GPIO pins are set.

```
1      void SetAllignment(int serialnum) {
2
3   byte dataVelocityMode[]=     {0x11,0x03,0x60,0x60,0x02,0x00,0
     x00,0xFE,0x00,0x00};
4   byte dataMaxSpeed[]=         {0x11,0x03,0x60,0x7f,0x02,0x00,0
     x61,0xA8,0x00,0x00};
5   byte dataInputConfig[]=      {0x11,0x03,0x20,0x7B,0x02,0x01,0
     x00,0x01,0x00,0x00};
6   byte dataExecMask[]=         {0x11,0x03,0x20,0x7D,0x02,0x00,0
     x00,0x02,0x00,0x00};
```

```
7
8   //--- Input Parameter for Analog Input---
9   byte dataSetpoint_Scaling[]=  {0x11,0x03,0x23,0x02,0x02,0x01,0
     x13,0x88,0x00,0x00};   //rpm/volt
10  byte dataSetpoint_Offset[]=   {0x11,0x03,0x23,0x02,0x02,0x02,0
     xCF,0x2B,0xFF,0xFF};   // Negative Offset , -12500, offset
     =-2.5*scaling
11    //--- Configuration Digital Input---
12    byte dataConf_Digital_Input[]=         {0x11,0x03,0x20,0x70,0
     x02,0x03,0x00,0x05,0x00,0x00};   // Configuration digital
     input 3 as Quickstop
13    byte dataDigital_Input_Exec_Mask[]=    {0x11,0x03,0x20,0x71,0
     x02,0x04,0x00,0x20,0x00,0x00};   // Esecution Mask for Quick
     stop and position Marker
14
15
16
17     sendMessageToEPOS(dataShutdown,10,serialnum);
18     sendMessageToEPOS(dataVelocityMode,10, serialnum);
19     sendMessageToEPOS(dataMaxSpeed,10, serialnum);
20     sendMessageToEPOS(dataInputConfig,10, serialnum);
21     sendMessageToEPOS(dataSetpoint_Scaling,sizeof(
    dataSetpoint_Scaling),serialnum);
22     sendMessageToEPOS(dataSetpoint_Offset,sizeof(
    dataSetpoint_Offset),serialnum);
23     sendMessageToEPOS(dataExecMask,10, serialnum);
24     sendMessageToEPOS(dataConf_Digital_Input,10,serialnum);
25     sendMessageToEPOS(dataDigital_Input_Exec_Mask,10,serialnum);
26     sendMessageToEPOS(dataShutdown,10, serialnum);
27     delay(50);
28    Serial.println(F("Set allignment ended"));
29  }
30
31
```

- **ISR**: they are the Interrupt Service Routine, used in combination with two push-buttons: one for the Emergency Stop command and one for the alignment occurred command.

```
1       //--- ISR for interrupt 'alignment ok' ---
2   void allign_status(){
3     allignm_ok=1;
4     detachInterrupt(digitalPinToInterrupt(2));
5   }
6
7   //--- ISR for interrupt 'Emergency stop' ---
8   void emergencyStop(){
```

```
  EM=HIGH;
  digitalWrite(Enable_Pin, HIGH);
  digitalWrite(13, HIGH);
}

```

Chapter 5

Experimental tests

5.1 Benchmark test

The first trial done consists of a benchmark test for the electrical elements. They are disassembled from the mechanism and positioned over a testing board, built for the purpose, such that the motors can move freely without an applied inertial load. All the cables connections are wired and the power supply is provided through a variable bench power supply [8]. In figure 5.2 are presented the developed test environment, where they can be seen, starting on the left, the actuators, the drivers, the Arduino Mega board with its additional hardware and the lasers sensors. On the bottom left corner of the picture there is the wiring supply panel, connected to the power supply.

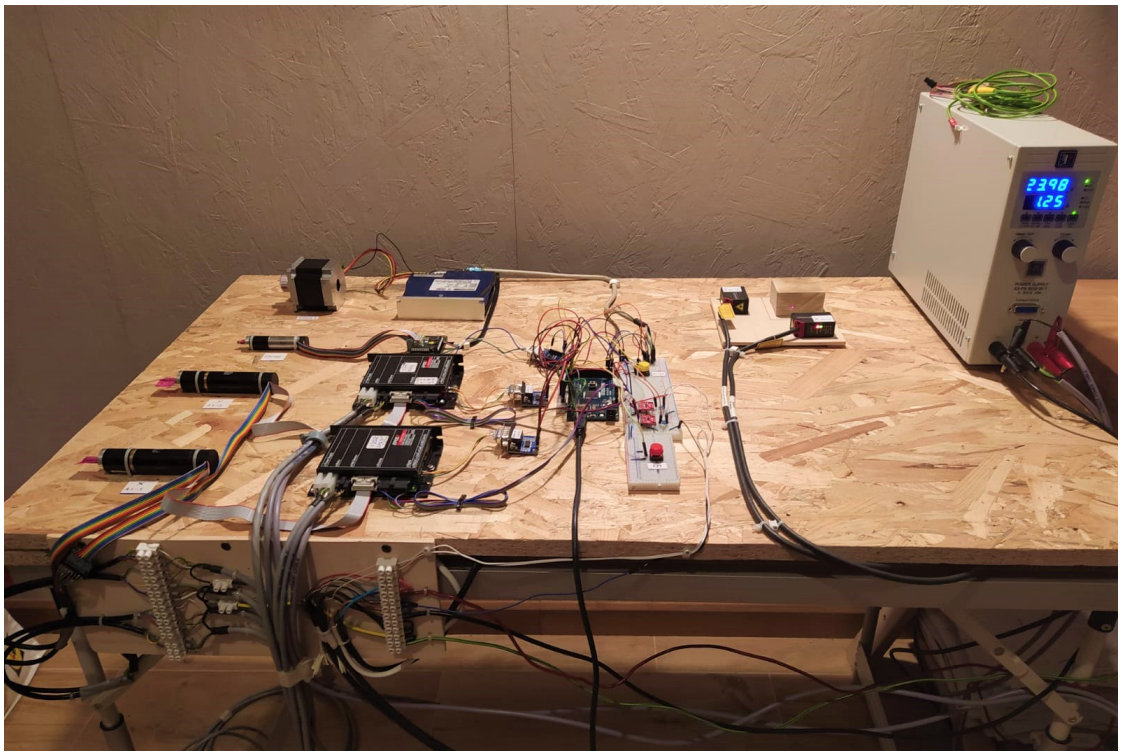

Figure 5.1: Picture of the developed testbench

This test has the following purposes: for first it is useful to check if all the devices work correctly and if the communication protocol is effective without faults, debugging the Arduino code. Also the test permits to see if it needs to make changes over the control architecture or if additional hardware is necessary.

The testbench trial was conducted in this way:

1. the power supply cables of motor drivers and of the sensors are connected to power supplier, that is set to provide a DC Voltage of 24V, with a maximum current threshold of 2A (calculated referring to datasheets and set in order to avoid low current supply faults)

2. the Arduino board is connected to PC through USB cable. The algorithm code is uploaded using the Arduino IDE. The USB connection provides both serial communication for debug commands from terminal and power supply for the board. The sketch is the one discussed in chapter **4.4**, with the difference that all the velocity setting values are reduced. This because high speed motions of the motors in no load conditions may damage them.

3. The control algorithm is started. At the beginning of each docking stage a serial command from keyboard is requested to go ahead. At every docking stage it is checked if the correct driver and motors are configured correctly and if the expected rotor motions are performed.

4. when the last stage is performed and the state machine returns in idle state the test ends

After, a set of test executions the following features are added to the system:

- a control panel is added. It consists of two normally open push buttons, one for the Emergency stop and the other for the alignment completed commands. The first permits to stop the docking execution at any time and shutdown the motors. The second push button is used during the "alignment and soft docking" state to notify that the soft docking is performed. This two push button are connected to Arduino such that once they are pressed the 5V logic level is connected to the respective Arduino input pins. These pins are attached to two interrupts and to respective ISR.

- two more digital pins are configured, one for each Epos2 24/5. They are used by Arduino board to send the "stop motor" command without using the rs232 protocol. During the tests it has been noticed that when the velocity control mode is applied to the motors, the stopping of the motion is not immediate and the serial communication causes delays. The use of a dedicated digital command signal permits to stop the motion faster and avoid undesired movements.

The schematics in figure 5.2 shows the schematic of the connections of the Arduino Mega with the additional hardware mounted on a breadboard:

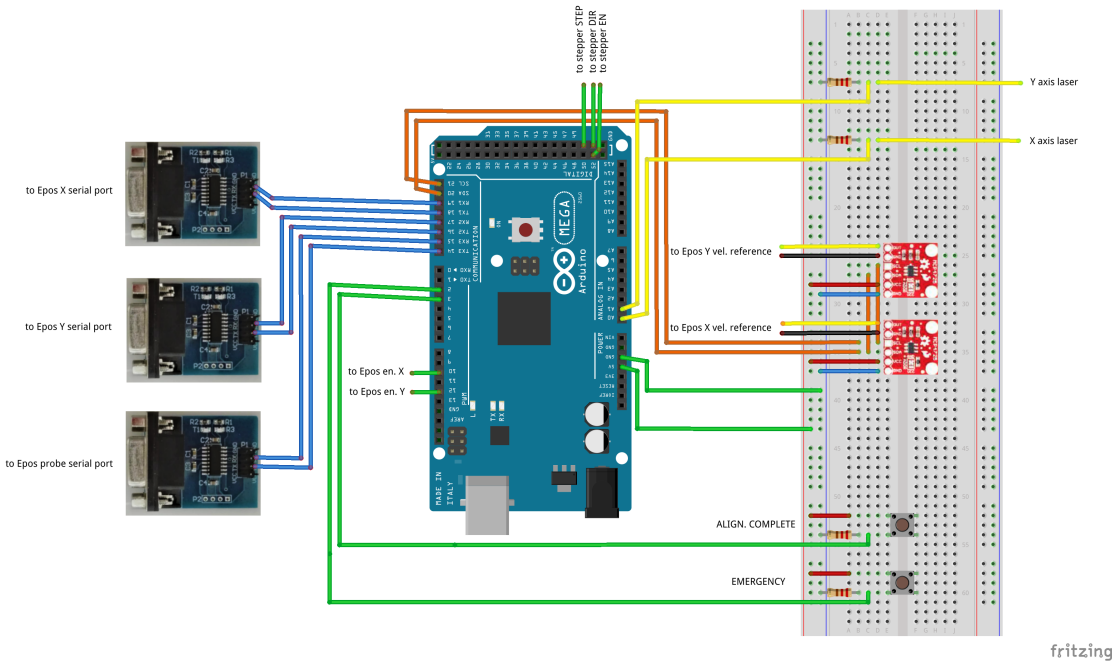

Figure 5.2: Arduino board and the additional hardware schematics

5.2 test over the docking mechanism

This benchmark test is done to verify the control system behaviour with the electrical hardware mounted over the male mechanism to control. The main purposes of this operation are the following:

- test the mechanical movements
- select the driver parameters that regulate the motors motion (such that the control gains, the maximal profile velocities, accelerations etc.)
- define and study the power consumption in steady state and in execution of the system
- define the execution time for each docking stage

Before the execution the mechanism must be prepared. The actuators and the sensors are placed and screwed on their accommodations. The mechanism is positioned at its initial position, with the probe at the fully retracted position and the rail slides at the central zero position. The Arduino board and its external hardware are connected as the benchmark test previously discussed and the power supply is provided to all the devices. Before the stages execution it is necessary to run the regulation tuning

through the Epos Studio software. This tool permits to tune automatically the PID controller parameters of the control loops and to save it on the motor's drivers.

Again the docking stages are performed one at a time, using the PC keyboard command and the Arduino serial terminal to skip from a state to the next one. The following results are obtained:

- the control parameters are set and the relative positions of the probe and the sliders are defined. In tables 5.1 and 5.2 these parameters are reported.

Control Parameters	
Epos2 24/5	
Max. Profile Velocity [rpm]	11000
Profile type in pos. Mode	trapezoidal
Max. Following Error [qc]	1000
Setpoint Scaling Factor [rpm/V]	2500
Setpoint Offset [rpm]	-6250
Epos2 24/2	
Max. Profile Velocity [rpm]	12000
Profile type in pos. Mode	trapezoidal
Max. Following Error [qc]	2000
QuickStop Deceleration (rpm/s)	10000

Table 5.1: Epos Control Parameters

Relative Positions	
Probe Positions [qc]	
Full Retraction	0
Precharge	-4149248
Petals Opening	-5284467
Full Extension	-5920291
Sliders Positions [qc]	
Home position	0
High Endrun X Axis	+206416
Low Endrun X Axis	-203344
High Endrun Y Axis	+152064
Low Endrun Y Axis	-163840

Table 5.2: Relative Motors positions

The velocity and acceleration parameters are selected in order to avoid sudden movements, that may causes peak currents and mechanical failures. In particular

in the "alignment and soft docking" phase the control parameters are set such that the velocity control loop was as responsive as possible and the maximum normal force applied on the probe edge never exceed the value of 10N. If this constraint is not respected an hypothetical soft docking on orbit with two satellites with our specifications fails.

- A study of the power consumption of the system is performed. The Epos controllers, the stepper driver and the laser sensors are powered by the use of a variable power supplier. All the devices can work with a supply voltage of 24V. The power consumption is obtained inspecting the absorbed current by each devices. The measurements are collected in the worst case conditions, with the maximum loads that the motors have to mode during the mating operations.

Absorbed current [mA]		
Device	Stand by State	Operating State
Epos2 24/5	40	90 in pos. Mode 180 in vel. Mode (Max)
Epos2 24/2	20	80
P7000 Stepper	40	800
uEpsilon Sensor	35	40

Table 5.3: Absorbed devices currents when they are in stand by mode and when enabled

The Arduino Mega board uses a separate supply, given by USB connection. It also provides the supply power to the additional elements (max3232 shields, DAC converters, pushbuttons) through its dedicated output voltage pins. The DAC converters use a 3.3V DC voltage, while DAC modules and pushbuttons a voltage of 5V. To measure how much current the Arduino modules need it is supplied with the laboratory power supply through the dedicated voltage input pin and it is inspected the absorbed current displayed on the power supplier. The maximum current value reached during the operation is 200mA.

From the measurements stated before it is possible to calculate the power consumptions through the relation:

$$P[W] = V[V] * I[A]$$

Max. Power Consumption [W]		
Device	Stand by State	Operating State
Epos2 24/5	0.96	2.16 in pos. Mode 4.32 in vel. Mode (Max)
Epos2 24/2	0.48	1.92
P7000 Stepper	0.96	19.2
uEpsilon Sensor	0.84	0.96
Arduino Modules		1

Table 5.4: Power consumption of each devices

To identify the worst case and therefore the maximum power needed, the single docking stages are examined and for each of them the absorbed power is calculated, considering what are the devices that are enabled in that stage:

Stage	**Operating Devices**	**Total Power Cons. [W]**
Full Extension Precharge Full Retraction	Epos2 24/2 Arduino Modules	7.48
alignment and Soft Docking	Epos2 24/5 (x2) Arduino Modules uEpsilon Sensors (x2)	13
Homing	Epos2 24/5 (x2) Arduino Modules	8.44
Hard Docking	P7000 Stepper Arduino Modules	24.28

Table 5.5: Power consumption at each stage's execution

- The last study over the mechanism concerns the execution time. It is measured the time required to perform each docking stage, deriving an estimation of the total execution time. In the case of stage 2 the timing cannot be determined a priori, because it depends on the relative distance and velocities of the two spacecrafts.

Execution Time of Docking Phases	
Stage 1 (Full Extension)	
Extension	85s
Petals Opening	10s
TOTAL	95s
Stage 2 (Alignment and Soft Docking)	
Drivers Config.	10s
Alignment	–
TOTAL	10s + alignment time
Stage 3 (Precharge)	
TOTAL	18s
Stage 4 (Homing)	
X Axis Motion	25s (Max.)
Y Axis Motion	20s (Max.)
TOTAL	45s
Stage 5 (Full Retraction)	
TOTAL	61s
Stage 6 (Hard Docking)	
TOTAL	22s

Total Docking Execution Time	251s + alignment time

5.3 Final test introduction

The last experimental test to perform on the mechanism consist of the execution of the mating approach involving the use of a robotic manipulator. For this purpose a dedicated test environment was designed. It consist of a cage structure, composed by aluminium profiles, where the male half part is bolted at the top in vertical position. Below it the manipulator is posed with the female passive part connected to its end-effector joint 5.3.

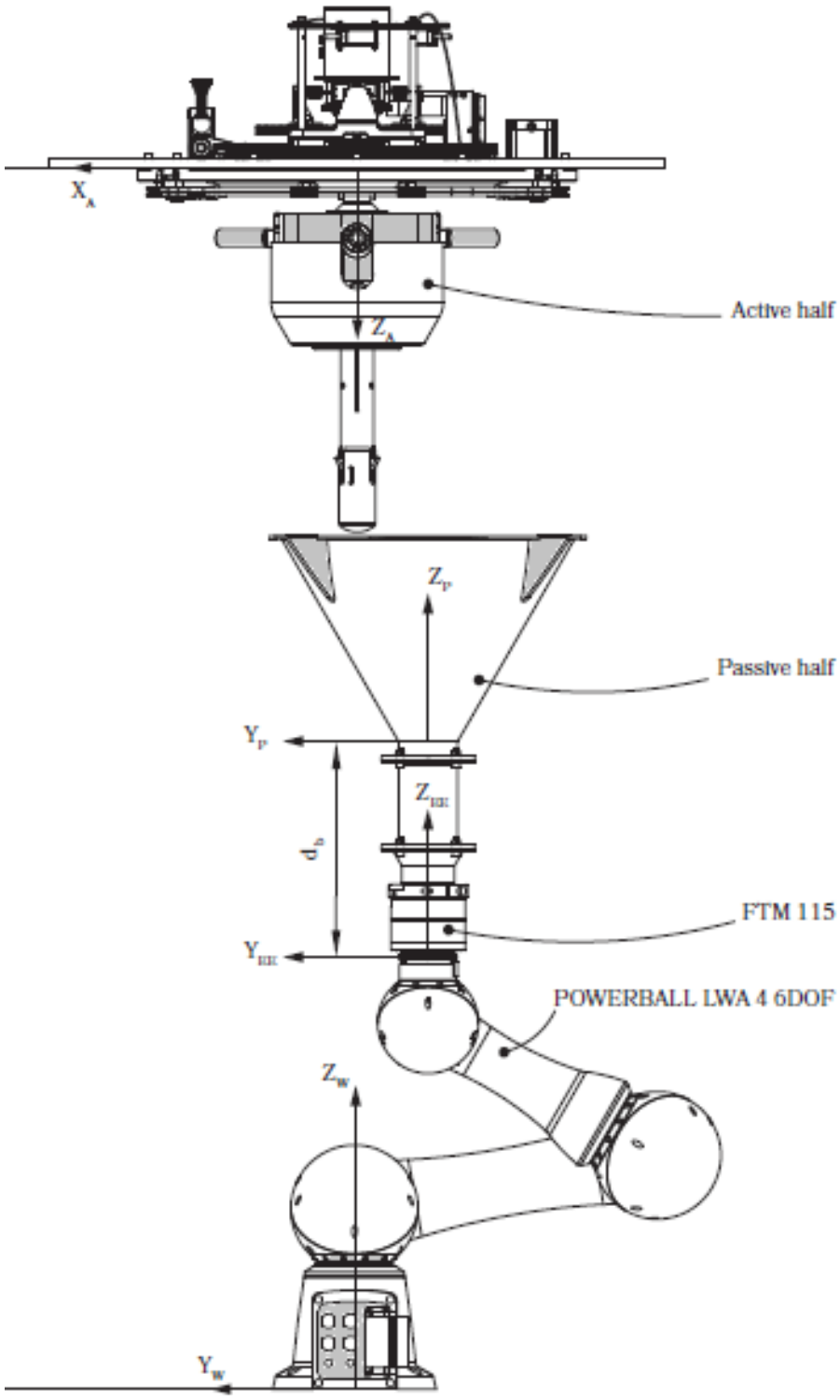

Figure 5.3: Testing environment with robotic manipulator

The chosen manipulator is a POWERBALL LWA 4 6DOF, manufactured by Schunk GmbH, in combination with a force/torque sensor FTM 115, also manufactured by Schunk.

The task of the manipulator is to simulate the relative motions of the two spacecrafts in on orbit conditions. To do that a set of fixed trajectories will be used, coming from

the multibody models developed in [16]. In this way, during the docking operations the male element remains still in his housing, while the female part is moved by the robotic arm.

Talking about the male control architecture, it is chosen to think about a suitable solution for the integration of the hardware in the test rig. For this reason a PCB shield is designed, where all the additional electrical components that the Arduino needs are placed together and the Arduino Mega board is accommodated over it. The PCB schematics and layout are developed using the KiCad program. The 3D model of the board is shown in figure 5.4. This solution, in addition to being a more orderly one, decreases the fault probability caused by jumper cable disconnection, and permits to have all the control hardware in an unique device.

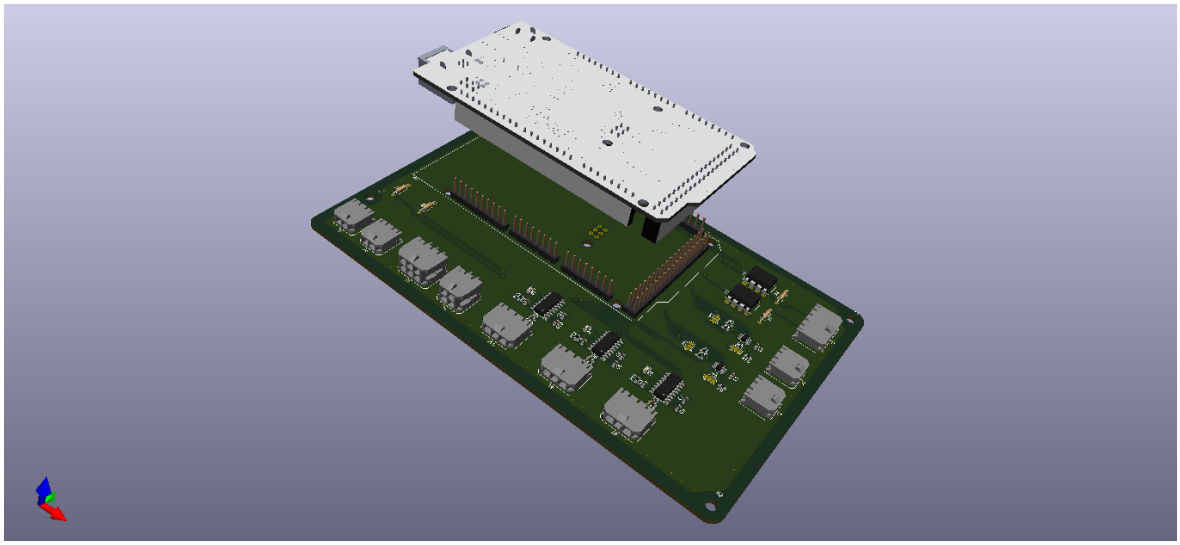

Figure 5.4: 3D Model of the developed Control Board

This experimental test has yet to be done. The purpose of this section was to prepare the test environment, provide the necessary devices and tools and give an introduction of the test execution for future works.

Chapter 6

Conclusions

In this chapter the obtained results in this thesis work are summarized, and the envisaged future work is discussed.

The target of this project was to design and develop a suitable control system for a docking mechanism for on orbit servicing missions between spacecrafts. The first step was to study the state of the art of the present docking mechanism technologies, in order to create a solid knowledge about it.

A deep study of the mechanical plant that was already assembled in TAS-I laboratory was conducted, taking into account the final mission, the expected behaviour and all the possible issues. The technical specifications of the mechanical parts and of the electrical one are collected and the compatibility of the components with the mechanism is tested.

Once the central docking mechanism behaviour is known it was selected an appropriate control architecture to implement over it and the selection of the additional electronic hardware was done. A central control architecture was designed, using an Arduino Mega board as master and the motor controller drivers as slaves. The Arduino board communicate with the driver using the rs232 serial protocol and have to manage the docking motion using also a couple of position sensors.

The docking operations were examined and scheduled, highlighting the necessary actions to performs. So, the docking manoeuvres were organized though a finite state machine. It consists of seven states, that correspond to the seven docking stages: Full Extension, Alignment and Soft Docking, Precharge, Homing, Full Retraction, Hard Docking and Shutdown. For each stage it was studied what could be the best control strategy to apply to the motors and how to manage the sensors measurements.

The control software was written, implementing the state machine into Arduino code. A set of communication functions were developed to communicate with the controller through the selected serial protocol. Also the motion function were written, able to manage the desired mechanism movements. The result was an Arduino sketch to upload on the Arduino memory, which can works in debugging mode with an USB

connected PC or in standalone mode.

The control system was assembled and tested in two test sessions. The first was a benchmark test, with the actuators disjointed to the mechanism and in no load conditions. In this session it was checked the correctness of the communications and of the motions. Also additional improvements are added to the control system in view of future tests. In the second session all the components were assembled together with the mechanical plant and a full docking operation was performed. Here it was possible to check the mechanical movements, finely adjust the control parameters, collect the power consumption during the operations and define the execution timing.

The result of this thesis work is a full working prototype of a central docking mechanism for small satellites, able to accomplish a docking manoeuvre and ready to the final docking tests. Currently the final test environment is under construction and the final technical refinements are in development.

There are still some issues to be solved:

- the state transition between the soft docking and the precharge states has to be implemented yet. For now the transition is done by the use of a manual pushbutton. A reasonable idea could be to insert a time counter in the process such that when the probe position adjustments along the base plan are not requested for a certain amount of time, the soft docking could be considered performed. This is a software solution that doesn't needs of additional hardware sensors.

- the operation of undocking has yet to be implemented. This could be done easily improving the developed state machine and reusing the docking Arduino control functions.

- a fault error management could be implemented. Actually, if the docking manoeuvre fails it is necessary to reset the devices and restart the operations from the beginning.

Appendix A

The Epos drivers

A.1 Device Control

State	Statusword [binary]	Description
Start	x0xx xxx0 x000 0000	Bootup
Not Ready to Switch On	x0xx xxx1 x000 0000	Current offset will be measured Drive function is disabled
Switch On Disabled	x0xx xxx1 x100 0000	Drive initialization is complete Drive parameters may be changed Drive function is disabled
Ready to Switch On	x0xx xxx1 x010 0001	Drive parameters may be changed Drive function is disabled
Switched On	x0xx xxx1 x010 0011	Drive function is disabled
Refresh	x1xx xxx1 x010 0011	Refresh of power stage
Measure Init	x1xx xxx1 x011 0011	Power is applied to the motor Motor resistance or commutation delay is measured
Operation Enable	x0xx xxx1 x011 0111	No faults have been detected Drive function is enabled and power is applied to the motor
Quickstop Active	x0xx xxx1 x001 0111	Quickstop function is being executed Drive function is enabled and power is applied to the motor
Fault Reaction Active (disabled)	x0xx xxx1 x000 1111	A fault has occurred in the drive Drive function is disabled
Fault Reaction Active (enabled)	x0xx xxx1 x001 1111	A fault has occurred in the drive Selected fault reaction is being executed
Fault	x0xx xxx1 x000 1000	A fault has occurred in the drive Drive parameters may be changed Drive function is disabled

Figure A.1: Epos driver-States of the drive

The state machine describes the device states and the possible control sequence. Each single state represents a special internal or external behavior. The state of the drive also determines which commands are accepted. States may be changed using the Controlword register and/or according to internal events. The current state can be

read using the Statusword read command [1].

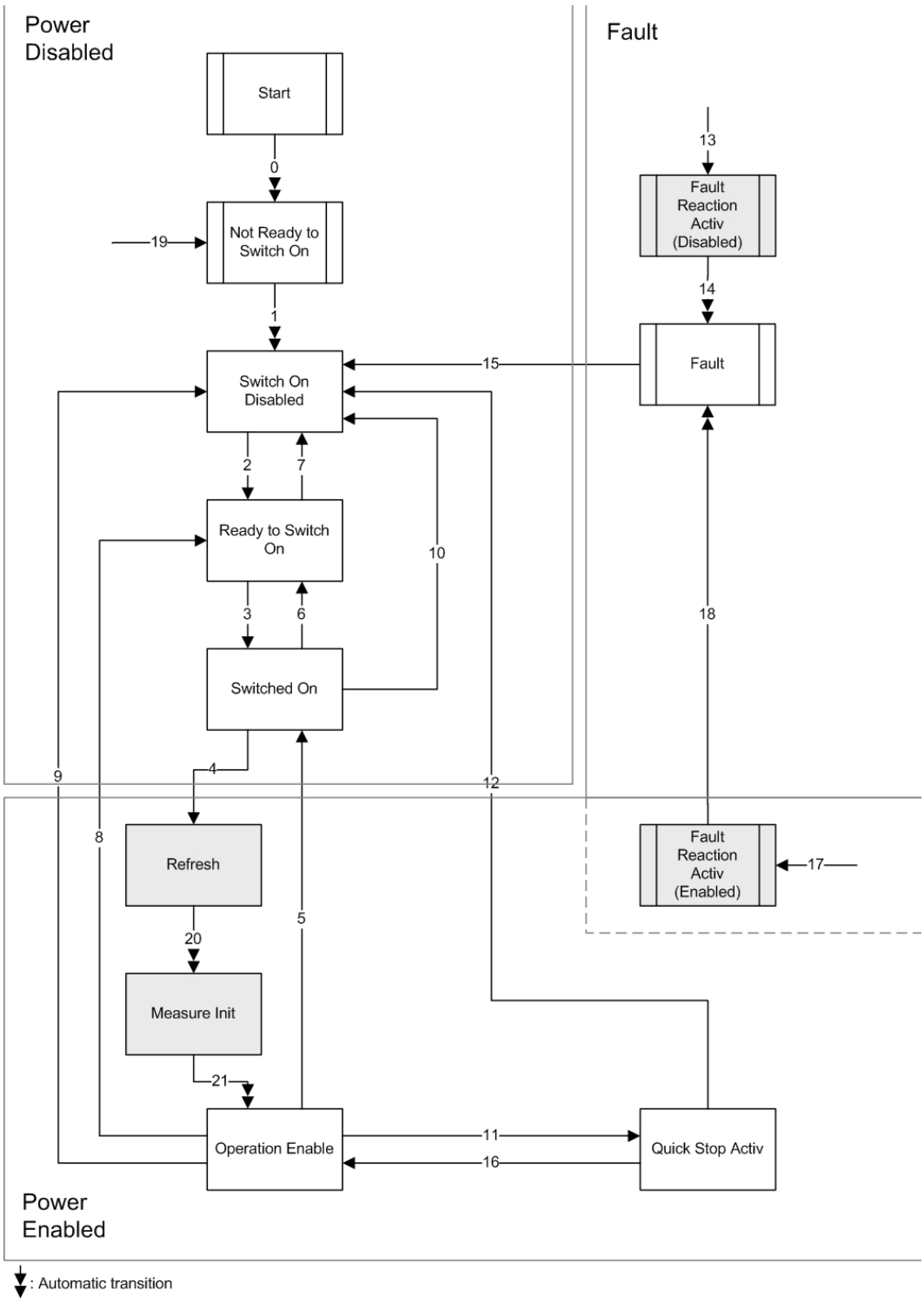

Figure A.2: State Machine of the Device Control Architecture

Transition	Event	Action
0	Reset	Initialize drive
1	Drive has initialized successfully	Activate communication
2	«Shutdown» command received	
3	«Switch On» command received	
4	«Enable Operation» command received	Refresh power section
5	«Disable Operation» command received	Disable power section; disable drive function
6	«Shutdown» command received	
7	«Quickstop» or «Disable Voltage» command received	
8	«Shutdown» command received	Disable power section/drive function
9	«Disable Voltage» command received	Disable power section/drive function
10	«Quickstop» or «Disable Voltage» command received	
11	«Quickstop» command received	Setup Quickstop profile
12	«Disable Voltage» command received	Disable power section/drive function
13	A fault has occurred not during «Operation Enable» or «Quickstop» State	Disable power section/drive function
14	The fault reaction is completed	
15	«Fault Reset» command received	Reset fault condition if no fault is present
16	«Enable Operation» command received	Enable drive function
17	A fault has occurred during «Operation Enable» or «Quickstop» State	Execute selected fault reaction
18	The fault reaction is completed	
19	A Node Reset was received	Initialize drive
20	Refresh cycle finished	Enable power section
21	Measure Init cycle finished	Enable drive function

Figure A.3: Epos driver-Transition events

A.2 Modes of Operation

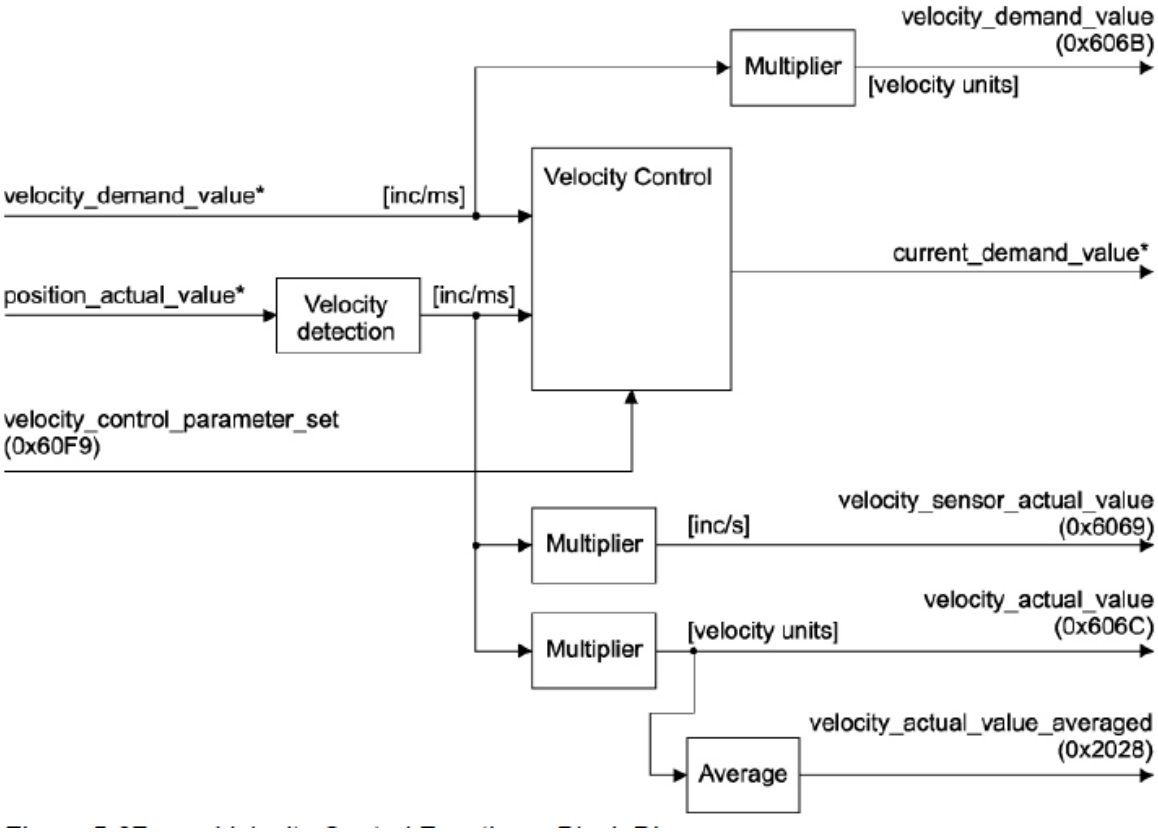

Figure A.4: Velocity Mode Block Diagram

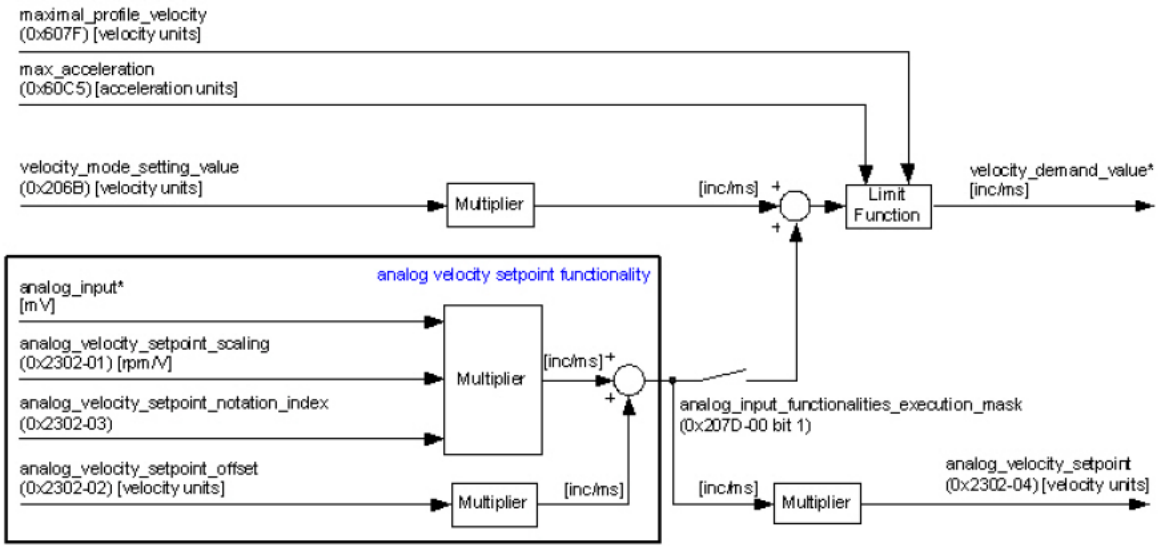

Figure A.5: Velocity Control Function

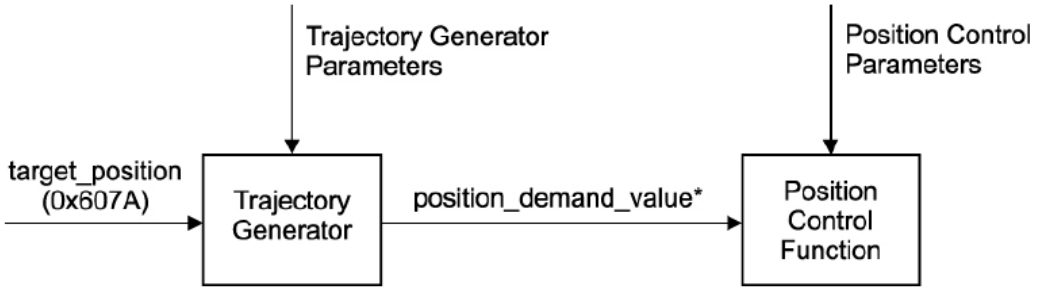

Figure A.6: Profile Position Mode Overview Diagram

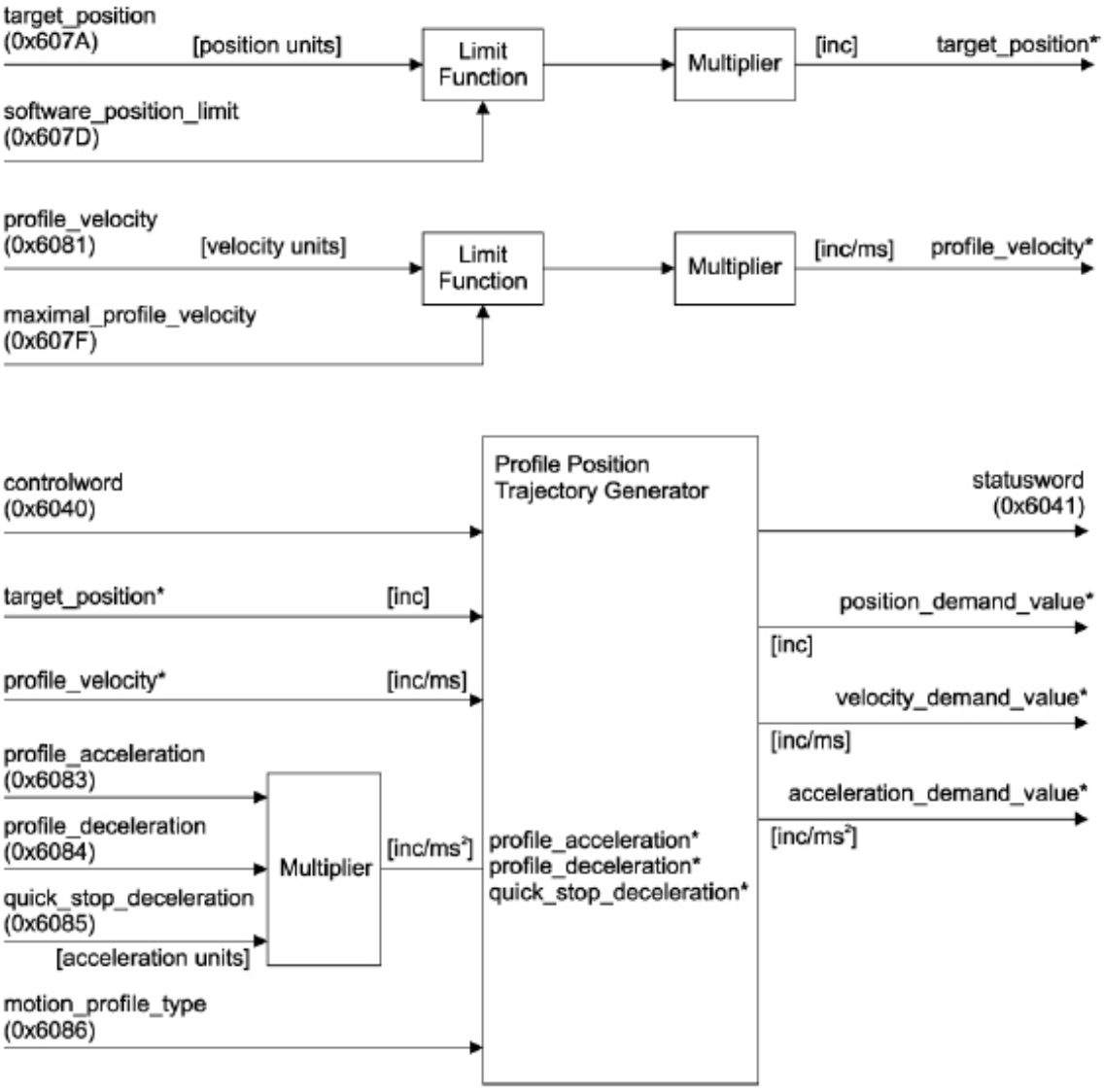

Figure A.7: Profile Position Mode diagram

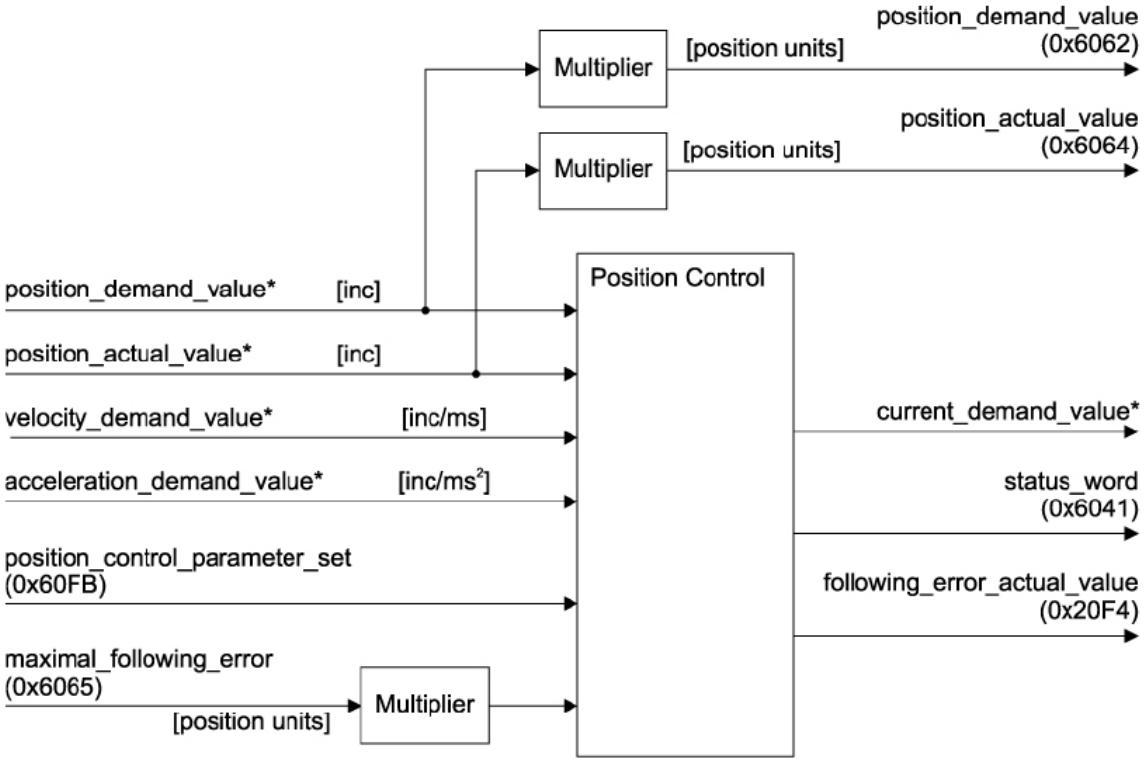

Figure A.8: Profile Position Control Function

A.3 Communication

Sending Data Frame

When sending a frame to Epos, you will need to wait for different acknowledgment [1].

- After sending the first frame byte (OpCode), you will need to wait for the EPOS "Ready Acknowledge".
- Once the character "O" (okay) is received, the slave is ready to receive further data.
- If the character "F" (failed) is received, the slave is not ready to send data and communication must be stopped.
- After sending the checksum, you will need to wait for the "End Acknowledge". The slave sends either "O" (okay) or "F" (failed).

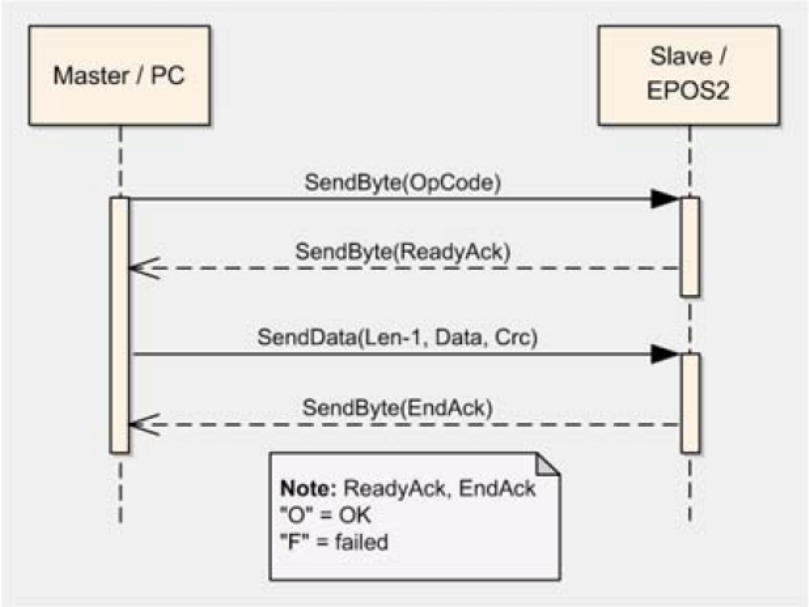

Figure A.9: Sending a data frame to Epos through rs232

Reading Data Frame

The data flow sequence is identical as for sending a data packet, only in the other direction. The master must also send the two acknowledges to the slave.

- The value of the first field must always be 0x00, thus representing the operation code describing a response frame.
- After receiving the first byte, the master then must send the "Ready Acknowledge".
- Send character "O" (okay) if you are ready to receive the rest of the frame.
- Send character "F" (failed) if you are not ready to receive the rest of the frame.
- If the EPOS2 does not get an "O" within the specified timeout, the communication is reset. Sending "F" does not reset the communication.
- After sending the "Ready Acknowledge" ("O"), EPOS2 sends the rest of the data frame. Then the checksum must be calculated and compared with the one received. If the checksum is correct, send acknowledge "O" to the EPOS2, otherwise send acknowledge "F".

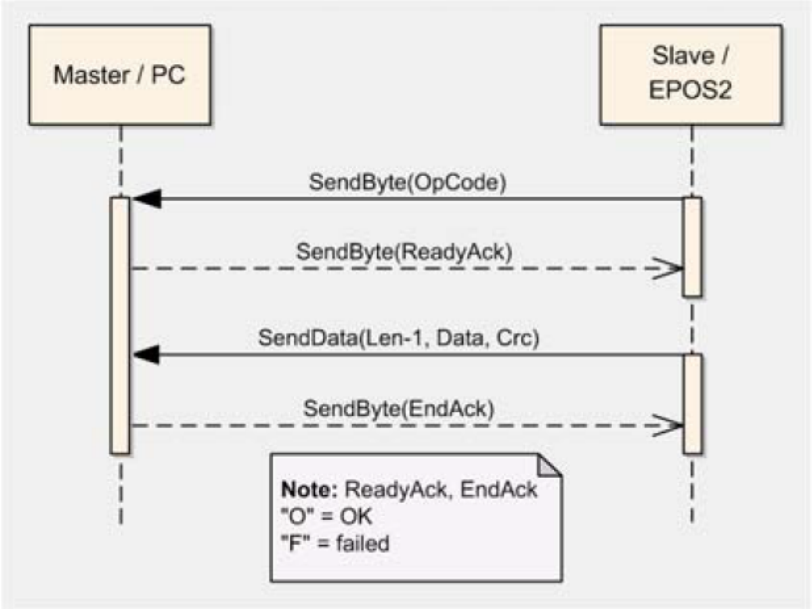

Figure A.10: Reading a data frame from Epos through rs232

Frame Structure

The data bytes are sequentially transmitted in frames. A frame composes of a header,a variably long data field and a 16-bit long cyclic redundancy check (CRC) for verification of data integrity.

OpCode (8-bit)	Len-1 (8-bit)	Data[0] (16-bit)	...	Data[Len-1] (16-bit)	CRC 16-bit
HEADER		DATA			CRC

Figure A.11: Epos data frame for rs232 communication

HEADER It consists of 2 bytes. The first determines the type of data frame to be sent or received. The next field contains the length of the data fields.

OpCode Operation command to be sent to the slave.

Len-1 represents the number of words (16-bit value) in the data fields. It contains the number of words minus one. The smallest value in this field is zero, which represents a data length of one word. The data block must contain at least 1 word. DATA The data field contains the parameters of the message. This data block must contain at least one word. The low byte of the word is transmitted first.

Data[i] The parameter word of the command. The low byte is transmitted first.

CRC The 16-bit CRC checksum. The algorithm used is CRC-CCITT. The CRC calculation includes all bytes of the frame. The data bytes must be calculated as a word.

Bibliography

[1] Maxxon motor. *EPOS Positioning Controller. Communication Guide*. October 2006.

[2] *International Docking System Standard (IDSS)*. October 2016. URL: `https://www.internationaldockingstandard.com`.

[3] John G. Cook et al. "ISS Interface Mechanisms and their Heritage". In: (January, 2011).

[4] Maxxon motor. *EPOS2 Positioning Controllers. Firmware Specification*. November 2017.

[5] Kollmorgen. *P70530 (DC) High Performance Micro-Stepping Drive Reference Guide*. Revision C 2/2012.

[6] Maxxon motor. *EPOS2 Positioning Controller. EPOS2 24/5 Hardware Reference*. Edition May 2016.

[7] Maxxon motor ag. *Maxxon EC (BLDC) motor*. April 2016 edition.

[8] Maxxon motor. *EPOS2 Positioning Controller. Cable Starting Set*. Edition October 2014.

[9] Maxxon motor. *EPOS2 Positioning Controller. EPOS2 24/2 Hardware Reference*. Edition December 2013.

[10] Joseph N Tatarewicz. "The hubble space telescope servicing mission". In: *fesb* (1998), p. 365.

[11] W. Fehse. *Automated rendezvous and docking of spacecraft*. vol. 16. Cambridge University Press, 2003.

[12] Micro-Epsilon Messtechnik GmbH Co. *Instruction Manual, optoNCDT 1420*. 2008.

[13] E. Stoll et al. "On-orbit servicing". In: *IEEE Robotics Automation Magazine* 16.4 (2009), pp. 29–33.

[14] *The Partnership: A NASA History of the Apollo-Soyuz Test Project*. Dover Pubns, 2013.

[15] "A review of space robotics technologies for on-orbit servicing. Progress in Aerospace Sciences". In: (2014).

[16] Tharek Mohtar. "Design and modeling of a space docking mechanism for cooperative on-orbit servicing". In: (2020).

[17] *Arduino Language Reference*. URL: `https://www.arduino.cc/reference/en/` (visited on 2020).

[18] *Arduino.cc*. URL: `https://www.arduino.cc` (visited on 2020).

[19] Michelle Belleville. *Hubble Space Telescope Main Page*. URL: `https://www.nasa.gov/mission_pages/hubble/main/index.html`.

[20] *SAPERE Project*. URL: `https://www.ctna.it/project/sapere/`.