POLITECNICO DI TORINO

Master's Degree in

Mechatronic Engineering

Master's Thesis

Design and development of a control system for the Robotic Arm Schunk LWA 3

Supervisor
Marcello Chiaberge
Co-Supervisor
Andrea Merlo

Candidate
Fabio Antonacci

Academic Year 2020/2021

To my family

Abstract

This thesis presents the design of the control system for a robotic arm, the Schunk Lightweight Arm 3. This manipulator is used as part of the on-ground simulation of on-orbit satellite servicing performed at the Rendezvous & Docking facility of Thales Alenia Space in Turin. This area was built in the framework of the STEPS (Sistemi e Tecnologie per l'EsPlorazione Spaziale) program; here, the docking maneuver is simulated on a micrometric black granite floor where two vehicles (the chaser and the target) move floating over a thin air film in order to obtain a frictionless dynamic and have a testing environment able to simulate orbiting conditions. The RV&D facility is also equipped with two robotic manipulators necessary for the capture but more in general for servicing (e.g., refueling, repairing and/or replacement of parts, removing of space debris) of a target. The control development started from the study of the mechanical and electrical configuration of the robotic arm. The technical specifications of actuators and motor drivers were examined and a suitable control architecture has been selected. The final solution consists of a workstation that acts as the central control system, connected via CAN bus to the terminal block of the LWA 3, powered by a 24 VDC voltage. The control system was entirely designed using the Python programming language. The protocol used to exchange data between the master and all the modules of the Schunk LWA 3 is unified and independent from the bus interface used. The module receives serial data, interprets it and acknowledges the command. A command typically is composed by an Identifier, followed by a Command ID, a Parameter ID or a Motion ID, based on the task that the manipulator has to perform, and data bytes if required. The first step of the control design was the creation of a Python script to initialize and enable the CAN communication from the workstation to the robotic arm. The next step was the construction of a CAN message to give commands to the LWA 3: each one of those is the concatenation of a series of strings that correspond to a different section of the CAN message. All the functions for building the string of data were divided in different Python scripts, based on the type of tasks that they perform. They can vary from general commands (e.g., resetting the encoders, halting the motion of the arm or returning to the home configuration) to more complicated ones like the joint control and some direct kinematics functions. Joint control can be performed either choosing the velocity at which each module must move

or setting the final position that it must reach. Also, some functions were implemented to directly send the arm in pre-defined configurations from where it is easier to reach objects to be caught. A crucial task was the creation of a direct kinematic function that constantly computes the position and orientation of the end-effector and prints them on the screen. The commands were sent to the arm through a Graphical User Interface that replaced its previous version of the Command Line Interface launched from the terminal. The final result is a window that gives the user a more intuitive control system. The GUI has a section for sending commands to the arm on the left and a telemetry one on the right where all the relative positions and velocities of each module, the position and orientation of the end-effector are displayed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since 1957, when the first artificial satellite was launched, the Soviet Sputnik, the space exploration has made great strides. Until today about 6000 satellite have been launched for scientific use, by which it was possible to expand the knowledge of the phenomena occurring in the solar system and beyond, military and commercial use, allowing the reaching of new frontiers in the field of communications and cannot be forgotten of course the recent undertakings which have enabled the creation of the first orbiting telescope, the Hubble Space Telescope, and the first international space station, the ISS.

The human thirst for knowledge requires technological progress to enable new scientific ventures. For this reason we are faced with the need to exploit the already launched flight systems even more and to construct large structures in situ, considering as guideline the need to reduce, reuse and recycle as far as possible. All these needs require some aspect of what satellite servicing can offer, therefore we can consider the satellite servicing as a tool able to enable the creation of a new architectures needed to conquer the next frontiers in space.

On-orbit servicing involves a new class of space missions in which a servicer (or chaser) spacecraft is launched into the orbit of a target (or client) spacecraft, the client. The servicer navigates to the client with the intention of manipulating it, using a robotic arm. A crucial operation of the on-orbit servicing missions is the docking maneuver, defined as the joining of the two separate free-flying space vehicles. Before starting this procedure however, a crucial task is to catch the satellite on which the maintenance needs to be performed.

This thesis work was developed in the Rendezvous & Docking (RV&D) facility of Thales Alenia Space in Turin where the docking maneuver is simulated with two satellites floating on a granite surface. The aim of this work is to create a control system for the Schunk Lightweight Arm 3, one of the two robotic arms in the facility, that has the function of catching the satellite. The control tasks and commands created range from simple setup function used at the power on, to the joint control of every one of the seven modules of the Schunk LWA 3, to the computation of the pose of the end-effector via Direct Kinematics. Also, some commands where implemented to move the robotic arm in some pre-defined configurations from which the motion to the operational positions defined by the user is easier.

This dissertation contains, besides this introductory part, five chapters. The next one gives an overview of the history and characteristics of on-orbit servicing missions.

Chapter 3 firstly introduces the manipulators and robotic arms from a mechanic and structural point of view. Then, the characteristics and setup of the Schunk LWA 3 are explained.

Chapter 4 gives an overview of the Controller Area Network (CAN) protocol, starting from how it was born to the actual state of the ISO 11898 and the main features of the protocol today. Then follows a detailed description of how it is implemented for the Schunk LWA 3, focusing in particular on the construction of a CAN message based on the various tasks that need to be performed by the manipulator.

Chapter 5 describes the development and the structure of the control system as well as the Graphical User Interface (GUI), focusing on the various packages created to perform all the functions required for the operation of the robotic arm. Later in the chapter, the tests performed to guarantee the correct functioning of the control system and of the GUI are explained.

Chapter 6 presents the conclusions of this work and the possibilities for future development.

After the conclusions there are two Appendices: the first one is a user-manual to properly start-up and use the the manipulator and the framework, and avoid critical situations that can compromise the correct functioning of the arm. In the second appendix it is possible to find the code that implements the control algorithms described in Chapter 5.

Chapter 2

On-orbit servicing missions

Except for manned servicing operations, there is no maintenance infrastructure for space systems. The traditional approach is to build in reliability and to replace the system in case of failure. Space systems therefore offer a limited degree of flexibility to adapt to evolving conditions during their long lifetimes. The on-orbit servicing could change this paradigm by providing a physical access to the satellite after it has been launched and offering repair services and a broader range of upgrades that would be cheaper alternatives to satellite replacement.

In the following sections there will be described the reasons why interventions in orbit are necessary, quoting some missions that have now become milestones in the history of satellite servicing. An overview on experience and ongoing projects in this field will follow, with a particular description in the last section of the systems used for the simulation of rendezvous and docking maneuvers, typical of maintenance missions.

2.1 The importance of satellites servicing

Every sector of satellite utilization could use the servicing to increase the efficiency and bringing the benefits of space operations at lower overall cost.

One of these sectors is the commercial and strategic. Indeed they provide a lot of facilities being used as repeaters for broadcasters, for communications and surveillance to great distance, for connections between computer and also for the GPS (Global Positioning System) network. Satellite servicing allows to deriving more utility from these activities promoting the expansion of the sector. The most important applications of on-orbit servicing in this area are refuelling and replacement of damaged components. These operations can extend the satellite's useful life, improve technology, reduce the life cycle cost, decrease the redundancy in the building of satellite, and increase the mission performance, providing additional maneuvering or deorbit propulsion capability. All of these operations can be conducted from specific space area, that includes the most used orbits like the Geostationary Earth Orbit (GEO) belt, that surrounds the Earth at high altitude and low inclination, and the Low Earth Orbit (LEO) near to the polar inclinations. Moving in this zone, the services can satisfy the requirement of many customers. However other servicing missions could also be conducted in other orbits but on an as-needed basis. [11]

Another essential benefit that comes from the servicing missions is the possibility to modify the orbit of the satellite in order to relocate it in the correct position or to avoid a space detritus. Many communications and weather monitoring satellites, located in GEO, suffer periodic or not variations of their position, due to perturbations from astronomical source like the solar wind and gravitational field of planets or also linked to the gravitational potential of Earth (e.g. Earth Oblateness). Therefore it is clear that in order to avoid physical and communicational interference, and in this way reduce the risk of mission failure, it is necessary to intervene by correcting the satellite orbit. Another menace must also be fronted for the mission's success, that is growing and will reach point of no return in the near future, the space debris. The space junk includes all orbiting useless or unused objects like fragments of satellite, materials and dust expelled from the rocket engines and also stages of rockets, often with residual fuel still inside them therefore potentially explosive upon impact with other objects. These impacts can be attributed at least partially to the 200 explosions recorded in orbit whose cause is unknown. The problem concerns mainly the GEO and LEO orbits, densely populated areas where an impact between two objects could trigger a cascading action, the so-called "Kessler Syndrome", causing immeasurable damage to spacecrafts before fully operational and functioning and therefore economic losses. An example of the danger of an on-orbit collision is that one occurred in February 2009 between one of the American satellites of a private constellation, Iridium 33, and the Russian communications satellite Cosmos 2251, that was in disuse for at least five years at the time of impact. The collision, occurred at 800 km altitude from earth, caused a detritus cloud composed of at least 600 pieces of size greater than 10 cm. Some of them reentered in significant numbers will likely remain in orbit for the next 25-50 years.

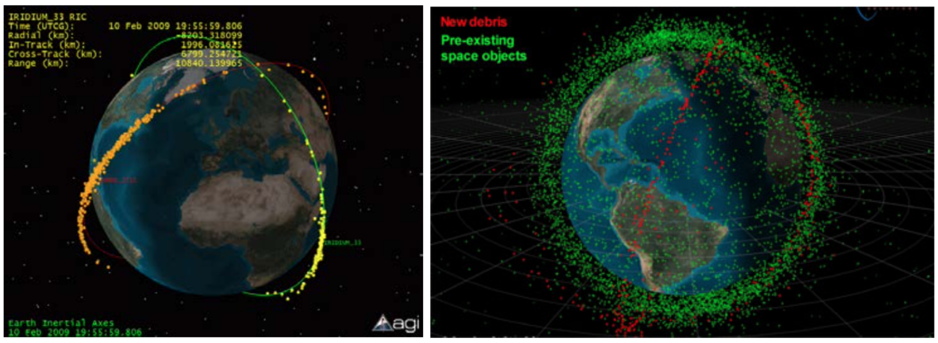


Figure 2.1: A map of the debris clouds just after (on the left) and after three years (on the right) the Iridium/Cosmos collision (credits: Celestrak/AGI Viewer 9)

To avoid these disasters all probes on orbit are catalogued by the "United States

Strategic Command" (USSTRATCOM). In this catalogue are listed the actual orbital parameters of about 15000 objects, whose size limit is between 5-10 cm, for altitude below some thousand kilometres and 0.5-1 m for higher orbits (up to geostationary), to constantly track and detect their position. Only the 6% of these objects are operative satellites, the 24% are spacecrafts in disuse, the 17% are stages of rockets and about the 40% are fragments of spacecrafts. In order to increase the knowledge of circumterrestrial environment and make it safer, even Europe has recently started its program of Space Situational Awareness (SSA). However, detecting and monitoring the location of space debris is not enough, the most effective measure to mitigate them is their removal by deorbit either by the re-entry in atmosphere or by an appropriate vehicle which exploits the servicing technologies such as orbital maneuvering, autonomous rendezvous, docking and robotic manipulation. [13]

Besides commercial and strategic benefits, the satellite servicing offers also scientific and technological ones. Obviously, the first applications with these benefits are the repair and refurbishment, in fact they allow the rapid development of new technologies, improving the mission performance, and therefore the scientific return, by replacing or upgrading subsystems thus increasing the reliability. Furthermore the satellite servicing is the main enabler for new space architectures: the ISS is one example of such structures which could be followed by large observatories to reach new frontiers in astronomy, power farms or refueling depots to supply the spacecrafts in near-Earth or planetary mission and also interplanetary spaceships. All of these structures are physically large, expensive and not launched in one piece. Only human and robotic servicing can assures that each of their elements are assembled, configured for operation, tested and even maintained and upgraded.

2.2 The history of servicing missions

During the last 40 years several repair and maintenance missions on orbit have been performed. Below are listed those that have helped to define the concept of on-orbit servicing.

- **Skylab 2** [1973]: a mission designed to repair the NASA's first space station Skylab. During its launch the detachment of a cover had destroyed a solar array and damaged a micrometeoroid shield. These failures threatened the mission's success because any attitude adjustments to maximize the power produced from the solar arrays to maintain the station's operation would lead to its subsequent overheating. Therefore the only solution was a servicing mission to replace the thermal shield. The crews of the Skylab 2 and 3, after a series of Extravehicular Activities (EVAs), concluded the first demonstration of on-orbit repair successfully. [10]

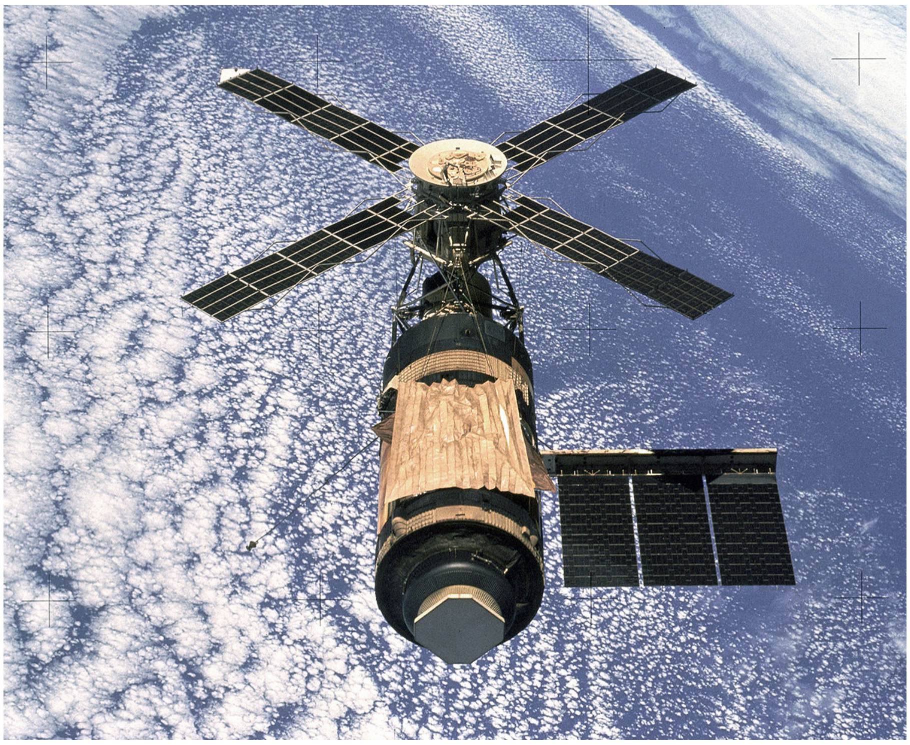


Figure 2.2: The repaired Skylab (credits: NASA)

- **Solar Maximum Repair Mission (SMRM)** [1984]: this mission allowed the successful first use of a space shuttle in the satellite servicing. The probe Solar Maximum Mission (SMM) was launched to investigate the solar phenomena. It operated for one year before a failure in the attitude control system occurred. It is worth noting why its operational life was significantly increased by the direct intervention of a manned space mission, in fact the crew's astronaut EVAs of the space shuttle Challenger fully restored the probe operation and also equipped it with a hook in order to make it able to be easily docked by a robot arm in case of new repairs, extending thus its life for another five years.

- **Hubble Space Telescope Servicing Mission** [1993-2009]: the greatest space telescope ever built has revolutionized the history of astronomy thanks to several and important findings. The most significant is the discover of the mysterious form of energy that moves the universe called Dark Energy, thanks to the observations of life cycle of supernovae. The first problems for Hubble occurred few days after the launch. In fact all the first images that came from the telescope were completely blurred, because of an optical flaw in the primary mirror and the thermal shaking induced by the solar arrays during orbital sunrise and sunset. A servicing mission was already planned to refurbishment and maintenance but not to resolve a problem of such magnitude. The first servicing mission was an extraordinary achievement, 35 hours of EVAs were necessary to bring Hubble to the planned capabilities. There were four other missions whose purpose was

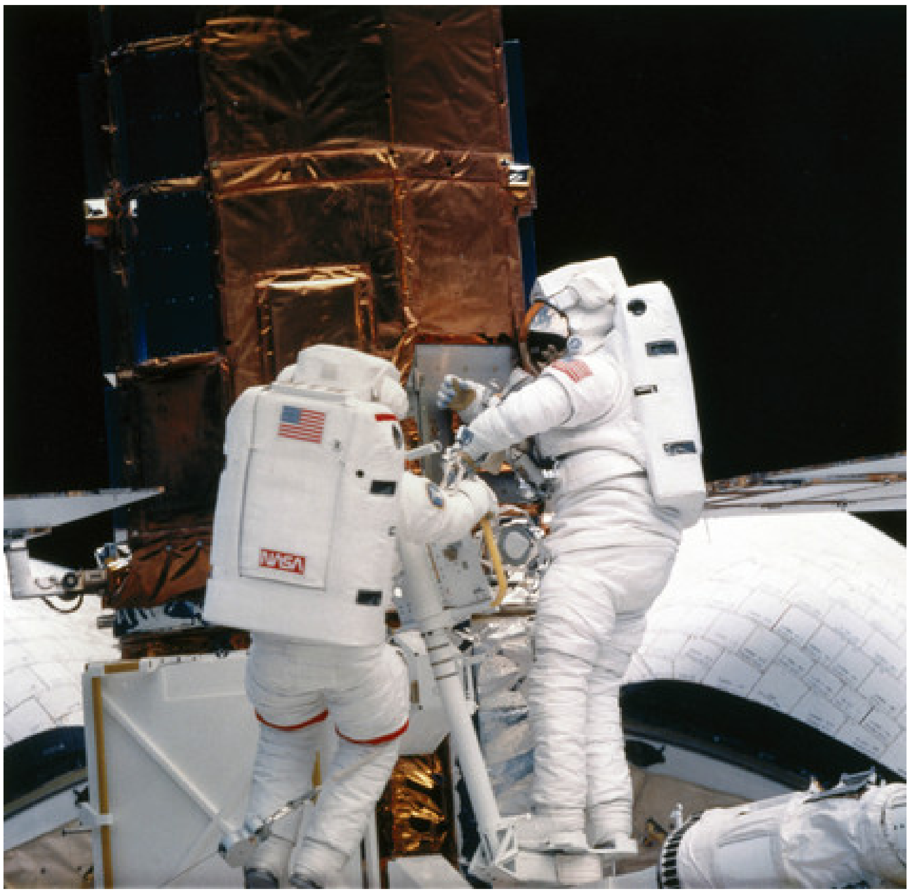


Figure 2.3: On orbit repair of Solar Max (credits: NASA)

primarily refurbishment, maintenance and adding of new generation instrument to improve the performances of entire telescope. These missions were a clear demonstration of the benefits and versatility of satellite servicing. [5]

- **International Space Station (ISS)** [1998-2011]: the ISS is the humankind's largest artificial satellite that orbit around the Earth. It serves as an orbital human outpost where a wide variety of research is conducted from the study of astronomical phenomena to the investigation on biological effects produced by the space on living beings. The sheer size not allowed the assembly, test and launch of the entire station, therefore the final integration and checkout of the each element occurred on orbit, where 143 spacewalks were necessary to complete its assembly for a total of about 900 hours of EVAs. In addition to assembly, the astronauts had to perform a variety of on-orbit maintenance tasks including clearing solar array panels snagged during the deployment, repairing a torn array and replacing failed component. The most recent was the repairing of a failed cooling system. Also the robots have played an important role in ISS construction and maintenance. The space shuttle robotic arm was used to remove new elements and transfer them to the Space Station Remote Manipulator System (SSRMS) for berthing. The last one is also used to grasp and berth

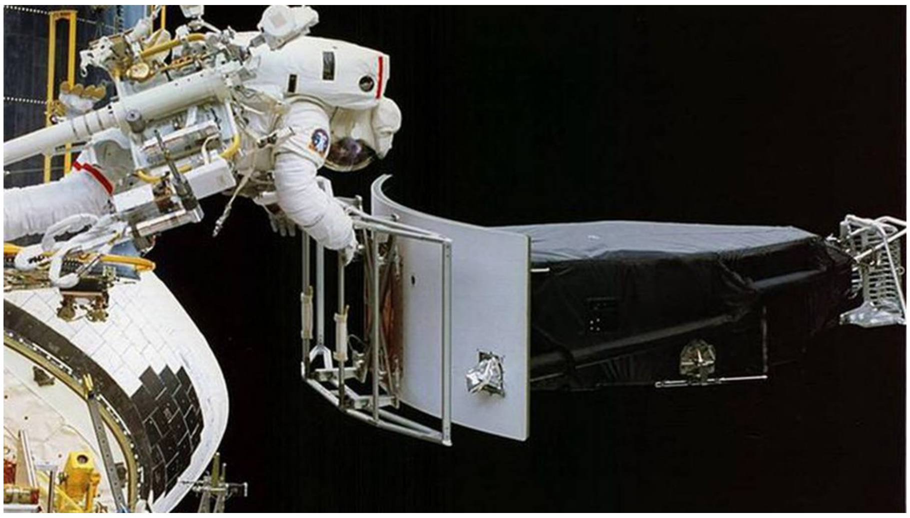


Figure 2.4: the Astronaut J. Hoffman removes Wide Field and Planetary Camera 1 during the change-out operations (credits: NASA)

visiting vehicles and provides even a platform for spacewalking astronauts for the assembly and maintenance operations.

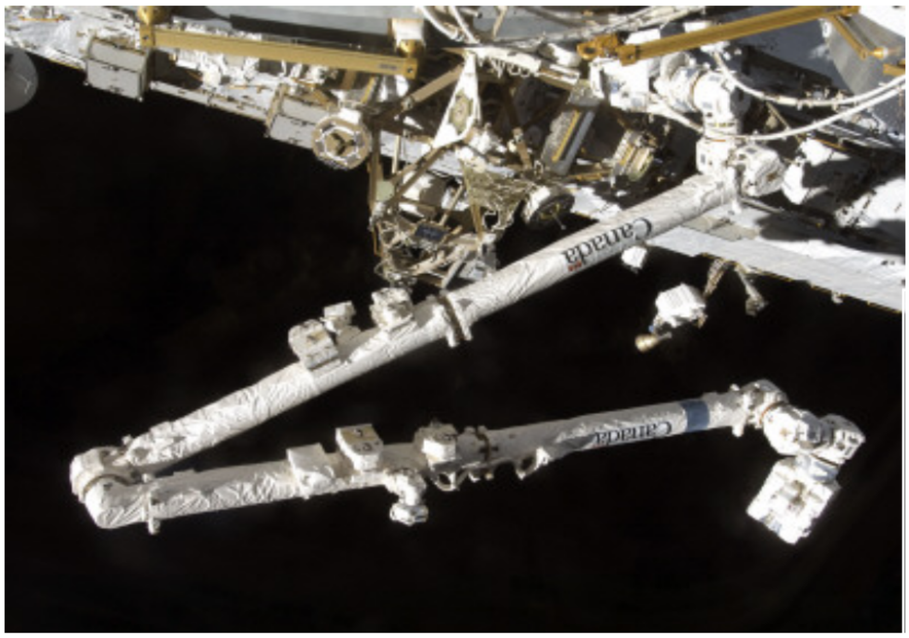


Figure 2.5: The Canadian- Built Space Station Remote Manipulator System during undocking activities (credits: NASA)

The ISS is equipped with another manipulator, Dextre (the Canadian Special Purpose Dexterous Manipulator), that can perform more precisely with respect to the manipulators mentioned previously, thanks to two independent robotic arms. Therefore in some cases can replace the astronauts, removing failed components and installing spare units. Moreover, operating telerobotically from the ground, it will be used to service and refuel the satellites. [8]

The last improvement to ISS was introduced in 2011r. This is a "human equiv-

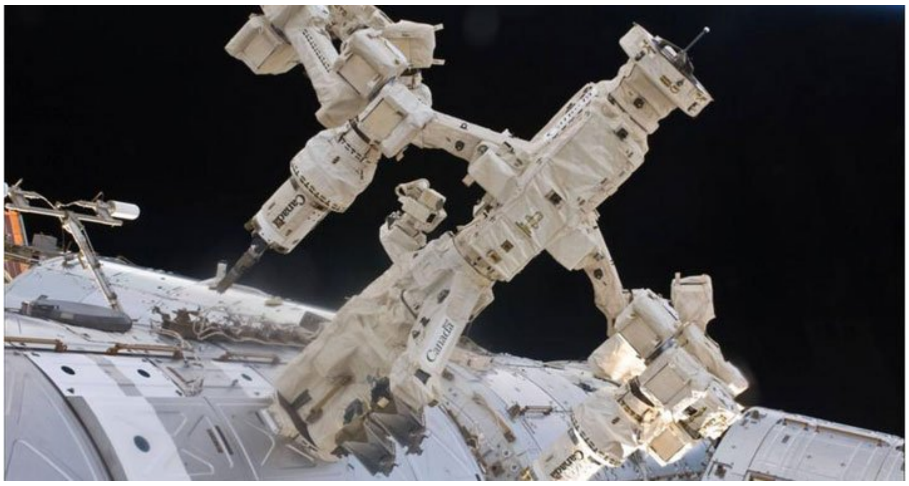


Figure 2.6: Dextre, built by the Canadian Agency, has arms more than 9 feet in length and can attach power tools as fingers (credits: NASA)

alent" robot, Robonaut 2 (R2), that is able to handle tools with a dexterity exceeding that of suited astronaut, therefore can help humans work and explore where the risks are too great for people.

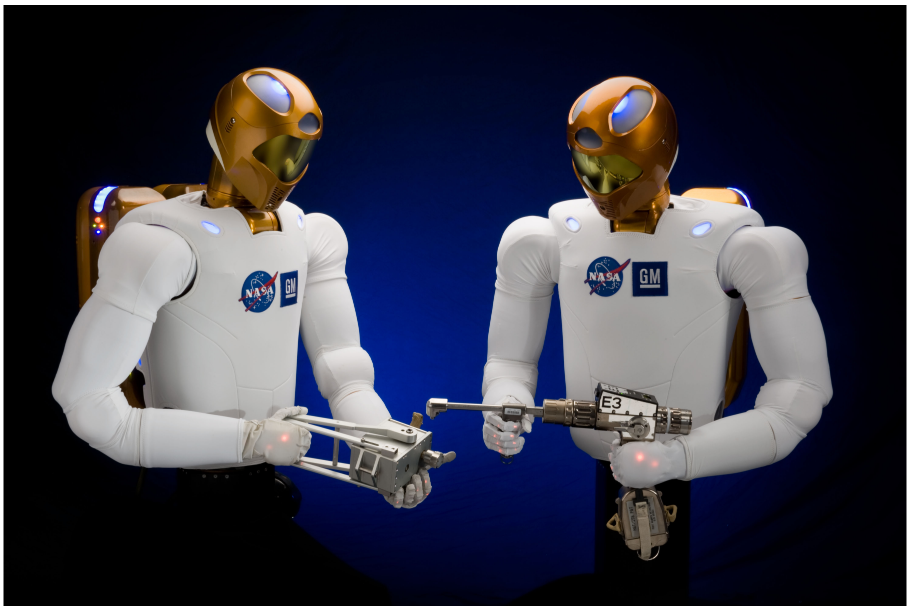


Figure 2.7: Robonaut 2 (credits: NASA)

It is clear that the human servicing model is justified in the significant applications, where the cost of the mission is comparable to that of the vehicle to maintain, but is not applicable to most commercial space systems being extremely expensive. Therefore over the years it has been developed a new autonomous and teleoperated vehicle technology capable of performing maintenance through robotic manipulators (like Dextre), reducing mission costs. Some historical and recent examples of this technology demonstration activities, that were fundamental in the development of the modern servicing technologies, are listed below.

- **Engineering Test Satellite No. 7 (ETS-VII)** [1997]: the ETS-VII was a satellite developed and launched by the National Space Development Agency of Japan (NASDA, now JAXA). It was the first successful demonstration of autonomous rendezvous and docking involving a "chaser" satellite, that for the first time was equipped with a robotic arm 2 metre long, the first free-flying space robot in history, able to grasp a "target" satellite. It also supports several experiments on teleoperation, latency, assembly of a space structure and dynamics coordination between the arm and the spacecraft. Although it was originally intended to be used for one and a half years, the satellite was functional for a period of almost five years.

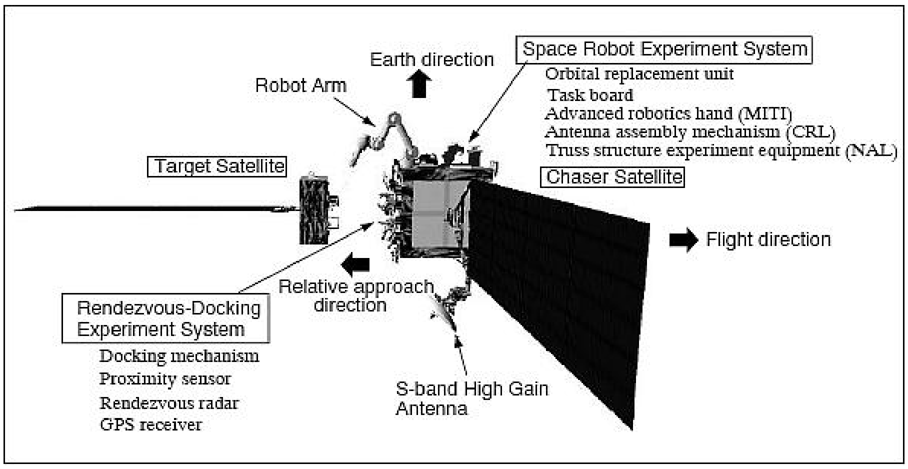

Earth direction
Space Robot Experiment System
Orbital replacement unit
Task board
Advanced robotics hand (MITI)
Antenna assembly mechanism (CRL)
Truss structure experiment equipment (NAL)
Robot Arm
Target Satellite
Chaser Satellite
Flight direction
Relative approach direction
Rendezvous-Docking Experiment System
Docking mechanism
Proximity sensor
Rendezvous radar
GPS receiver
S-band High Gain Antenna

Figure 2.8: Configuration of the ETS-VII autonomous docking maneuver (credits: Tohoku University)

- **XSS-10 and XSS-11** [2003-2005]: the two microsatellites, built by the United States Air Force Research Laboratory, demonstrated the key technologies for satellite servicing. Indeed, the first acquired and tracked its second stage and performed a series of inspections from 100m to 35m of distance from this object. The second one demonstrated autonomous operations with a "non-cooperative" space object (i.e. without accommodations for servicing).

- **The Orbital Express mission** [2007]: this mission was conducted by the Defence Advanced Research Project Agency (DARPA) and was the first mission that completed successfully all robotic satellite servicing activities. The launched system consisted in two spacecraft: The Autonomous Space Transport Robotics Operations (ASTRO) vehicle, that performed autonomous docking with the other spacecraft, NEXT-generation serviceable Satellite (NEXTSat). During the mission, lasted 4 months, were demonstrated also the fuel transfer and some orbital replacement units activities such as the insertion of battery in NEXTSat and the change of the flight computer on ASTRO. [3]

Furthermore in recent years some new international initiatives were announced from

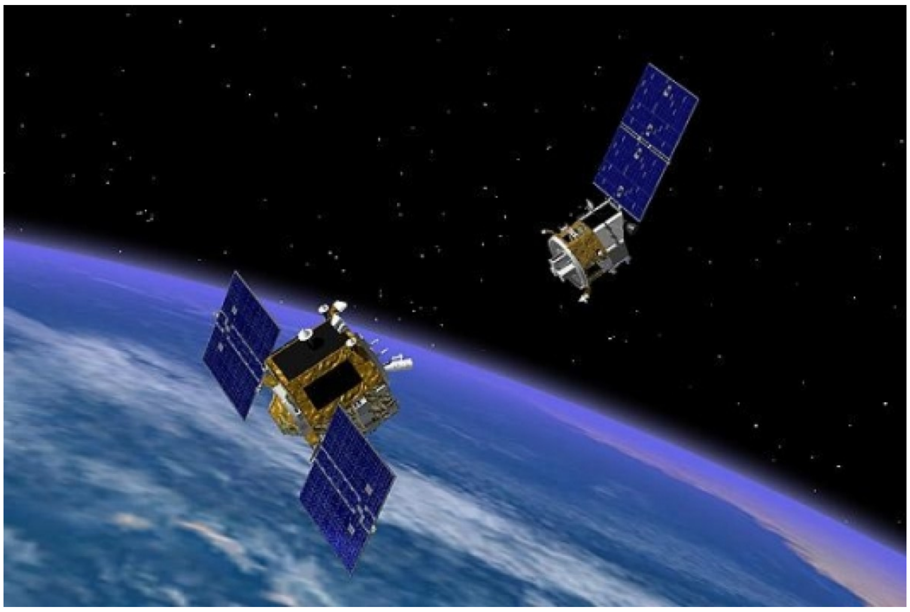


Figure 2.9: Artistic rendition of The Orbital Express mission during unmanned operations.

Canada's MacDonald, Dettwiller and Associates Ltd. (MDA) and Deutsches Zentrum für Luft und Raumfahrt e.V. (DLR), the German Aerospace Centre. The first one is involved in the design of a mission that could allow the satellite refuelling and the moving of the inoperative satellite in "graveyard" orbits. The servicer would rendezvous and dock with the satellite's apogee kick motor (a special motor employed on satellites destined for GEO orbit and used to reach the zero inclination), connect the tank to a fuel line and deliver propellant. The business model idea is based on the possibility of having customers paying per kilogram of fuel that has been successfully added to their satellite, opening in this way a new market share in space.

2.3 Ongoing projects and on-ground simulation

The most important ongoing project regarding on-orbit satellite servicing is the one provided by SpaceLogistics, a wholly owned subsidiary of Northrop Grumman that uses its fleet of commercial servicing vehicles: the Mission Extension Vehicle, the Mission Robotics Vehicle and the Mission Extension Pods.

The Mission Extension Vehicle is the most used out of the three and can provide servicing to geosynchronous satellite operators. It docks with customers' existing satellites providing the propulsion and attitude control needed to extend their lives. [18]

MEV-1 successfully launched on an International Launch Services' Proton rocket on October 9, 2019 from Baikonur, Kazakhstan. It has been contracted for a five-year life extension to an Intelsat-901 satellite, providing docked propulsion and servicing capabilities such as station keeping, altitude control, incline reduction, and inspection.

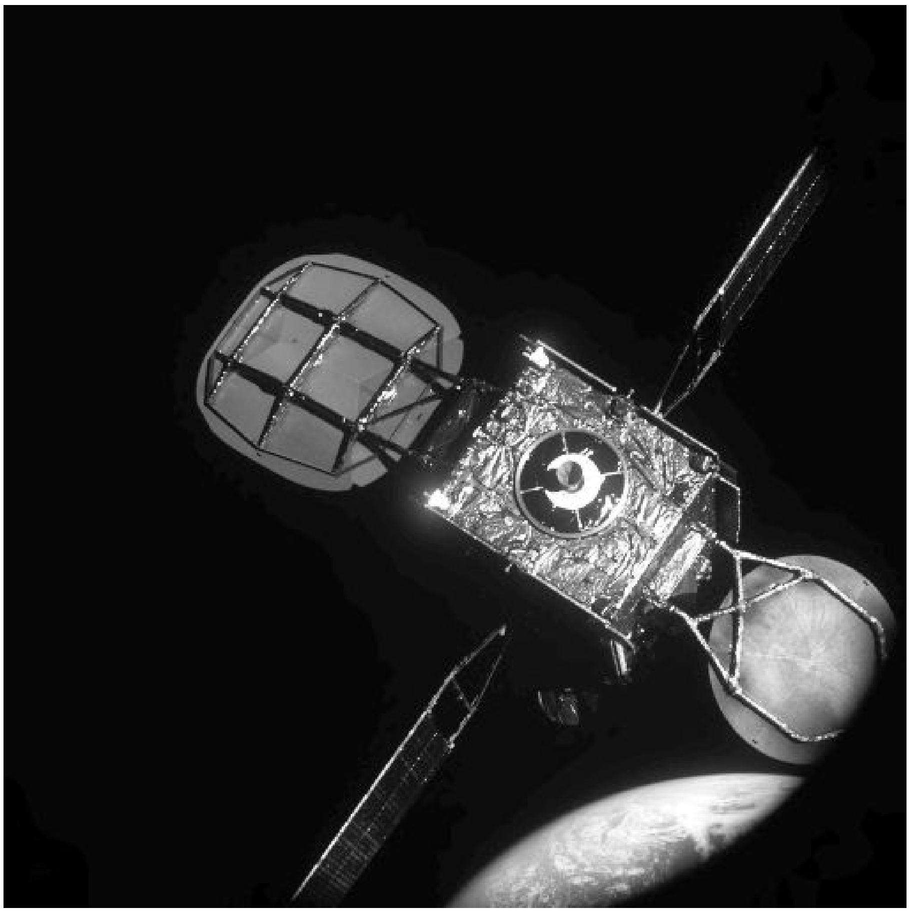


Figure 2.10: The servicing satellite MEV-1 (credits: SpaceLogistics)

The docking to the satellite was completed on February 25, 2020. Once connected to its client satellite, MEV performs on-orbit checkouts before relocating Intelsat-901 to its orbit, then it uses its own thrusters and fuel supply to extend the satellite's lifetime. When the customer no longer desires MEV's service, the spacecraft will undock and move on to the next client satellite. According to the Northop Grumman company, the spacecraft can perform multiple docking and undocking maneuvers over its 15-year life span.

The second Mission Extension Vehicle (MEV-2) launched August 15, 2020 along with the Northrop Grumman-built Galaxy 30 satellite. MEV-2 will dock with the Intelsat IS-1002 satellite in early 2021. [17] The other two servicing vehicles in the SpaceLogistics fleet, the Mission Robotics Vehicle (MRV) and the Mission Extension Pods (MEPs) are next generation systems that have not been used for any mission yet. The former, MRV, is a robotic servicing vehicle that installs the MEPs. The MRV can perform all the functions of an MEV while adding new robotic capabilities for additional services. The latter, instead, are smaller and less expensive life extension services that only perform orbit control.

One of the critical requirements of a generic on-orbit servicing mission is to ensure a safe Rendezvous and Docking (RV&D) process. Therefore this phase has to be analysed, simulated and verified in detail, not only with the classical approaches e.g. numerical simulation but even with systems that can reproduce realistic working conditions. The German Aerospace Center DLR, for example, with over two decades of experience in this field, managed to realize two different facilities for on-ground simulations. The first one was the EPOS (European Proximity Operation Simulator), able to provide test and verification capabilities for complete RV&D process. This facility consists of a rail system mounted on the floor of the laboratory to move an industrial robot toward the other up to the distance of 25m, to have a physical real-time simulations of the last critical phase (separation ranging from 25 to 0m) of the approach process; two KUKA industrial robots, one mounted on the rail system bearing the client satellite mock up and other fixed on the ground bearing the docking system and RV sensors, each having 6 degree of freedom (DoF) are used to simulate the client and servicer satellite motion, respectively.

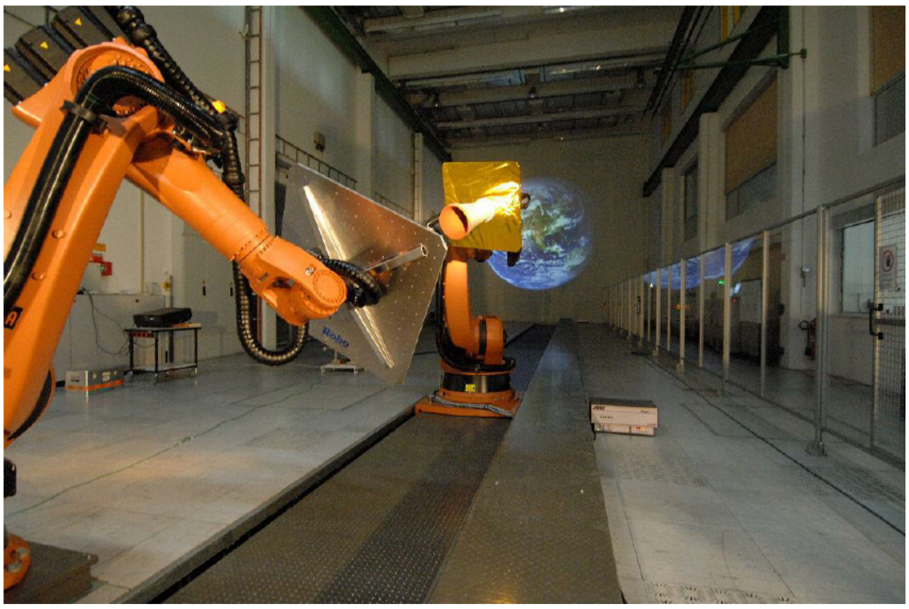


Figure 2.11: The EPOS facility (credits: DLR)

The facility has some limitations imposed by the configuration system, for instance the fixed length of the linear rail limits the possible simulation scenarios and thus the possible simulation setups with true-scale satellite models to 25m RV&D maneuvers. But the most critical limitations regard the simulation of satellite rotational motion, in particular the rotation around the roll-axis. [9]

The EPOS facility could be used also in closed loop applications, like DEOS (Deutsche Orbital Servicing Mission). For such “hardware in the loop” scenario the RV&D sensors and the robotic manipulator arm have to be mounted on one robot and a typical mock-up for the client satellite on the other robot. The sensors can measure the relative position and attitude of the client satellite and the on-board computer calculates on this basis the necessary thrusters or reaction wheel commands, that will

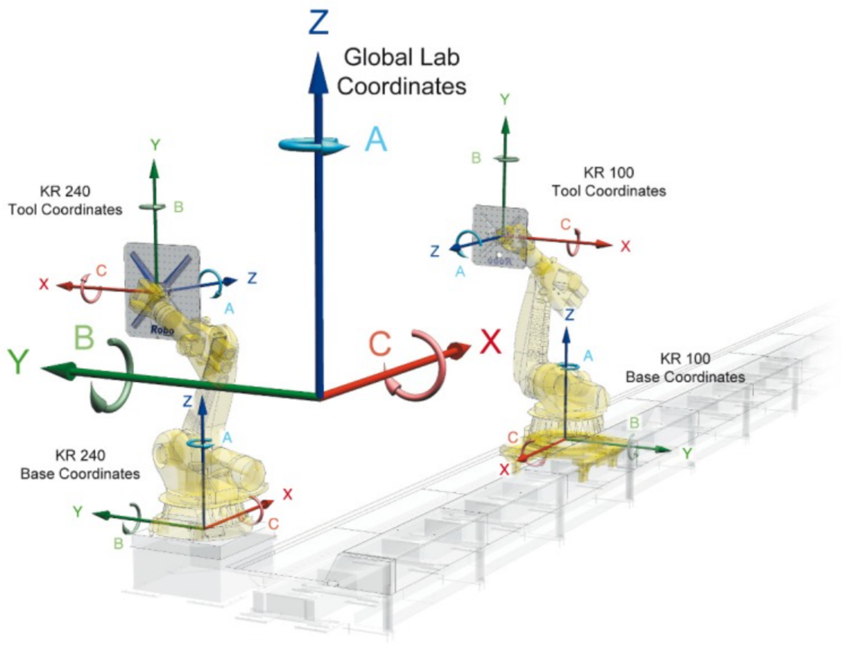


Figure 2.12: The EPOS reference coordinate systems (credits: DLR)

feed in a real time simulator. This dynamic simulator computes for the next sample an update of the position and attitude of the spacecraft based on control forces and torques, then these upgraded values of position and attitude will be commanded to the facility.

The new robotic experimental facility which was recently built at the DLR to support the development and experimental validation of such orbital servicing robots is called OOS-SIM (On-Orbit Servicing Simulation). First presented to the public in 2013, the facility allows reproducing a close-proximity scenario under realistic three-dimensional orbital dynamics conditions. It is a simulator for on-orbit servicing tasks such as assembly, maintenance and repair work on satellites that are in orbit around the earth. It also investigates the system's applicability for the removal of non-functioning orbiting target satellites.

Two large industrial robots hold the servicing unit (or the chaser satellite) and the target satellite and simulate their weightlessness. A sensitive lightweight robot arm with gripper is mounted on the servicing unit. The OOS-SIM system has two benefits. On the one hand, it is able to simulate the multi-body dynamics of free-flying robots in orbit without the influence of gravity. For this purpose, the movement of the chaser satellite is realized with an industrial robot with six degrees of freedom. The system also allows the simulation of contacts between the lightweight robot and the target satellite – with observation of the orbital dynamics. Force-torque sensors at the interfaces between the satellites and the industrial robots measure the contact forces and supply them to a real-time simulation of the multi-body dynamics of the free-flying robot and the target satellite ("hardware-in-the-loop" method). In this application domain the configuration

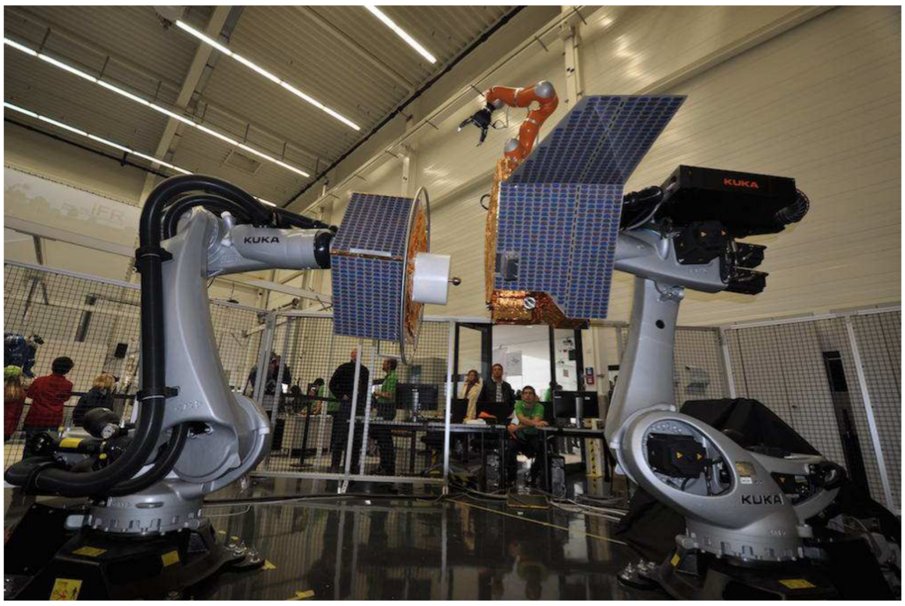


Figure 2.13: Overview of the OOS-SIM. On the left there is the Client satellite; on the right, the Servicer satellite and manipulator (credits: DLR)

of our facility represents a worldwide unique example. The second application of the facility is to provide a test and analysis platform for the development and validation of orbital robot control and image processing methods. With gravity-compensated dynamics, robot control requires suitable solutions to make the interaction between robot arm and satellite base motion controllable. Chaser satellite can be operated either with or without its actuators (e.g. thrusters of the attitude control system). Both strategies are current research topics at the Institute of Robotics and Mechatronics. The lightweight robot can also be commanded by an operator through teleoperation (or telepresence, with feedback of contact forces sensed by the robot arm). A stereo camera is mounted on the robot's gripper to support the robotic tasks (such as the capture process). A sun simulator and black curtains allow the simulation of different orbital lighting conditions under which image processing methods can be tested. The ultimate goal of this facility is to demonstrate the capture of a non-cooperative tumbling target satellite. The non-cooperative character is based on the assumption that the target satellite is not controllable and does not have any helpful visual features on its surface. One possible deployment scenario is Active Debris Removal, which removes target satellites from orbit by crashing them in a controlled manner so that they burn up in the Earth's atmosphere and fall into the sea. [20] The on-ground servicing simulation facility in Thales Alenia Space is the Rendezvous & Docking facility, an area that was built in the framework of the STEPS (Sistemi e Tecnologie per l'EsPlorazione Spaziale) program and co-financed by the Piedmont Region. Here the docking maneuver is simulated on a micrometric black granite floor where two vehicles (the chaser and the target) move floating over an air film of about 60μm in order to obtain a frictionless dynamic and have a testing environment able to simulate orbiting conditions.

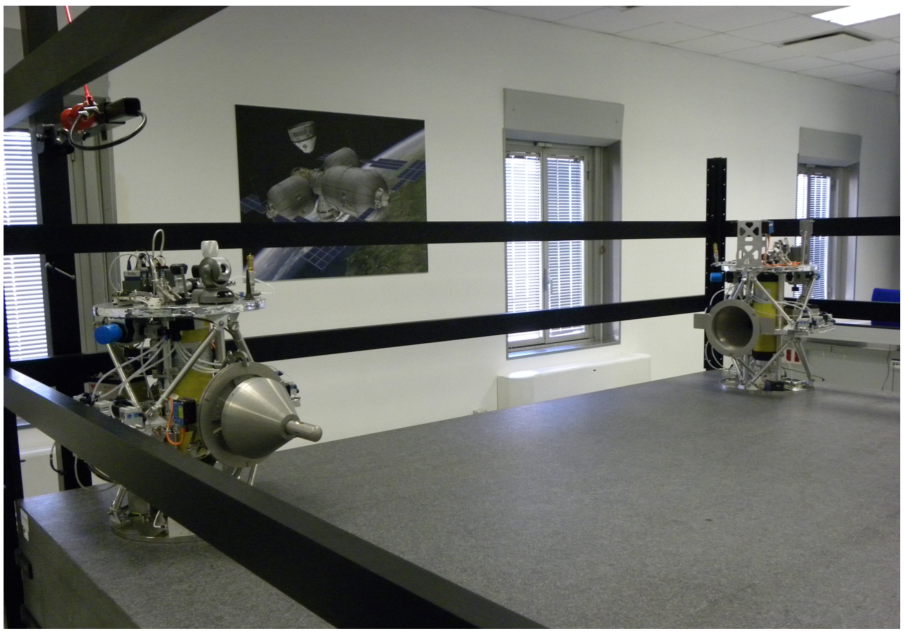


Figure 2.14: Rendezvous & Docking facility at Thales Alenia Space

This air film is fed by an on-board pneumatic system, also able to provide thrust and torque for attitude and position control. The chaser moves automatically toward the target and, once they are close enough, docking mechanism is turned on for final docking. The trajectory is corrected by Navigation and Control software on the basis of information from on-board sensor: thrusts and torques for controlling position, attitude and speed are automatically managed and through a Pulse Width Modulator actuators commands ON/OFF are generated. The area is also equipped with two robotic arms produced by Schunk (the Lightweight Arm 4P and the Lightweight Arm 3) to implement and test on the hardware the collaborative control algorithms between the arm and the chaser, necessary for the capture but more in general for servicing (e.g. refueling of propellant, repairing and/or replacement of parts, removing of space debris) of a target. The creation of a control system for the Schunk LWA 3 is the objective of this thesis.

Chapter 3

Schunk LWA3

3.1 Manipulator modelling and kinematics

A manipulator arm can be schematically represented as an open kinematic chain of rigid bodies, links, connected to each other by joints which constitute the degrees of mobility of the structure corresponding to the joint angles. Of the two extremities, one is constrained to a base and the other is connected to an end-effector.

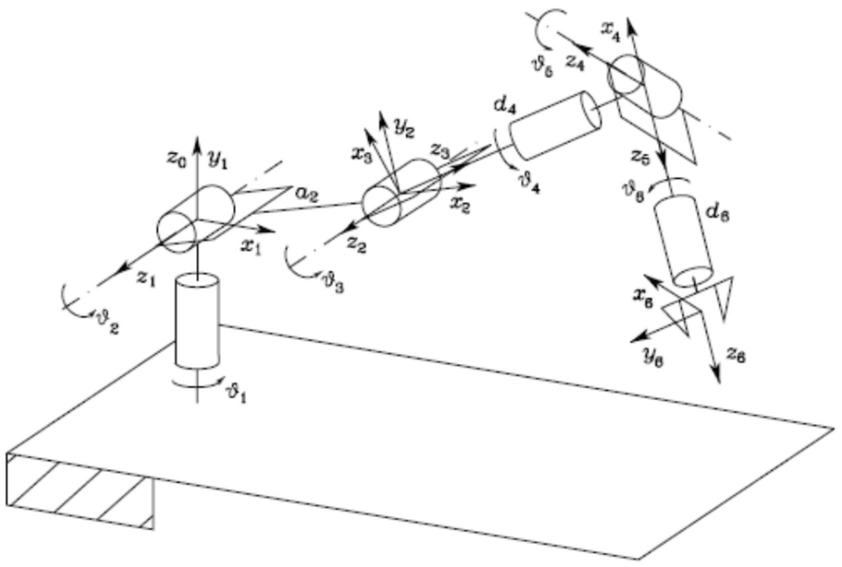


Figure 3.1: Schematization of a 6 DoF manipulator as open kinematics chain

The resulting motion of the structure is obtained by motion composition of each link with respect to the previous one. Therefore in order to manipulate an object in the space it is necessary to know the position and orientation of the end-effector, linked to the joint angles of manipulator by means of the direct kinematics equation that can be written in term of transformation matrix as:

$$\mathbf{T}(\mathbf{q}) = \begin{bmatrix} \mathbf{R}(\mathbf{q}) & \mathbf{p}(\mathbf{q}) \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{3.1}$$

where, considering a manipulator with $\mathbf{n}$ degrees of freedom, $\mathbf{q} = [\mathbf{q}_1 \ \mathbf{q}_2 \ ... \ \mathbf{q}_n]$ is the vector of joint angles, $\mathbf{0}^T = [0 \ 0 \ 0]$ $\mathbf{R}$ and $\mathbf{p}$ are the rotation matrix and position vector, that represent respectively the orientation and position of the end-effector with respect to the base frame. Therefore the matrix $\mathbf{T}(\mathbf{q})$ represent the transformation matrix from the base to the end-effector reference frame in homogeneous coordinates. Actually, the orientation of a rigid body can be represented in terms of so-called *minimal representation*, that can be extracted from the Rotation matrix as shown below. Consider two reference frames, one fixed in the space (x, y, z) and the other fixed with the body (x', y', z') like in Figure 3.2.

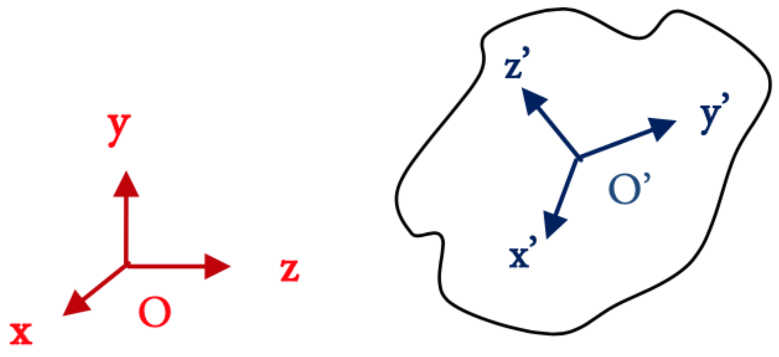


Figure 3.2: Inertial (red) and Body (blue) reference frame of a generic rigid body

The relative orientation between these frames, and hence the orientation of the body, can be described by means of the rotation matrix, R, whose nine components represent the direction cosines of the unit vectors of one frame relative to the other. Therefore the rotation matrix has a geometrical meaning, indeed it is possible to consider it as result of a sequence of three consecutive rotations each one around a coordinate axis, each of which describes the rotation about an axis in space needed to align the axes of the reference frame with the correspondent axes of the body frame. These matrix are called *elementary rotations* and their representations are shown in the following equations:

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & -sin\alpha \\ 0 & sin\alpha & cos\alpha \end{bmatrix}, \mathbf{R}_y(\alpha) = \begin{bmatrix} cos\alpha & 0 & sin\alpha \\ 0 & 1 & 0 \\ -sin\alpha & 0 & cos\alpha \end{bmatrix},$$

$$\mathbf{R}_z(\alpha) = \begin{bmatrix} cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These rotations are positive if they are made anticlockwise about their relative axis. The order in which the succession occurs is not commutative and therefore, must be specified. The components of a rotation matrix are linked by six relations due to its orthonormal property, therefore only three components are linearly independent and constitute the minimal representation of the body's orientation. This representation can be obtained by using a set of three angles (α, β, γ), the *Euler Angles.* There are twelve distinct sets of Euler angles, with regard to the sequence of possible elementary rotations; below the so-called ZYX representation is presented.

Consider (α, β, γ)the set of Euler angles. The overall rotation described by these angles is obtained as composition of the following elementary rotations:

- Rotate the reference frame by the angle γ about axis z; this rotation is described by the rotation matrix R_z (γ) which is defined in the equations above.
- Rotate the current frame by the angle β about axis y'; the corresponded rotation matrix is $R'_y(\beta)$.
- Rotate the current frame by the angle α about axis x"; the corresponded rotation matrix is $R''_x(\alpha)$. [7]

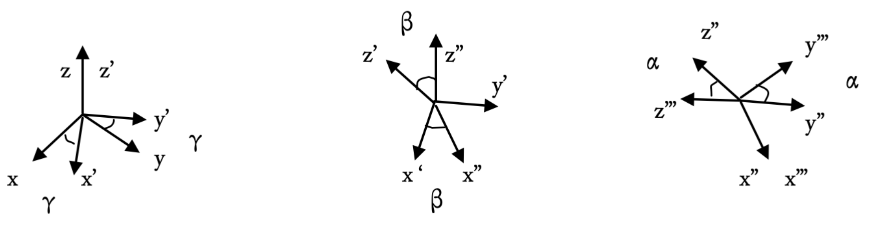


Figure 3.3: Representation of Euler angles ZYX

The resulting frame orientation is obtained by composition of rotations with respect to the current frame. The correspondent Rotation matrix is:

$$\mathbf{R}_{ZYX}(\alpha) = \begin{bmatrix} c_\gamma c_\beta & c_\gamma s_\beta s_\alpha - s_\gamma c_\alpha & c_\gamma s_\beta c_\alpha + s_\gamma s_\alpha \\ s_\gamma c_\beta & s_\gamma s_\beta s_\alpha + c_\gamma c_\alpha & s_\gamma s_\beta c_\alpha - c_\gamma s_\alpha \\ -s_\beta & c_\beta s_\alpha & c_\beta c_\alpha \end{bmatrix} \tag{3.2}$$

To solve the inverse problem, that is to compute the set of Euler angles from a given Rotation matrix

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

it is sufficient to compare the two matrices above, if β is in the range (-$\pi/2$, $\pi/2$) the solution is:

$$\alpha = Atan2(r_{32}, r_{33})$$

$$\beta = Atan2(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2})$$

$$\gamma = Atan2(r_{21}, r_{11})$$

where Atan2(y, x) is a two argument arctangent function. The solution in this case is $\beta = \pi/2, 3\pi/2$ is:

$$\alpha = Atan2(-r_{32}, -r_{33})$$

$$\beta = Atan2(-r_{31}, -\sqrt{r_{11}^2 + r_{21}^2})$$

$$\gamma = Atan2(-r_{21}, -r_{11})$$

Both of these solution degenerate when $cos\beta = 0$, i.e. $\beta = \pm\pi/2$; in this case only the sum or the difference of α and β can be computed and singularities can occur. The same rotation matrix expressed in (3.2) is obtained using the RPY (Roll-Pitch-Yaw) Angles representation in which each of the three rotations, in order $R_x(\psi)$, $R_y(\theta)$ and $R_z(\phi)$, takes place about an axis in the fixed reference frame.

An alternative representation of the orientation between two frames can be obtained in terms of a rotation angle, θ, about a unit vector, **r** , the *Angle/Axis representation.* The corresponding Rotation matrix is:

$$R(\theta, \mathbf{r}) = \mathbf{r}\mathbf{r}^T + cos\theta(I - rr^T) + sin\theta S(r)$$

where the $\mathbf{S}(\mathbf{r})$ matrix is a skew-symmetric dyad. It is clear that the rotation (θ, **r**) cannot be distinguished from (-θ,-**r**), being represented by the same matrix R. On the other hand in the case sin $\theta = 0$, i.e. $\theta = 0$ or π, can occur singularities easily verifiable considering the solution of the inverse problem.

$$\theta = cos^{-1}(\tfrac{r_{11}+r_{22}+r_{33}-1}{2})$$

$$r = \tfrac{1}{sin\theta}\begin{bmatrix} r_{32}-r_{23} \\ r_{13}-r_{31} \\ r_{21}-r_{12} \end{bmatrix}$$

These drawbacks can be overcome by a four-parameter representation, called *Euler parameters*, that in terms of equivalent axis $\mathbf{r} = [r_x r_y r_z]^T$ and angle θ are given by:

$$\varepsilon_1 = r_x sin\theta/2$$

$$\varepsilon_2 = r_y sin\theta/2$$

$$\varepsilon_3 = r_z sin\theta/2$$

$$\eta = cos\theta/2$$

These four components are not independent, indeed squaring each component and summing them, the result is always 1, therefore, an orientation might be visualized as a point on a unit hypersphere in four-dimensional space. The Euler parameters can be considered as (3x1) vector ε plus a scalar η or a (4x1) vector $\mathbf{Q} = \varepsilon,\, \eta$. In this last case it is called a *Unit quaternion.* [4] The equivalent Rotation matrix is defined as:

$$R(\varepsilon,\eta) = (\eta^2 - \varepsilon^T\varepsilon)I + 2\varepsilon\varepsilon^2 - 2\eta S(\varepsilon)$$

The equivalent Euler parameters, given a Rotation matrix, are:

$$\varepsilon_1^2 = 1/4(r_{11} - r_{22} - r_{33} + 1)$$

$$\varepsilon_2^2 = 1/4(-r_{11} + r_{22} - r_{33} + 1)$$

$$\varepsilon_3^2 = 1/4 - (r_{11} - r_{22} + r_{33} + 1)$$

$$\eta^2 = 1/4(r_{11} + r_{22} + r_{33} + 1)$$

In order to avoid numeric singularities it is sufficient to select the maximum absolute value component, assign to it the positive sign and derive the others accordingly. The different orientation representations are related to the angular velocity of the body through the following kinematics relations:

- for the rotation matrix $\dot{R} = S(\boldsymbol{w})R$
- for the Euler angles $\boldsymbol{w} = T(\phi)\dot{\phi}$
- for angle/axis parameters $\dot{\theta} = \mathbf{r}^T\boldsymbol{w}$

- for the unit quaternion $\dot{\mathbf{Q}} = 1/2(\mathbf{Q}\Omega)$ [2]

where the vector $\boldsymbol{w} \in \mathbf{R}^{3x1}$ denotes the angular velocity of the body frame with respect to the reference frame; the $\phi = [\alpha, \beta, \gamma]^T$ is the Euler angles vector and the Ω is the skew-symmetric matrix linked to the angular velocity. Representing the orientation with the minimal representation, using for instance the Euler Angles, it is possible to describe the manipulator posture by means of a (mx1) vector, with $m \leq n$, called state vector.

$$\mathbf{x} = \begin{bmatrix} \mathbf{p} \\ \phi \end{bmatrix} \tag{3.3}$$

where $\mathbf{p}$ describes the end-effector position, ϕ its orientation and m the DoF of the manipulator in the task space. This vector is defined in the space in which the manipulator task is specified, the so-called *operational space*, whereas in the *joint space* (configuration space) we refer to the manipulator posture with the (nx1) joint vector, $\boldsymbol{q}$, mentioned before. According to this notation the direct kinematic relationship can be written in another form than (3.1) as:

$$\mathbf{x} = k(\mathbf{q}) \tag{3.4}$$

where k $(\cdot)$ is generally a nonlinear vector function that allows the computation of the operational space variables from the knowledge of these in the joint space. Deriving with respect to the time the equation (3.3) we obtain the relationship that allows to express the end-effector linear, $\dot{\mathbf{p}}$, and angular $\boldsymbol{w}$, velocities as a function of joint velocities, $\dot{\boldsymbol{q}}$:

$$\dot{\boldsymbol{x}} = \begin{bmatrix} \dot{\mathbf{p}} \\ \boldsymbol{w} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J}_p(\mathbf{q}) \\ \boldsymbol{J}_o(\mathbf{q}) \end{bmatrix} = \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} \tag{3.5}$$

where $\boldsymbol{J}$ is the (mxn) *Jacobian* matrix of the manipulator.

Once understood how the position and the orientation of end-effector are affected by the manipulator's joints, in order to implement a control for the robotic arm it is necessary to define its dynamic model. This can be derived either using Lagrange's method based on energy balance or the Newton-Euler method based on force balance and written in the compact matricial form below.

$$\boldsymbol{H}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}_q(\boldsymbol{q}, \dot{\boldsymbol{q}})\dot{\boldsymbol{q}} + \boldsymbol{g}_q(\boldsymbol{q}) + \tau_h = \tau \tag{3.6}$$

where $\boldsymbol{q}$, $\dot{\boldsymbol{q}}$, $\ddot{\boldsymbol{q}} \in R^{nx1}$ are the joint, joint velocity, joint acceleration vector; $\boldsymbol{H}_q \in R^{nxn}$ is the Inertia matrix of the manipulator arm; $\boldsymbol{C}_q \in R^{nx1}$ is the nonlinear dependent velocity term, that takes into account the centrifugal and Coriolis forces; $\boldsymbol{g}_q \in R^{nx1}$ is the vector of moments generated by the presence of gravity; $\tau_h \in R^{nx1}$ is the vector of joint-space forces and moment exerted by the end-effector on the environment; $\tau \in R^{nx1}$ is the actuation torques vector applied to the joints. In the model of the equation (3.7) the viscous and static friction forces have been neglected, thus considering only the ideal behavior of the manipulator. Such nonlinear characteristics will be compensated during the numerical simulations. Furthermore, it is worth noting that the dynamics equation of the manipulator (3.7) is defined in the joint space, denoted by the subscription $\boldsymbol{q}$. The joint space description of the manipulator tasks can only be used for a well-defined environment and task geometries, therefore it is generally more convenient to define the manipulator tasks in the operational space, especially when the manipulator needs to perform a variety of contact tasks in an unknown environment. For this reason, it is useful considering also the dynamics model in the operational space, expressed as:

$$\boldsymbol{H}(\boldsymbol{x})\ddot{\boldsymbol{x}} + \boldsymbol{C}_x(\boldsymbol{x}, \dot{\boldsymbol{x}})\dot{\boldsymbol{x}} + \boldsymbol{g}_x(\boldsymbol{x}) + \boldsymbol{F}_h = \boldsymbol{F} \tag{3.7}$$

where $\boldsymbol{x}$, $\dot{\boldsymbol{x}}$, $\ddot{\boldsymbol{x}} \in R^{mx1}$ are the state, state velocity and state acceleration vector; $\boldsymbol{H}_x \in R^{mxm}$, $\boldsymbol{C}_x \in R^{mx1}$, $\boldsymbol{g}_x \in R^{mx1}$, are the representation of the same Inertia matrix, nonlinear dependent velocity term and the vector of gravity forces expressed in the equation (3.7), respectively, with the only difference that are defined in the operational space, as denoted by the subscription $\boldsymbol{x}$; $\boldsymbol{F}_h \in R^{mx1}$ is the vector of forces and moment due to environment contact and $\boldsymbol{F} \in R^{mx1}$ is the vector of forces and moment acting on the end-effector. The relationship between the operational and joint space parameters can be given by the following equation:

$$\begin{aligned}
\boldsymbol{H}_x &= (\boldsymbol{J}\boldsymbol{H}_q^{-1}\boldsymbol{J}^T)^{-1} \\
\boldsymbol{C}_x\dot{\boldsymbol{x}} &= \boldsymbol{H}_x\boldsymbol{J}\boldsymbol{H}_q^{-1}\boldsymbol{C}_q\dot{\boldsymbol{q}} - \boldsymbol{H}_x\dot{\boldsymbol{J}}\dot{\boldsymbol{q}} \\
\boldsymbol{g}_x &= \boldsymbol{H}_x\boldsymbol{J}\boldsymbol{H}_q^{-1}\boldsymbol{g}_q
\end{aligned} \tag{3.8}$$

Easy derivable taking in consideration the fundamental link between the operational and joint variables of position and force, the first expressed by the equation (3.5) and the second expressed by $\tau = \boldsymbol{J}^T\mathbf{F}$, respectively.

3.2 Forward kinematics of the Schunk LWA 3

The forward kinematics determine the position and orientation of the end-effector of a robotic arm when the joint angles values are given by mapping the joint space to the task space. The manipulator considered in this thesis has seven revolute joints rotating around fixed axis on previous links. The 7-DoF manipulator has a similar structure to the human arm. The arm has 7 revolute joints arranged to form the shoulder, elbow, and wrist portions as shown in Figure 3.4. Forward kinematics problem has no complexity to derive the equations comparing to the inverse kinematics because it is a straightforward problem. Hence, there is always a forward kinematics solution of a manipulator. The Denavit-Hartenberg method is the most common method for describing the manipulator kinematics that led to derive the forward kinematics equations.

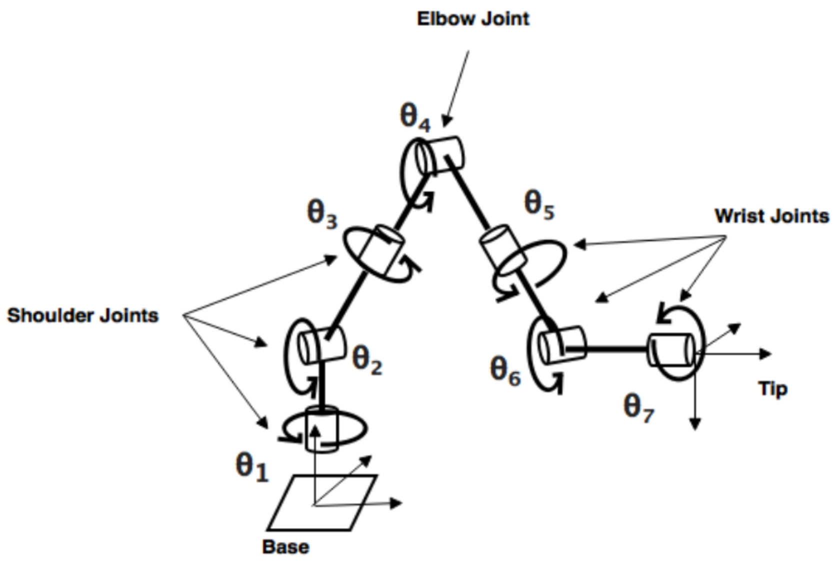


Figure 3.4: The manipulator structure

3.2.1 Denavit-Hartenberg representation

The forward kinematics of the arm can be developed by a systemic procedure using a combination of conventions to determine and analyze the successive effect of the consecutive joint motions to finally place the end-effector at a specific position and orientation. These conversions allow to simplify the intricacy of the forward kinematics analysis in the complex arm with higher number of axes. The solution of the forward kinematics problem can be accomplished by calculating the transformation between the fixed base frame and the end-effector frame. The Denavit-Hartenberg procedure is the convention generally used to calculate the transformation between the frames [1]. Each joint in the manipulator connects two links: every joint *i* connects the link *i-1* to the link *i*. When joint *i* is actuated the link *i* moves. Link 0 is always fixed. In Denavit-Hartenberg method, the homogeneous transformation matrix is represented by a product of four basic transformations: two translations and two rotations parametrized by the four

Denavit-Hartenberg parameters as equation (3.9) illustrates.

$$T_i^{i-1} = Rot_{z_i}(\theta_i)Trans_{z_i}(d_i)Trans_{x_i}(a_i)Rot_{x_i}(\alpha_i) =$$

$$= \begin{bmatrix} cos(\theta_i) & -sin(\theta_i) & 0 & 0 \\ sin(\theta_i) & -cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha_i) & -sin(\alpha_i) & 0 \\ 0 & sin(\alpha_i) & cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The final result of this matrix multiplication is:

$$T_i^{i-1} = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i)cos(\alpha_i) & sin(\theta_i)sin(\alpha_i) & a_i cos(\theta_i) \\ sin(\theta_i) & cos(\theta_i)cos(\alpha_i) & -cos(\theta_i)sin(\alpha_i) & a_i sin(\theta_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

where θ_i is the joint angle, d_i is the link offset, a_i is the link length and α_i is the link twist. The matrix T_i^{i-1} is a function of a single variable θ_i because the manipulator in this study has only revolute joints, the remaining three parameters are constant for each link. The joint angle θ_i is the angle measured about z_{i-1} between x_{i-1} and x_i. The link offset d_i is the distance from o_{i-1} to the intersection of the x_i and z_{i-1} axes along z_{i-1}. The link length a_i is the distance from o_i to the intersection of the x_i and z_{i-1} axes along x_i. The link twist α_i is the angle measured about x_i between z_{i-1} and z_i. The homogeneous transformation matrix T_i^{i-1} expresses the position of joint frame center o_i and orientation of the attached frame i with respect to the previous joint frame *i1*. So the transformation from the base to the end-effector is given by

$$T_7^0 = T_1^0 T_2^1 T_3^2 T_4^3 T_5^4 T_6^5 T_7^6 \tag{3.10}$$

Where the transformation matrix T_i^{i-1} is given by the rotation matrix R_i^{i-1} and the translation vector p_i^{i-1} as shown in equation (3.11).

$$T_i^{i-1} = \begin{bmatrix} \mathbf{R}_i^{i-1} & \mathbf{p}_i^{i-1} \\ \mathbf{0} & 1 \end{bmatrix} \tag{3.11}$$

The rotation matrix $\mathbf{R}_i^{i-1}$ in Denavit-Hartenberg method can be combination of rotations around z and x. The corresponding rotation matrices are shown in equation (3.12). $\mathbf{p}_i^{i-1}$ is a 3 by 1 vector that represents coordinates of the position of the frame center (p_x, p_y and p_z).

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha_i) & -sin(\alpha_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) \end{bmatrix}, \mathbf{R}_z = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i) & 0 \\ sin(\theta_i) & cos(\theta_i) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

The assigned frames must satisfy two conditions of the Denavit-Hartenberg method. The first one is that the x_i axis is perpendicular to z_{i-1} and the second is that the axis x_i must also intersect z_{i-1}. Assigning the coordinate frames for a manipulator can be achieved by many different assigned ways but such that the Denavit-Hartenberg conditions are satisfied.

3.2.2 Coordinate frames assignment

The procedure of the Denavit-Hartenberg convention algorithm starts by assigning the coordinate frames to derive the forward kinematics equations of a manipulator. The first step in the process is the choice of the z axes for the 7 joints by assign z_i to be the axis of actuation for joint *i+1*. Thus, z_0 is the axis of actuation for joint 1, z_1 is the axis of actuation for joint 2, z_2 is the axis of actuation for joint 3, z_3 is the axis of actuation for joint 4, z_4 is the axis of actuation for joint 5, z_5 is the axis of actuation for joint 6, z_6 is the axis of actuation for joint 7, z_7 is the axis of defining an important direction for the end-effector. The base frame then is established by choosing the origin o_0 at any point on z_0 as shown in Figure 3.5, the x_0 and y_0 axes are conveniently chosen to form a right-hand frame. The origins from o_1 to o_7 are chosen to be the points of the intersection between z_1 and z_{i+1}. After that assigning the x axis from x_1 to x_7 where x_i must intersect and be perpendicular to the axis of z_{i-1} and z_i. The y_i axis established to form a right-hand frame, see Figure 3.5 below.

3.2.3 Forward kinematics equations

The forward kinematics equations can be obtained after assigning the coordinate frames and establishing the link parameters d_i, a_i, α_i and θ_i that form the D-H table. The link length a_i is distance along x_i from o_i to the intersection of the axis of x_i and the axis of z_{i-1}. From the assigned frames shown in Figure 3.5 there is no distance along any x axes from the origins to the point of intersection between the x_i and z_{i-1} so that the link length is zero in all the frames in the D-H table. The link offset d_i is the distance along the axis z_{i-1} from the origin o_{i-1} to the intersection of the axis x_i and the axis of z_{i-1}. The assigned frames in Figure 3.5 shows that there are l_b^s along z_0, l_s^e along z_2, l_e^w along z_4 and l_w^t along z_6 while there is no any link offsets along z_1, z_3 and z_5. The link twist α_i is the angle between z_{i-1} and z_i measured about x_i. Figure 3.5 illustrates that about x_1 there is an angle of $-\pi/2$ (rad) between the axis of z_0 and z_1, same angle is around the axes of x_3 and x_5. The angle about x_2 between the axes of z_1 and z_2 is $\pi/2$ (rad), same angle around the axes of x_4 and x_6. The figure shows that there is no angle around x_7 between the axis of z_6 and z_7. The Link angle θ_i is the angle between x_{i-1} and x_i measured about z_{i-1} since all the joints are revolute joints so the joint angles from θ_1 to θ_7 are variables. The Denavit-Hartenberg parameters are

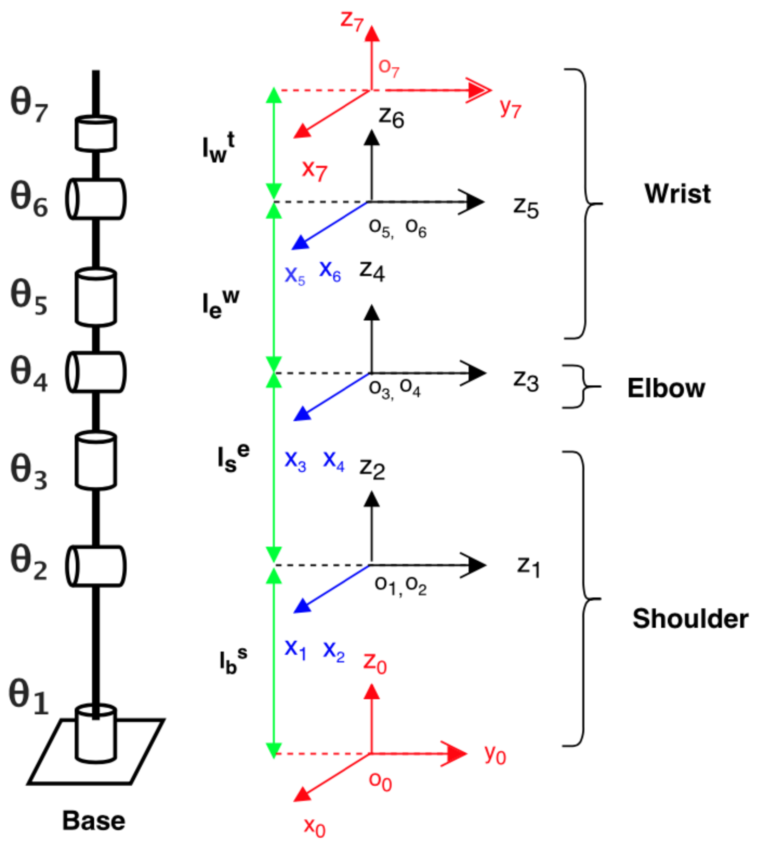


Figure 3.5: The coordinate frames assigned to the manipulator.

shown in Table 3.1

Frames	Link angle [rad]	x_i^{i-1} [mm]	y_i^{i-1} [mm]	z_i^{i-1} [mm]
$R_0 \to R_1$	θ_1	0	0	133.1
$R_1 \to R_2$	θ_2	0	-13.1	-166.9
$R_2 \to R_3$	θ_3	0	182.6	13.1
$R_3 \to R_4$	θ_4	0	-13.6	-145.4
$R_4 \to R_5$	θ_5	0	159.1	13.6
$R_5 \to R_6$	θ_6	0.565	-10.785	-117.4
$R_6 \to R_7$	θ_7	-0.571	141.6	10.885
$R_7 \to R_{ee}$	0	0	0	-36.9

Table 3.1: Denavit-Hartenberg parameters

The homogeneous transformation matrices T_7^0 are computed by substituting the parameters above in equation (3.10) for each joint. The position and orientation transformation matrix of the end-effector relative to basis coordinate system is shown in equation (3.13).

$$T_7^0 = T_1^0 T_2^1 T_3^2 T_4^3 T_5^4 T_6^5 T_7^6 = \begin{bmatrix} R_{11} & R_{12} & R_{13} & p_x \\ R_{21} & R_{22} & R_{23} & p_y \\ R_{31} & R_{32} & R_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

3.3 Schunk LWA3 data and setup

The Schunk Lightweight Arm 3 is, as mentioned before, a 7 degrees-of-freedom manipulator. The arm is based on the servo-electric swivel units PRL (Positive Rotational Lock) with integrated motor controller units and a through-hole for cable feed- through. The combination of a high compact performance, and new materials for the connection technology allows the doubling of the payload to nominal 5 kg. The standard design of the LWA 3 is available as a 7-axes system.

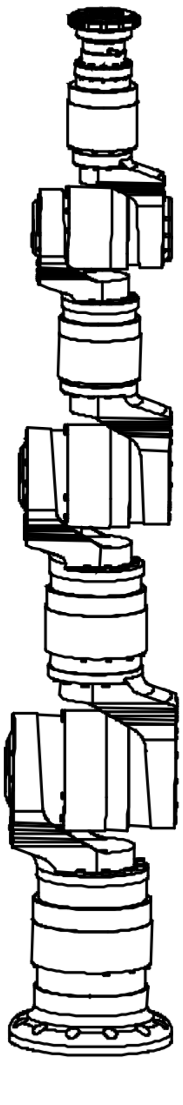


Figure 3.6: Design of Schunk Lightweight Arm 3

The open software-architecture enables the connection and operation with any type of end-effectors to the servo-electric "wrist" of the arm, e.g.:

- Servo-electric 2-finger parallel grippers PG;
- Schunk Anthropomorphic Hand SAH;
- Schunk Dexterous Hand (SDH), which was available at the Thales Alenia Space RV&D facility and can be seen in Figure 3.7 on the left of the robotic arm.

The force-torque sensor FTC may be used for the power feedback. All joint functionalities are stored as macros directly in the respective PRL and may be actuated from a superposed control level.

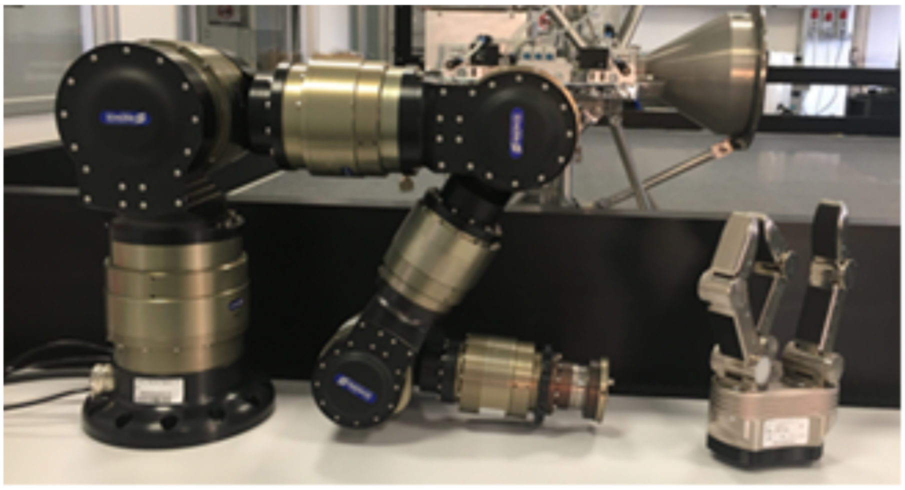


Figure 3.7: The LWA 3 and the SDH

The technical data of the LWA 3 are:

- repeatability of 1 mm;
- power supply is 24 VDC, maximum current of 15 A (for safety reasons, the current deployed during tests in the RV&D facility was never higher than 2 A);
- maximum payload of 5 kg that can be increased as described later [14].

The PRL modules are not directly connected to the power supply and to the workstation used to control them, as can be seen in Figure 3.8.

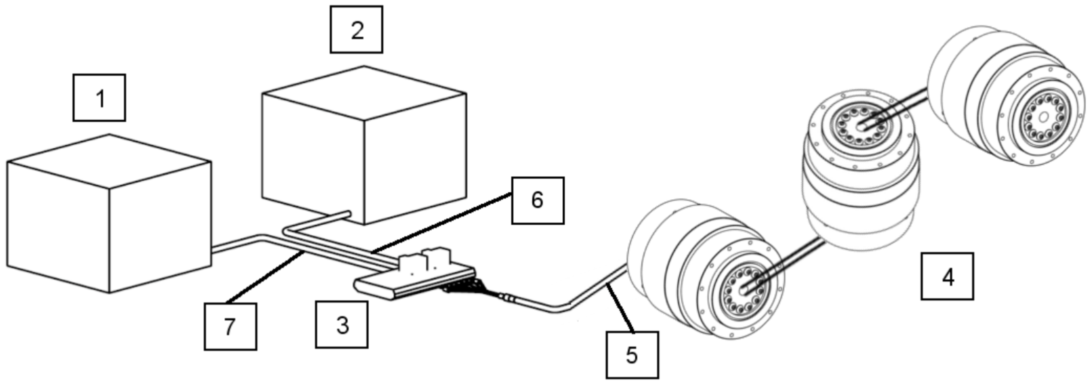


Figure 3.8: Circuit diagram for power supply (courtesy of Schunk)

1. Power supply
2. PC, SPS or suitable control
3. PowerCube Terminal Block
4. PowerCube-Module 1 to n
5. Bus cable from Terminal Block to module
6. Cable for comunication
7. Cable for power supply

The terminal block is connected to the power supply with pins for logic supply (one for 0 VDC and one for 24 VDC) and two pins come out of the block and go to the base of the robotic arm. In a similar way, the PC for the control and communication has a CAN cable with three wires (for CAN high, CAN low and Shield) that go to their respective pins of the terminal block which then sends the signals directly to the arm with the CAN bus. Figures 3.9 and 3.10 show the schematics of the terminal block with the pins and an actual photo, respectively.

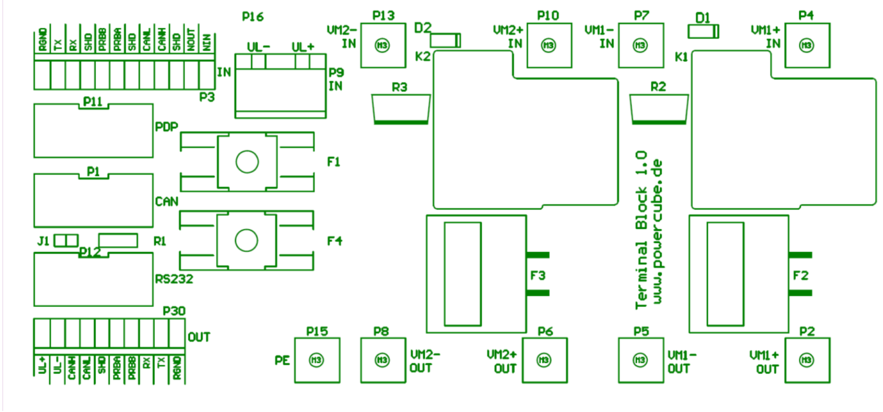


Figure 3.9: Schematics of the terminal block

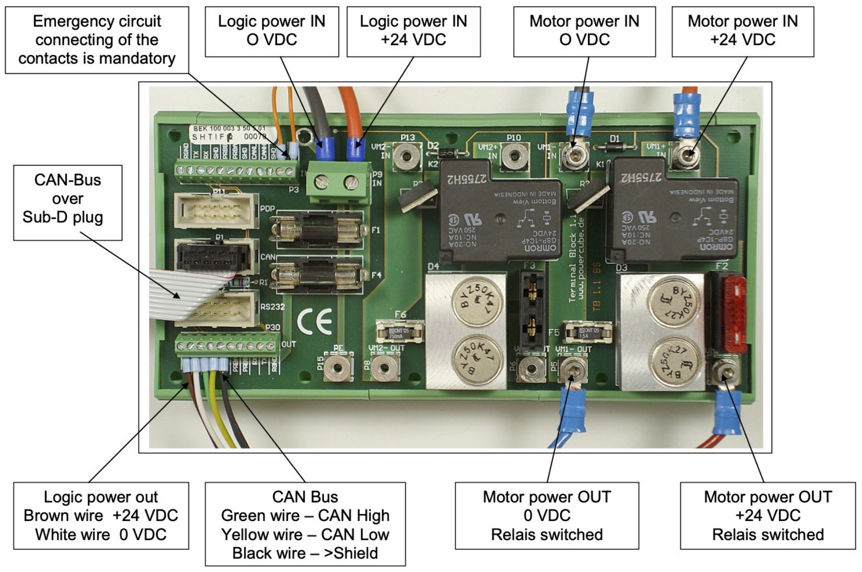

Emergency circuit connecting of the contacts is mandatory
Logic power IN O VDC
Logic power IN +24 VDC
Motor power IN O VDC
Motor power IN +24 VDC
CAN-Bus over Sub-D plug
Logic power out Brown wire +24 VDC White wire 0 VDC
CAN Bus Green wire – CAN High Yellow wire – CAN Low Black wire – >Shield
Motor power OUT 0 VDC Relais switched
Motor power OUT +24 VDC Relais switched

Figure 3.10: Terminal block of Schunk LWA 3 (courtesy of Schunk)

The advantages that the structure and design of LWA 3 brings are:

- Suitable for mobile applications, thanks to low energy consumption at 24 V DC;
- High torque, speed and repeat accuracy for rapid acceleration;
- Short cycle times and high process stability;
- Complete integration of control, regulator and power electronics does not require a control cabinet;
- Compact quick-change system to quickly and easily mount grippers and tools;
- Internal cabling with free wires Expandable without disruptive cables;
- High power density and extremely compact Lightweight construction and new design provide a weight-load capacity ratio of 2:1;
- maneuverable and flexible, as 7 axes provide redundancy during challenging handling tasks

The field of applications of the LWA 3 lightweight arm includes various sectors of robotics: inspection systems, service robotics, human-machine and human-human interactions are just some examples. When combined with different types of end-effectors, the LWA 3 can be used in tactile, sensor or camera based inspection systems or even with mobile platforms, such as light transport, monitoring and maintenance robots.

The new servo-electric swivel units PRL of the PowerCube series with integrated Motor-Controller unit and continuous center bore for the cable feed-through form its base. The combination of high power density and light materials for connecting technology enable twice the payload. Each PRL module is composed by a rotor and a stator linked to a plaque that connects every joint to the previous one. The separation of the two parts coincides with the line above the Schunk logo, visible in Figure 3.11.

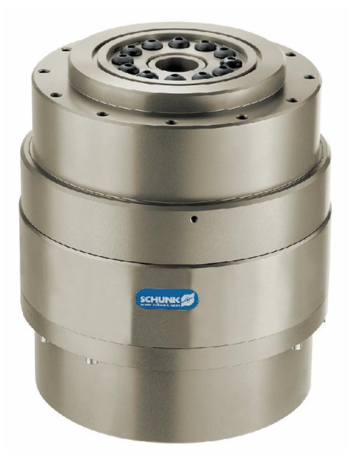


Figure 3.11: PRL module (courtesy of Schunk)

The seven PRL modules that make up the stucture of the arm come in different dimensions: 2x PRL 120, 2x PRL 100, 2x PRL 80, 2x PRL 60. [14] These motors are assembled together and also internally connected by cables routing through the hollow shaft integrated into the PRL-modules.

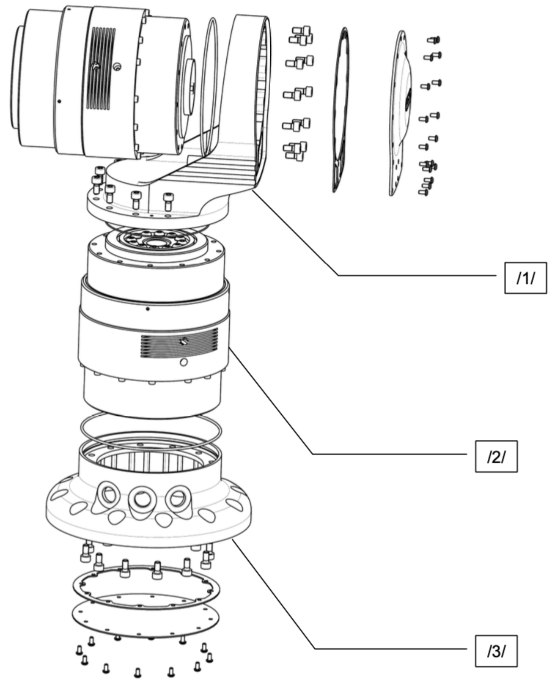


Figure 3.12: Example of mounting PRL modules (courtesy of Schunk)

1. Connection element 120
2. PRL 120
3. Pedestal PRL 120

The Rotary Module PRL meet the demands for reconfigurable, modular robot structures since they are consistently implemented. Due to the use of light but also very stable materials, the compact swivel units achieve a weight-payload ratio which is better than 2:1. The modules are equipped with an integrated power supply, control options, and universal communication interfaces. The individual PRL modules can be freely and flexibly assembled to an individual light-weight arm. Due to this combination of this flexible robotics solution, five degrees of freedom are already successfully and reliably integrated. There is an open software architecture that allows the control of the axes, however the PRL modules communication is based on the PowerCube software. The PowerCube protocol can be used via three different types of bus: CAN bus, RS232 and Profibus. All the specifics of the PowerCube software and the CAN communication will be the focus of the next chapter.

Chapter 4

The CAN protocol

The Controller Area Network - CAN - is a communication protocol developed by R. Bosch GmbH at the beginning of the 1980s as a working method for enabling robust serial communication, with a focus on the automotive area. In 1990 Mercedes-Benz was the first manufacturer to use this protocol in one of their flagship model, the S-class. In 1993 it became an international standard with the release of ISO 11898, CAN 2.0A and 2.0B, while in 1994 other CAN-related higher level protocols have been standardized, such as CANopen and DeviceNet. In 1997 24 millions CAN interfaces were produced; two years later there were already more than three times as many, and today almost every road vehicle uses this protocol [15]. The reason why CAN became so popular is that it provides an inexpensive and durable way to make the numerous micro-controllers present in a car communicate with one another, as it can reduce a lot the amount of cables needed. Today the CAN networks are also used in space and aerospace applications, robotics, railed transportation, hospitals and even coffee machines [12].

4.1 ISO 11898

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

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

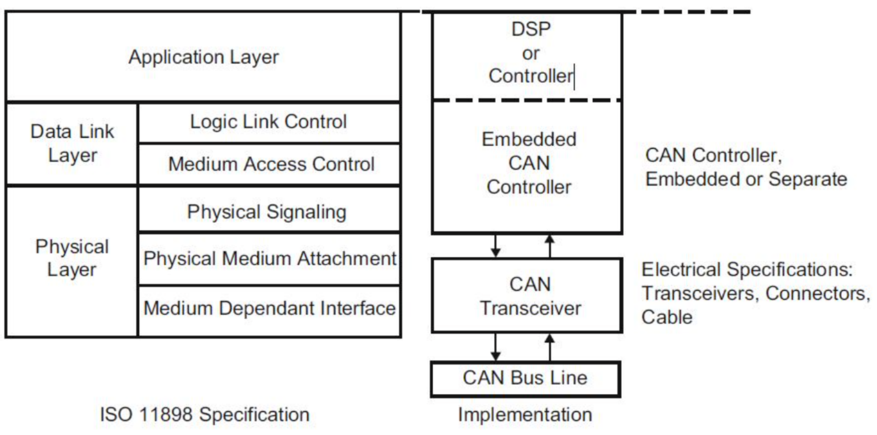


Figure 4.1: ISO 11898 standard architecture for CAN networks

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

A simple representation of the layers that form this communication structure is the one shown in figure 4.1. The **Application layer** provides high level communication functions that can be implemented by a software developer or handled by a higher level protocol such as:

- **CAL** (CAN Application Layer): originally developed by Philips Medical Systems, is an application-independent layer that is now maintained by the CAN in automation (CiA) user group.
- **CANopen**: built on top of CAL, uses some of its services and communication protocols. With this protocol every node in the network is associated to an Object Dictionary (OD) where all the parameters that describes the device and its behavior are contained. CANopen is now maintained by the CAN in automation (CiA) user group too.
- **DeviceNet**: developed by American company Allen-Bradley (now owned by Rockwell Automation), adapts the technology from the Common Industrial Protocol and exploits CAN, making it low-cost and robust compared to the traditional RS-485 based protocols.

The **Datalink layer** has the task of transferring the messages from a node to all the other ones. It is divided in Logic Link Control layer (LLC) and Medium Access Control layer (MAC) and handles bit stuffing and error control, waiting for acknowledgement from the receivers after a message is sent. Figure 4.2 shows how nodes are connected in

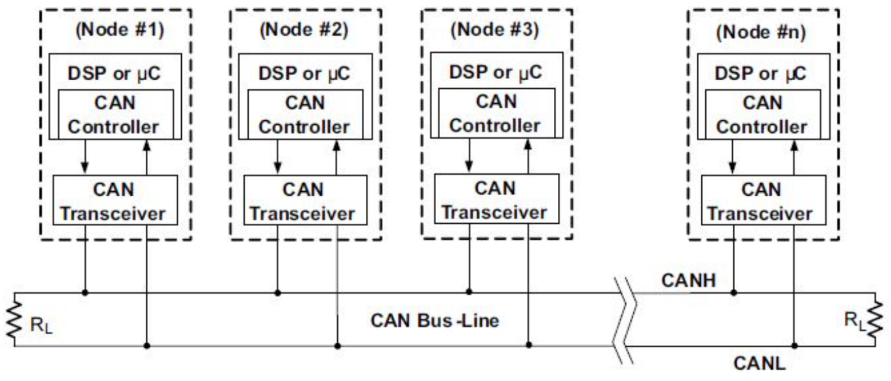


Figure 4.2: Example of the physical connection in a high-speed CAN network

a typical CAN network. The Physical layer implements physical signaling, bit encoding and decoding, bit transmitting and synchronization [12]. Different kinds of physical layer are used to satisfy the system specification both from a cost or performance point of view, the most common are:

- High-speed CAN hardware: it's the most used physical layer, with two wires that allow communication at 1 Mbit/s rate. Used most for anti-lock brake systems, engine control modules and emission systems, it is also known as CAN-C (ISO 11898-2).
- Low-speed/fault-tolerant CAN hardware: also known as CAN-B (ISO 11898-3), also uses two wires for the communication, but up to 1 Mbit/s transfer rate. In the automotive field it is primarily used for comfort devices.
- Single-wire CAN hardware: using just a single wire for the communication, the transfer rate is limited to 33.3 kbit/s (88.3 kbit/s in high-speed mode). Also known as CAN-A, within an automobile it is used just for those devices that do not have any performance requirement, such as mirror adjusters.

4.2 Main features

Being such a robust, fast, safe and widely used protocol, CAN is inevitably complex. The communication is asynchronous, half duplex (usually differential signaling), up to 1 Mbit/s. The number of nodes is virtually unlimited, for this reason they do not have an address, but filter the data on the bus to determine if it's useful or not, and master-slave designation is not used. Logic values are "dominant" (low - 0) or "recessive" (high - 1), where dominant overrides recessive, so any node can start a message and an arbitration process is applied, without any loss of time or data. It is also worth noting that there is a high level of data security because a node that recognises that it is faulty after error checking, can disconnect itself from the network. Since all the nodes are

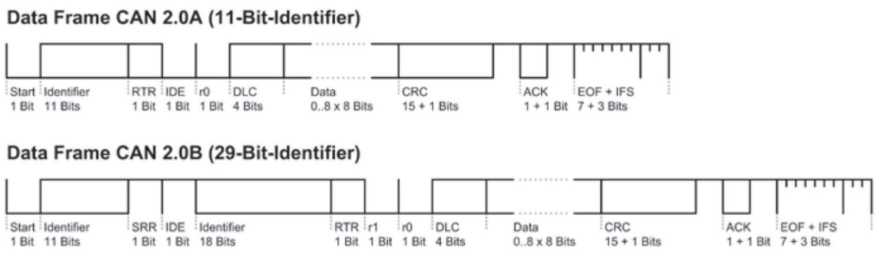


Figure 4.3: Standard structure of a CAN message

equal, it is impossible to simply send "only" the data on the bus, so the communication is based on **messages** or **frames** that allows to carry much more information. Figure 4.3 shows how typically a CAN message is organized for CAN 2.0A (standard) and can 2.0B (extended). Considering just the 11-Bit-Identifier frame, that is the one used for the Schunk Lightweight Arm 3 and the SDH, the message is divided in the following parts [15]:

- **Start of Frame (SOF) = 1 bit (low, dominant)**: always used to start a frame, the falling edge synchronizes all network nodes.
- **Arbitration Field = 12 bit**: ID and RTR combined make the Arbitration Field. Each message has its own **ID = 11 bit** that basically is the "name" of the message (CAN 2.0B has a 29 bit ID), the lower the identifier the higher the priority. **Remote Transmission Request (RTR) = 1 bit** is the last bit of the arbitration field. If RTR is high=recessive it means that the message is asking for data (*data request frame*), else if RTR is low the message contains the data itself (*data frame*) or does not have to contain data because it only triggers some operations that do not need it. These two messages have the same ID but different RTR so are different.
- **Control Field = 6 bit**: is composed by the **Identifier Extension Flag (IDE)** bit that indicates that the ID is completed, the r0 bit that is reserved and the **Data Length Code (DLC)** 4 bits that indicates how many bytes of data is the message carrying.
- **Data Field = 0–8 bytes of data**: contains the actual data of the message.
- **Cyclic Redundancy Check (CRC) Field = 16 bits**: the first 15 bits (CRC sequence) are only used for fault detection, adding redundant check bits at the transmission end, then these bits are recomputed at the receiver end and tested again, if there is a misalignment a CRC error occurred. The CRC field is closed by that last bit, called CRC delimiter (high).
- **Acknowledge Field = 2 bits**: all the nodes that have recognized the message as correct will send a dominant level in the ACK slot, if no node sends the

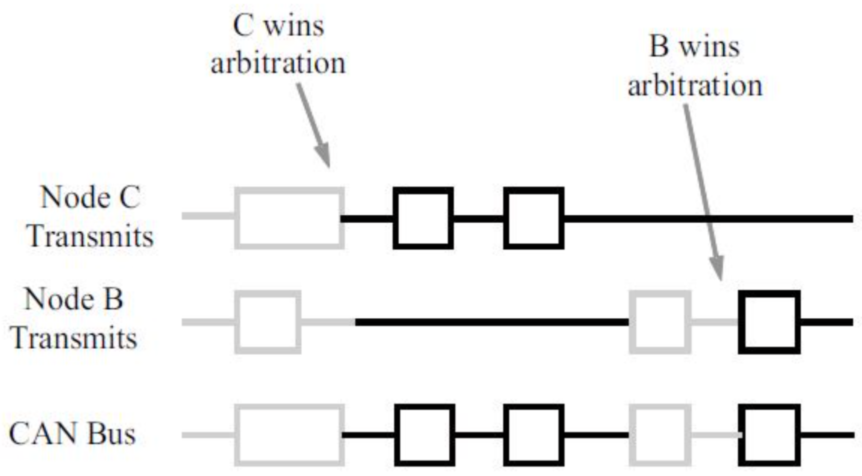


Figure 4.4: Arbitration example scheme on a CAN bus

low level an ACK error occurred. The acknowledge field is closed by the ACK delimiter (high).

- **End of Frame (EOF) = 7 bit**: indicates the end of the data frame (usually all recessive).
- **Inter Frame Space (IFS) = 3 bit (high)**: separates the frame from the following one. The time for this operation is used inside the node to transfer the message from the controller to a receive buffer or to transfer a message from a transmit buffer to the controller.

It is noticeable that inside the message structure a good part is dedicated to **error detection**, another important feature of the CAN protocol, with the CRC field and the ACK field. Another way an error active node or the transmit one can report an error is by transmitting an error frame (usually six consecutive low level bits), a particular message that violates the rules of bit stuffing and frame format, causing all the other nodes to send an error frame as well [6], and after that bus activity returns to normal. On the contrary, also an error passive node can indicate that they have detected an error by sending an error-passive flag (usually 14 recessive bits), so if the fault is not detected by an error active node or the transmit one the message will continue the transmission [19]. Generally hundreds if not thousand of nodes are connected and try to transmit messages in a single CAN network, and since there is no master-slave designation, **bit-wise arbitration** is used to give priority to a certain frame instead of another one. Figure 4.4 shows how it works. Node B and Node C start transmitting on the CAN bus at the same moment but after some identical bits, B tries to put on the bus a recessive (high) bit while C transmits a dominant (low) one. So B loses arbitration and stops transmitting, C finishes its message and after that B wins arbitration again, completing its frame. This functionality is part of the

ISO 11898 physical layer, which means that it is contained entirely within the CAN controller and is completely transparent to a CAN user [6].

4.3 Schunk LWA3 CAN communication

Like it was anticipated in paragraph 3.3, the PRL modules can communicate with one another and with the master PC through different standards: RS232, Profibus or CAN bus. In the case of the Schunk LWA 3 at Thales Alenia Space, the communication method that was used was the CAN bus. The hollow shaft drive allows a protected installation of cables and hoses inside the arm up to the "wrist". No cables are visible, and there are no interfering contours. An example of assembling inside one of the modules is the one in Figure 4.5. In the same figure it is possible to see 120 Ω resistance whose presence is necessary to terminate the CAN bus on both ends. The final resistance that completes the "chain" of termination is welded on the terminal block.

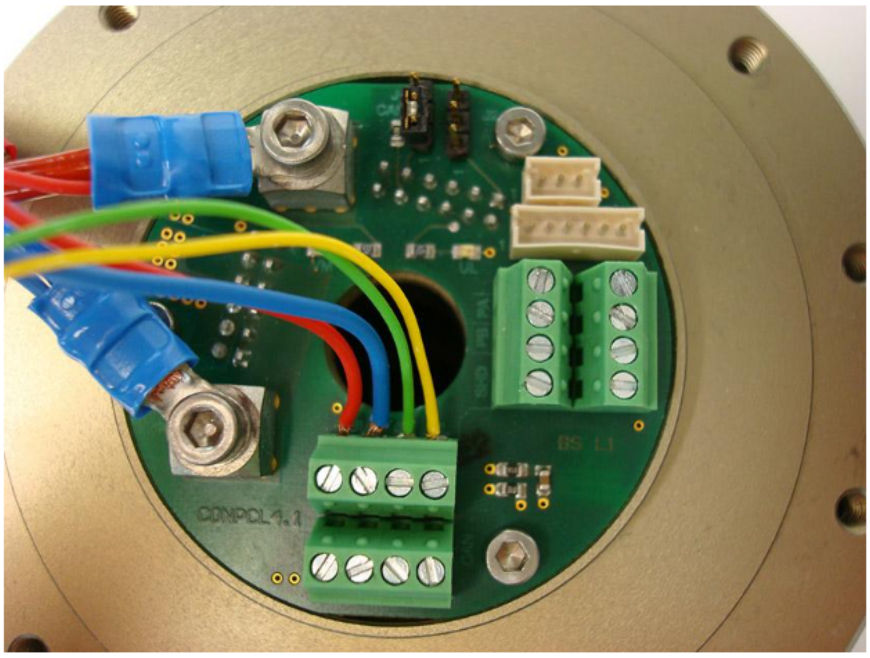


Figure 4.5: Assembling of the CAN bus (courtesy of Schunk)

4.3.1 PowerCube software

The original package of the Lightweight Arm 3 also included the PowerCube software that can be used for the control of the manipulator. A simple interface allows the user to choose which standard of communication is chosen by inserting the InitString that contains, besides the name of the standard, also the baud rate and the number of the port. By clicking the button ”Scan” in the window, the program scans the communication device specified with “InitString” and lists all detected modules. Double clicking a list box entry will open the module dialog box. Here it is possible to navigate between 6 pages: I/O Settings, Electrical Settings, Module Test, Identification, Drive/Controller Settings and finally General Settings. However, trying to control a robotic arm using this program is quite difficult. In addition to this, using the PowerCube software with the CAN bus is possible only whit one of the two interface boards supported by PowerCube: the ESD PCI331 (PCI board) or the ESD USB-Mini (USB interface). For this reason, the best way to go forward is to exploit the PowerCube communication protocol for the CAN network. The protocol used to exchange data between master and PowerCube is unified and independent from the bus interface used. The module receives serial data, interpretes it and acknowledges the command. The time necessary for this transfer depends on the bus interface and its parameters. A PowerCube command has this structure:

Identifier	CommandID	ParameterID or MotionID	Data 0 ... Data 7

Command ID, Parameter ID and following data bytes are identical for all communication interfaces (RS232, CAN, Profibus-DP). Only the Identifiers vary in dependency of the bus interface used. The PowerCube acknowledges all commands received (as far as not configured in another way) and the acknowledge data frame has the same structure as the command. All messages, either sent or received, are expressed using hexadecimal numbers. [16] A CAN-Bus-Identifier follows the CAN-Specification 2.0 Part A (11-bit-ID). There are three possible identifiers:

- **0x0A*M*** (cmd_ack) for the acknowledgement messages sent from module *M* to the master PC;
- **0x0C*M*** (cmd_get) for messages sent from the master to the module in order to read a parameter or a value;
- **0x0E*M*** (cmd_put) for messages sent from the master to the module to set a parameter or a value or to start a motion procedure.

In all cases *M* indicates the number of the module to which the message is referred ($1 \leq M \leq 7$). There are some commands that can be sent to all connected modules contemporarily, in this case the Identifiers listed above can't be used. They are replaced by the *Broadcast Identifier* **0x100**.

The Command ID is the first data transferred which states the command to be processed by the module. These commands are available:

- **Reset - 0x00** clears the error state and resets the encorders;

- **Home - 0x01** starts homing procedure;
- **Halt - 0x02** immediately stops one or all modules;
- **SetExtended - 0x08** set parameter (write target position or velocity, minimum or maximum position, velocity, acceleration, ...);
- **GetExtended - 0x0A** fetch parameter (read actual position, velocity, acceleration, current or minimum and maximum parameters);
- **SetMotion - 0x0B** set motion command (motion starts with target position or target velocity).

The first three Command IDs do not require any additional Parameter ID, Motion ID and obviously any data byte. On the contrary, for SetExtended and GetExtended it is necessary to specify the Parameter ID related to what needs to be read or written. The SetExtended also requires data bytes that must be sent to the LWA 3 to declare the parameter value. In a similar way, the SetMotion Command ID expects both the Motion ID and relative data bytes. The amount of data bytes sent with the command depends on the motion mode chosen. The data sent to the module will be immediately acknowledged (if not configured otherwise in the configuration word) and the data flow has the same exact structure of the message sent.

The third data transferred in the CAN message, after the Identifier and the Command ID, is the Parameter ID. Those available to the user are:

- **ActPos - 0x3C** returns the actual position in radiants (GetExtended command);
- **ActVel - 0x41** returns the actual velocity in rad/s (GetExtended command);
- **MinPos - 0x45** used to read or write the minimum position that a joint can reach, expressed in radiants (can be both a GetExtended or SetExtended command);
- **MaxPos - 0x46** used to read or write the maximum position that a joint can reach, expressed in radiants (can be both a GetExtended or SetExtended command);
- **MaxVel - 0x48** used to read or write the maximum velocity (in rad/s) that a joint can reach (can be both a GetExtended or SetExtended command);
- **MaxAcc - 0x4A** used to read or write the maximum acceleration (in rad/s^2) that a joint can reach;
- **TargetPos - 0x4E** set the target position that a joint must reach after it receives a following SetMotion command (SetExtended command);

- **TargetVel - 0x4F** set the target velocity that can be reached by a joint during motion (SetExtended command);
- **TargetAcc - 0x50** set the target velocity that can be reached by a joint during motion (SetExtended command).

For the commands that can be used to either fetch or set a parameter it is important to understand how to differentiate the message sent based on what the required action is. The solution is very simple: the robotic arm understand the request based on the Command ID used before the Parameter ID. Also, if a command is a SetExtended one, data bytes are required to complete the message. Finally, as an alternative to the ParameterID, for those Command IDs that require them, there are the Motion IDs. The two most important are:

- **FRAMP_MODE - 0x04** used to start a movement that stops at the position (in rad) entered via the data bytes (SetMotion command);
- **FVEL_MODE - 0x07** starts a motion with the chosen velocity (in rad/s), the movement stops when the maximum position is reached (SetMotion command).

It is worth noting that the PowerCube manual offers a wider range and choice of Command IDs, Parameter IDs and Motion IDs. However, for this thesis' work, it was chosen to include in the code of the control systems only the IDs reported above. In this way it has been possible to perform the required tasks while keeping the code as short and simple as possible.

Finally, it is important to pay attention to the format in which the data bytes must be specified. Even if PowerCube requires various types based on the necessity (char, unsigned char, short, unsigned short, long, unsigned long or float), the only one that has been used in this work has been the *float*. This means that the data at the end of the message must be expressed using floating point standard and ordered according to the Little Endian format. Once the data has been converted from decimal to floating point (IEEE-754) standard, the bytes must be separated and their order reversed. For example, the number 0.77 converted in float is 0x3F451EB8. Now, if the digits after 0x are divided in couples and these couples are reversed, the final result of the Little Endian ordering is: 0xB81E453F.

With all the tools described up until now it is possible to show what a message sent to the robotic arm would be like. If, for example, the velocity of module 2 has to be set to 0.77, the can message would be constructed as follows:

1. the Identifier is a *cmd_put* because data will be sent to the second joint of the arm, in this case 0x0E2;
2. Command ID is 0x08, SetExtended given that a parameter must be set;

3. SetExtended requires a Parameter ID, in this case TargetVel, 0x4F;
4. finally, the data bytes with reverse ordering expressing in floating point standard the chosen velocity for the module, in this case 0xb81e453f (as computed previously).

The message, separating the bytes is: 0xE2 0x08 0x4F 0xB8 0x1E 0x45 0x3F.

4.3.2 IXXAT CANAnalyser 3 Mini

As said in the previous paragraph, using the PowerCube software with the CAN bus is possible only whit one of the two interface boards supported by the program. For this reason, before starting the development of the control system, it has been necessary to test the CAN communication between the computer and the Lightweight Arm 3 using another interface. The one supplied at the RV&D facility is the USB-to-CAN Compact by Ixxat (Figure 4.6)

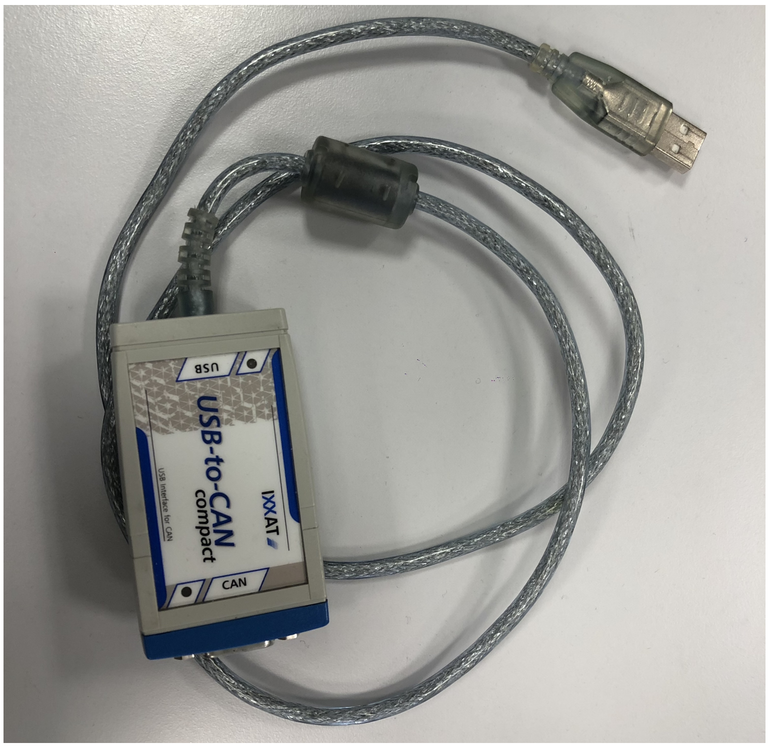
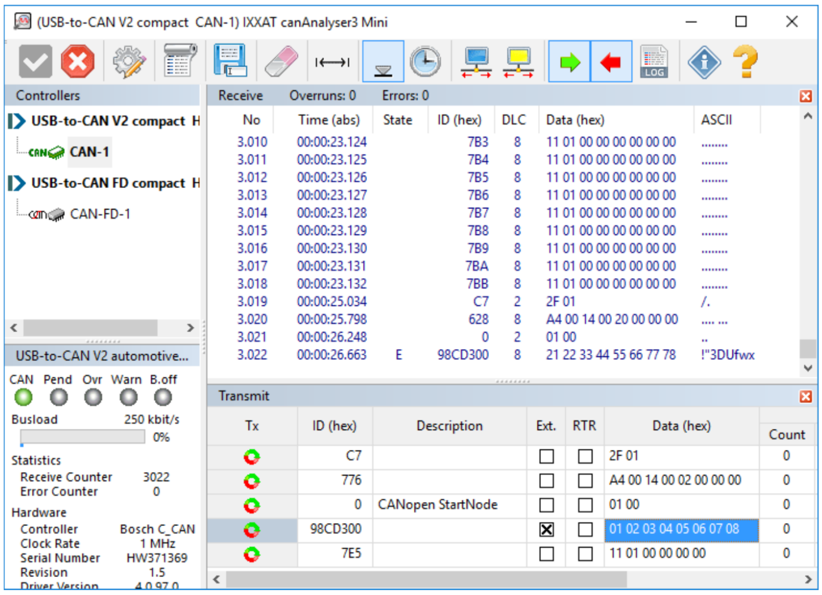


Figure 4.6: USB-to-CAN Compact Ixxat interface

This interface comes with a software that can be used to send the messages to the manipulator and see the acknoledgments that it returns. The USB-to-CAN Compact is easily plugged in and recognized by the program; communication is enabled by simply clicking on the green check in the top left corner. The transmission frame on the bottom of the screenshot in Figure 4.7 allows the user to send the CAN message providing the division of ID and data frame. In this case, the ID column refers only to the first byte of the CAN message described in the previous paragraph, the Identifier. The bytes relative to the Command ID, Parameter or Motion ID and the actual data bytes, must be entered in the column labeled "Data (hex)". For both the ID and the Data column, the "0x" before the byte can be omitted, because the program automatically recognize the number entered as hexadecimal. It is also possible to choose which format of CAN message is used, if the standard one or the extended one, by checking the box in the column "Ext." if necessary. The button on the far left in the Transmit window sends the message reported in the relative line when clicked. The receive windows shows the responses coming from the arm with the same structure used to send messages, and in addition to those, there are five more columns. The first two indicate the number and the time of each acknowledgement; the third indicates the state of the communication (for example errors); the fifth column "DLC" tells the Data Length, in other words how many data bytes are in the response; the seventh column reports any ascii conversion that can occur.


(USB-to-CAN V2 compact CAN-1) IXXAT canAnalyser3 Mini
LOG
Controllers
USB-to-CAN V2 compact H
CAN-1
USB-to-CAN FD compact H
CAN-FD-1
USB-to-CAN V2 automotive...
CAN Pend Ovr Warn B.off
Receive Overruns: 0 Errors: 0
Transmit

Busload 250 kbit/s 0%

Statistics
Receive Counter 3022
Error Counter 0

Hardware
Controller Bosch C_CAN
Clock Rate 1 MHz
Serial Number HW371369
Revision 1.5
Driver Version 4.0.97.0

No	Time (abs)	State	ID (hex)	DLC	Data (hex)	ASCII
3.010	00:00:23.124		7B3	8	11 01 00 00 00 00 00 00	....
3.011	00:00:23.125		7B4	8	11 01 00 00 00 00 00 00	....
3.012	00:00:23.126		7B5	8	11 01 00 00 00 00 00 00	....
3.013	00:00:23.127		7B6	8	11 01 00 00 00 00 00 00	....
3.014	00:00:23.128		7B7	8	11 01 00 00 00 00 00 00	....
3.015	00:00:23.129		7B8	8	11 01 00 00 00 00 00 00	....
3.016	00:00:23.130		7B9	8	11 01 00 00 00 00 00 00	....
3.017	00:00:23.131		7BA	8	11 01 00 00 00 00 00 00	....
3.018	00:00:23.132		7BB	8	11 01 00 00 00 00 00 00	....
3.019	00:00:25.034		C7	2	2F 01	/.
3.020	00:00:25.798		628	8	A4 00 14 00 20 00 00 00	....
3.021	00:00:26.248		0	2	01 00	..
3.022	00:00:26.663	E	98CD300	8	21 22 33 44 55 66 77 78	!"3DUfwx

Tx	ID (hex)	Description	Ext.	RTR	Data (hex)	Count
	C7		☐	☐	2F 01	0
	776		☐	☐	A4 00 14 00 02 00 00 00	0
	0	CANopen StartNode	☐	☐	01 00	0
	98CD300		☒	☐	01 02 03 04 05 06 07 08	0
	7E5		☐	☐	11 01 00 00 00 00	0

Figure 4.7: Screenshot from sotware Ixxat Cananalyser

If the message created in the previous paragraph has to be sent using The Ixxat Cananalyser software, the first column would contain the Identifier (E2), while all the other bytes (08 4F B8 1E 45 3F) will go in the Data column.

This software has been very useful for learning how to communicate with the LWA 3: how to order the message, divide it, send data and how to read the responses. The first steps of the control development have been made trying to send various messages to the arm and see if and how they would be received and their consequences, and made me understand which command the manipulator expects before others. The work carried out using Ixxat Cananalyser have been crucial for the design of the control system.

Chapter 5

Control system and Graphical User Interface

5.1 Tasks definition

The work done using Ixxat Cananalyser was very important, not only for learning how to construct and send a message to the LWA 3, but it also helped understanding what commands does the manipulator expects before other. First of all, after connecting the arm to the power supply, and powering on the latter, it is necessary to send the command *Reset* to clear any error state and reset the encoders. This has to be done for all the joints, so it is appropriate to use the *Broadcast* version of this command (with the Identifier 0x100) thus not having to repeat the action seven times. Now, it is possible to send any command for setting or reading parameters, but not to start motion. Before being able to do so, there is another command that must be sent to all joints, the command *Home*. This command starts the motors (that can be heard clicking) but does not change any of the joints' positions. The configuration in which the robotic arm is switched off is returned, indeed, as the new default home position. The command *Home* must be sent using the Broadcast Identifier, just like it is for *Reset*. After all these procedure are completed, the arm is ready to receive and perform any command, also the motion ones. As written in the previous chapter, there are two ways to start a joint movement: by specifying a target position (Motion ID 0x04) or a target velocity (Motion ID 0x07). Performing a movement in a safe way, however, requires the definition of a parameter through a SetExtended command. It is necessary to specify, with two consecutive messages, the target position and target velocity. One must be sent as a SetExtended command and the other as a SetMotion. There are two options of doing so:

- first set the velocity value using SetExtended Command ID (0x08) and TargetVel Parameter ID (0x4F), then start the joint movement specifying the position that has to be reached with the SetMotion Command ID (0x0B) and FRAMP_MODE (0x04);

- enter the target position with SetExtended Command ID (0x08) and TargetPos Parameter ID (0x4E), then start the movement specifying the joint velocity that has to be reached with the SetMotion Command ID (0x0B) and FVEL_MODE (0x07).

These two procedures may seem equivalent, however, after some tests performed with the Cananalyser software, it has been noted that actually they are not. In particular, the second strategy does not have the desired effect: after sending the SetMotion command, the motor does not stop its run when it reaches the target position but it keeps going until it reaches the end of stroke instead. This happens because the FVEL_MODE activates a movement based only on the velocity, ignoring the parameter of the requested position, thus stopping only when it arrives to the limit position. The PowerCube manual does not give a precise step by step procedure to control the arm so these tests where crucial to understand how to communicate with the manipulator to make it perform the required tasks in the correct way and order. After all these trials it has been possible to start the design of the control system to replace the not much user-friendly Cananalyser software. Now that the communication procedures are clear, it is possible to define which are the task that a reliable control system must perform:

- reset the encoders and clear the error state;
- change default home position and move the joints to the desired starting configuration;
- stop the motion immediately when a dangerous situation occurs;
- change parameters like maximum and minimum position to make sure that every motor can reach every position from 0° to 360° or from -180° to 180°, or at least cover as wider a range as possible, while avoiding any dangerous configuration;
- read position, velocity and other parameter requested by the user
- set target position, velocity and acceleration parameters
- start motion;
- perform different tasks at the same time (sending the same command to all joints or moving them all together);
- choose pre-defined position that might help the manipulator reach the target object more easily;

- compute the pose (position and orientation) of the end-effector through forward kinematics.

These tasks must be accomplished through a suitable user interface. During the course of this work, at first the control actions have been performed via a Command Line Interface, then after the completion of the design, the commands have been transferred on a more suitable and user-friendly Graphical User Interface, whose development will be the focus of paragraph 5.4.

5.2 Control structure

The user controls the arm through a user interface from the master (in this case a workstation or a PC) linked to the internal logic (the terminal board) via a communication interface (the CAN network). The terminal board is also connected to the power supply on one side and to the actuators (the joints of the manipulator) on the other. The modules' motors receive the commands via the CAN network and send back to the internal logic an acknowledgement after they perform the required tasks. The sensors also send messages to the master when the latter asks for data like actual position, velocity or acceleration. The control structure is schematized in Figure 5.1

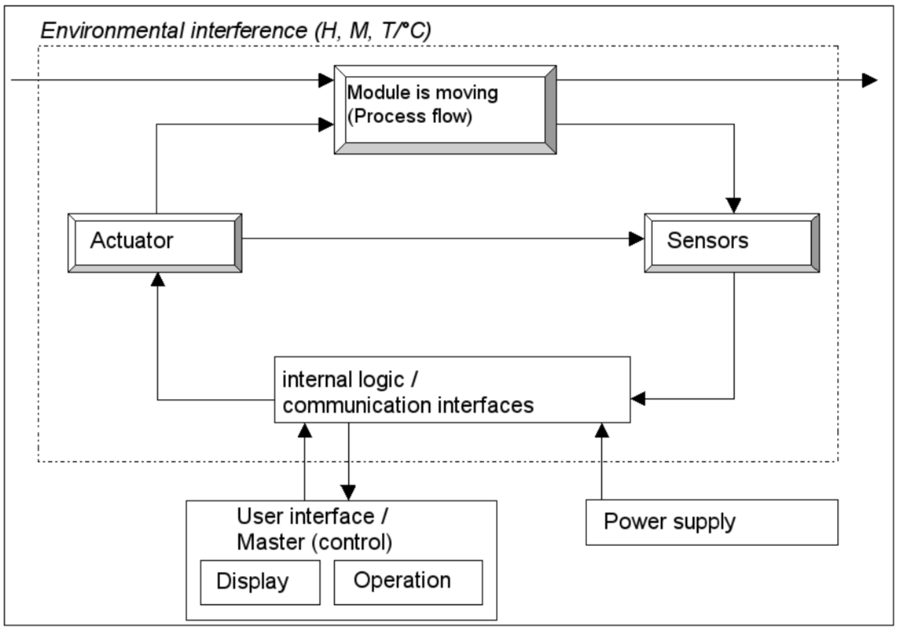

Environmental interference (H, M, T/°C)
Module is moving (Process flow)
Actuator
Sensors
internal logic / communication interfaces
User interface / Master (control)
Display
Operation
Power supply

Figure 5.1: Control Scheme of LWA 3

The first step is the creation of a short code that enables the communication via the CAN network. This code, called `init_can.sh`, must be launched from terminal after the power up of the computer and of the Lightweight Arm 3. Some Python scripts can be considered preparatory and complementary for the rest of the control system. As mentioned before, the Lightweight Arm 3 expects data in radiants and in floating point format with Little Endian byte ordering. For the user, providing data and values in such format would be very difficult and time consuming, for this reason three Python files have been created:

- `AngleConversion.py` that provides a function for converting from degrees to radiants and one that does the opposite; the second function comes in handy when, for example, the user sends commands to read the position or velocity of a joint and wants to see them in degrees rather than in radiants.
- `FloatingPoint.py` with a function that takes as input the data value in radiants and converts it from decimal to floating point according to the standard IEEE754; a second function does the opposite job taking as input the data provided by the arm as an output.
- `LittleEndian.py` where the two functions change the bytes ordering from Big Endian to Little Endian for input data and vice-versa for output ones.

In this way, it is possible to highlight two paths: one for encoding of data (crossed while feeding parameters to the arm) and the decoding one (undertaken when the output must be converted to a value of easy understanding for the user).

Encoding path:

Data input → AngleConversion.py (degrees to radiants) → FloatingPoint.py (decimal to IEEE754) → LittleEndian.py (reverse ordering) → LWA 3

Decoding path:

LWA 3 → LittleEndian.py (direct ordering) → FloatingPoint.py (IEEE754 to decimal) → AngleConversion.py (radiants to degrees) → Data input

The file that sends messages to the LWA 3 and receives and stores the acknowledgement with possible output data is the `launch.py`. This script has just one function, *launcher(CAN_msg)*, that takes as input the CAN message built in the other files and separates the Identifier from the rest. This last part, that came in the function as a string, is converted, element by element, from hexadecimal to decimal format. After defining the variables of the bus type (CAN) and of the channel (CAN port 1), the message is officially sent to the LWA 3 split in two parts: the *arbitration_id* equal to the Identifier, and the *data* equal to the array of the remaining data bytes expressed as decimal numbers.

After a pause of one second, the *launcher(CAN_msg)* function stores the message that is received on the bus defined before and prints it on the screen after converting it from Little Endian and floating point format to decimal value.

After these preliminary scripts the design of the control system continues with the creation of other Python files that can carry out the required tasks. Each task has been associated with a function and all the functions have been divided in some Python codes, based on the type of command or action performed. For example, tasks related to the reading and writing of parameters are incorporated in the same file; the same is valid for motion related commands or for functions that perform more actions at the same time.

This was decided in order to avoid the creation of many scripts, like what would have happened if each task had its own, or a very long one that contained every function.

At the end, the control structure consists of five Python files besides the preparatory ones listed earlier in the paragraph:

- `GeneralCommands.py` contains the commands that can be sent with the Broadcast Identifier (0x100), i.e. Reset, Home and Halt that respectively carry out the tasks of clearing the error state and resetting encoders, sending the manipulator back in its home configuration and stopping motion immediately. None of these commands require a Parameter ID or Motion ID after their Command ID.

- `ParametersRW.py` where there are the functions to read and write parameters like minimum and maximum position, maximum velocity, acceleration or current. These functions need the GetExtended Command ID when reading the data, and the SetExtended one during the configuration of a parameter.

- `Telemetry.py` is the Python file for the functions used to read the position, velocity or acceleration of each joint.

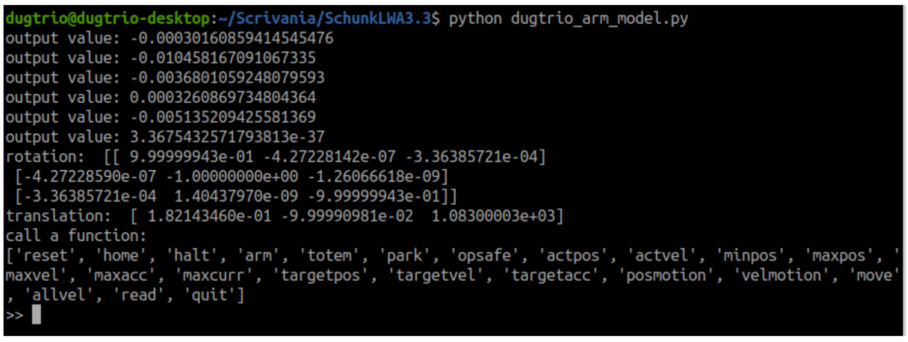
- `JointControl.py` stores all the functions related to motion, from the SetExtended commands to set the target position or velocity to the SetMotion ones.

- `Multiple.py` contains the commands aimed at all the joints contemporarily or those that forecast a sequence of functions being deployed consecutively: some examples of this category are the *Arm*, *Park*, *Totem* and *Op-safe* functions that will be explained in the next paragraph. This script also provides the functions to move all the joints at the same time with different rotations or to set their velocity with just one command.

Eventually, the last and most important file of the control system is the `dugtrio_arm_model.py`. This script is the core of the control system as it provides the kinematics parameters necessary to compute the required pose of the end-effector and also the call to all the functions defined previously: the first line of the codes, indeed, have the import statement for the other Python files created, whose function are necessary to control the arm. After these import statements there is the description of the arm as a class called **Dugtrio**: the first function, **__init__(self)** carries the geometric displacements between each joint of the arm in matricial form. It is worth noting that these values are the same as the Denavit-Hartenberg in Table 3.1. This function also calls the two successive ones that are necessary to compute the roto-translation matrix: the first is called **read(self)** and performs the task of reading the actual position of the joints and stores the results in a variable called *self.joints*. This variable is later used by the next function, **compute_rt(self)** that calculates the roto-translation from base to end-effector with the help of intermediate roto-translation matrices: some represent the movement from one fixed joint to the mobile configuration of the next one and others are relative to the rotation from the mobile reference frame to the fixed one of the same joint. The iteration of matrices multiplication for all joints couples gives the final result of the roto-translation matrix. The first three elements of the last column of this matrix are the x, y and z coordinates of the end-effector in the moment when the computations are performed. The roll, pitch and yaw are calculated with the help of an appropriate Python file, `RPYconversion.py`. This file takes the roto-translation matrix as an input and, using the relative formulas, gives as output the values that make up the orientation of the end-effector.

The functions that complete the Dugtrio class are simply the call to their corresponding ones from the imported files in order to compute the required task when they are launched from the Command Line Interface.

The final lines of this code are the `if __name__ == '__main__':.py` loop: when the file is launched from the terminal, this part of the script is executed, printing out on the screen the Command Line Interface, visible in Figure 5.2.



```
dugtrio@dugtrio-desktop:~/Scrivania/SchunkLWA3.3$ python dugtrio_arm_model.py
output value: -0.00030160859414545476
output value: -0.010458167091067335
output value: -0.0036801059248079593
output value: 0.0003260869734804364
output value: -0.005135209425581369
output value: 3.3675432571793813e-37
rotation:  [[ 9.99999943e-01 -4.27228142e-07 -3.36385721e-04]
 [-4.27228590e-07 -1.00000000e+00 -1.26066618e-09]
 [-3.36385721e-04  1.40437970e-09 -9.99999943e-01]]
translation:  [ 1.82143460e-01 -9.99990981e-02  1.08300003e+03]
call a function:
['reset', 'home', 'halt', 'arm', 'totem', 'park', 'opsafe', 'actpos', 'actvel', 'minpos', 'maxpos', '
maxvel', 'maxacc', 'maxcurr', 'targetpos', 'targetvel', 'targetacc', 'posmotion', 'velmotion', 'move'
, 'allvel', 'read', 'quit']
>> 
```

Figure 5.2: Command line control window

At the prompt, the user can type in any of the commands available among those that appear. Based on the user choice, the relative function is called and the required task executed. The script is structured in such a way that, after any movement of any joint, the roto-translation matrix and relative pose of the end-effector are computed and printed out on screen.

5.3 Experimental tests

The first task to carry out is the choice of a suitable home configuration for the Schunk LWA 3.

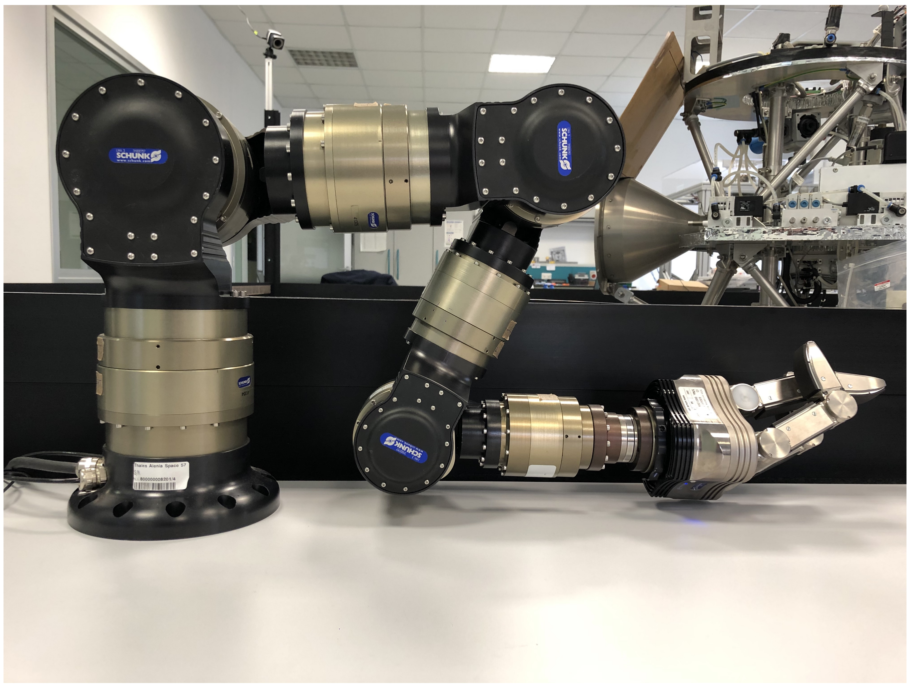


Figure 5.3: LWA 3 in Park position, its original home configuration

Figure 5.2 above shows the position in which the Schunk Lightweight Arm was locked at the beginning of my thesis. This position, that later will be defined as *Park*, is optimal for when the manipulator has to be transported from one location to another because it is the configuration that takes up the least space possible.

It is also important to notice that in the image above, all the joints are in their natural zero position. This means that, when the Schunk arm is asked for the position of each joints, the output data is equal to 0. However this configuration is very inconvenient as default home configuration, so it is crucial to find a better one.

None of the commands listed on the PowerCube manual allows to change the default home configuration, but it is reported that there is a way of doing so in an artificial manner: the Schunk LWA 3 always remember as its home position, the one in which it has been turned off and as a consequence, the one it has at power on. This allows the user to arbitrarily choose the position that he considers the most convenient. In this case the position chosen is the one called *Totem*, visible in Figure 5.3

The advantages of the *Totem* configuration is that it is more user friendly than the *Park* one, as the user can easily assess the position of each joint with respect to its maximum and minimum end of stroke. Also, any position is easier to reach from *Totem* than it is from *Park*. Lastly, the configuration was chosen because it was more compliant to safety requirements: if, for example, from the *Park* position joint 4 moves

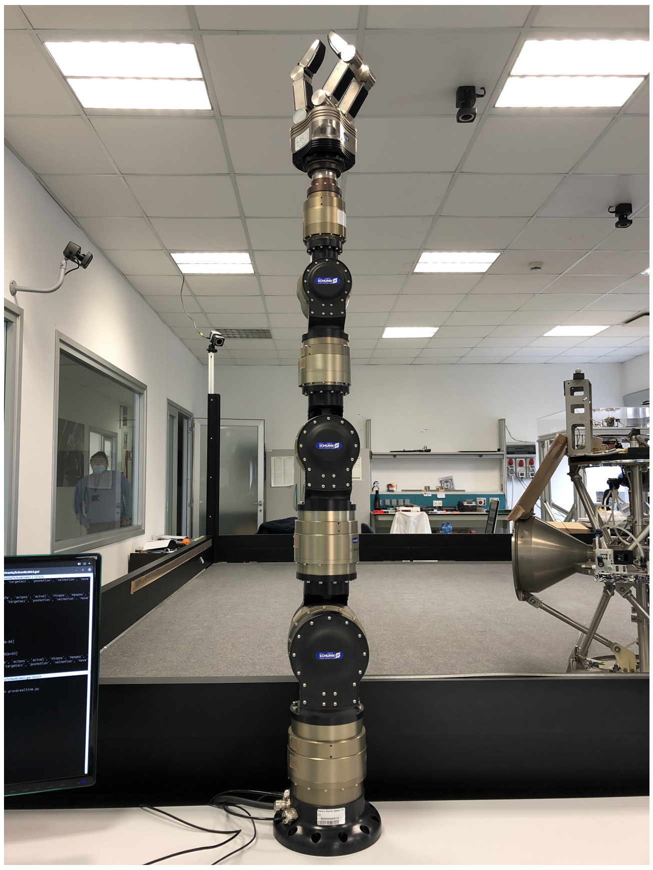


Figure 5.4: Totem and home position

in anti-clockwise direction, joint 6 would crash on the desk surface, to which it is already too close.

It is clear that the only advantage of *Park* is the fact that it occupies the least space possible and so it is a configuration useful only for situations like the transport described earlier in this paragraph.

After computing the displacement that would offset the home position, these values have been used as parameter for a FRAMP_POS motion command to move joints 2, 4 and 6. Of course these value have been obtained, checked and confirmed with a trial and error procedure, until the new setting would be more than acceptable. Joints 1, 3, 5 and 7 are already in their optimal setup and do not need any offset. The offset values for the new home position are reported in Table 5.1. Now that the joints are organized in a different way it is necessary to set the new home position as the user zero. Keeping the zeros related to *Park* would make very difficult for the user to understand where the joint is in its range of possible positions. The offsets in Table 5.1 are the entry values of an array used to change the position parameters, when required, to make them fall into line with the positions read by the arm. This simple line of code makes the user job a lot easier. For example, if the person that is controlling the arm wants to move joint 2 10° in the positive direction (anti-clockwise) from its *Totem*, he would have to enter 100°. This, however, would be a very counterintuitive computation that can be avoided

Joint	displacement [deg]
Joint 1	0
Joint 2	90
Joint 3	0
Joint 4	120.32
Joint 5	0
Joint 6	-115.74
Joint 7	0

Table 5.1: *Totem* offset displacements with respect to *Park* configuration

by letting the code automatically add 90° (the offset of joint 2) at the value entered by the user (in this case 10°). The *Totem* command, whose corresponding function is stated in the `Multiple.py` script, is crucial for recovering the home position when needed. However rare, it might happen that the power goes off and that the LWA 3 suddenly shuts down. If this situation occurs, when it is restarted, the robotic arm will identify as its home position the one in which it has turned off. In this case, having the *Totem* command helps recover the desired home position. The user can, indeed, launch the command right before the arm is manually turned off, thus restoring the best home configuration.

Another command stated in the `Multiple.py` Python file is *Arm*. The tasks performed are: sending the *Reset* command to all joints and the setting of maximum and minimum position (*MaxPos* and *MinPos*) that each module can reach during its movement. The need for such a command has become evident when it was noticed that some joints did not reach the target position set sometimes. For example, when joint 1 received a SetMotion task to reach the position of -180°, it did not carry out its task, stopping at -120° circa. It is clear that the problem to solve was the configuration of the end of strokes values in the LWA 3 memory.

After studying the structure of the manipulator in all its joints, it has been decided that the joints 1, 3, 5 and 7 can fulfill a complete round without causing or risking any structural danger. Their stroke can go from the -180° to the 180° position. The same cannot be said for joints 2, 4 and 6 whose maximum and minimum positions have been imposed at 120° and -120° respectively. Pushing the limit further was in fact a huge risk of collision between the joints in some configurations, but the values chosen guarantee safety in any situation.

The most important arm configuration that has been defined is the *Op-safe* one. The manipulator reaches this configuration when the homonym command is launched through the `Multiple.py` Python file. This position, seen in Figure 5.5, is a safe configuration from which the arm can easily move when it has to pick up an object. It can be considered a pre-operational position. The values at which the joints are moved, reported in table 5.2, have been decided with a series of tests to assess a

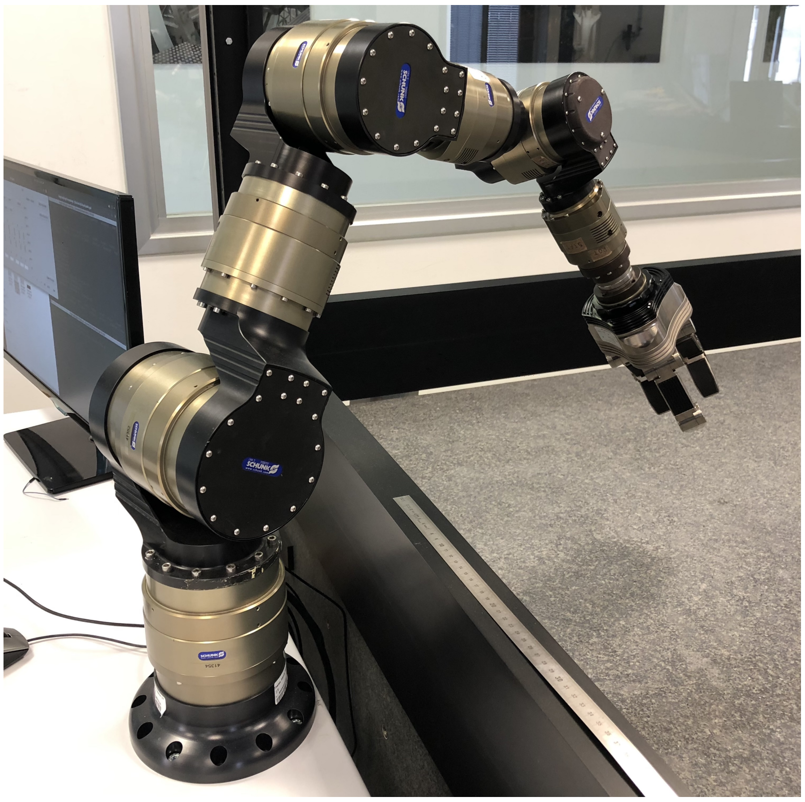


Figure 5.5: Op-safe position

suitable distance of the end-effector from the floor. Also important, the shift from *Park* or *Totem* to *Op-safe* must happen without any collisions or close calls with the surrounding environment in the RV&D facility.

Joint	displacement [deg]
Joint 1	70
Joint 2	30
Joint 3	0
Joint 4	70
Joint 5	20
Joint 6	50
Joint 7	0

Table 5.2: *Op-safe* displacements values with respect to *Totem* configuration

5.4 Graphical User Interface

All the tests described above were performed using the Command Line Interface shown in Figure 5.2. After the completion of the control system, all the tools and function have been transferred to a Graphical User Interface (GUI). A GUI is a more intuitive and easy to use window that replaces the CLI while it still carries out the same tasks. The GUI has been developed using the **tkinter** package ("Tk interface"), the standard

Python interface to the Tk GUI toolkit. With few lines of code, tkinter allows the creation of windows, frames, labels, buttons and more tools that have been used to create the Schunk LWA 3 GUI. A screenshot is visible in Figure 5.6.

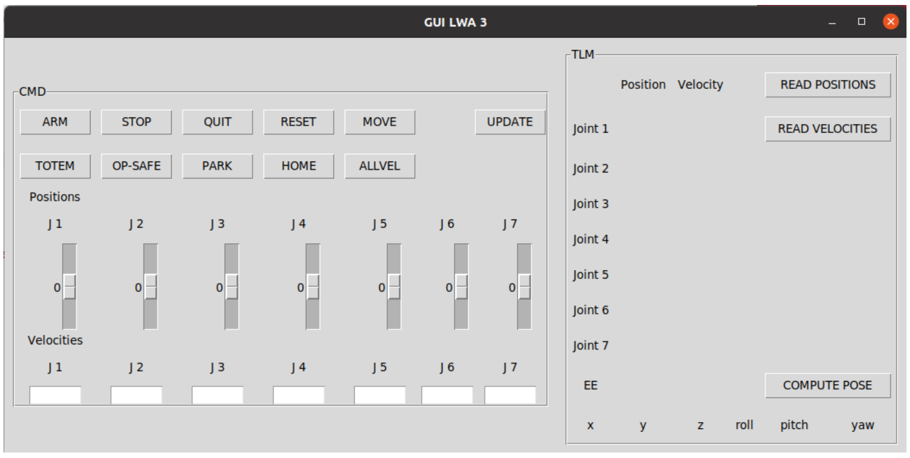
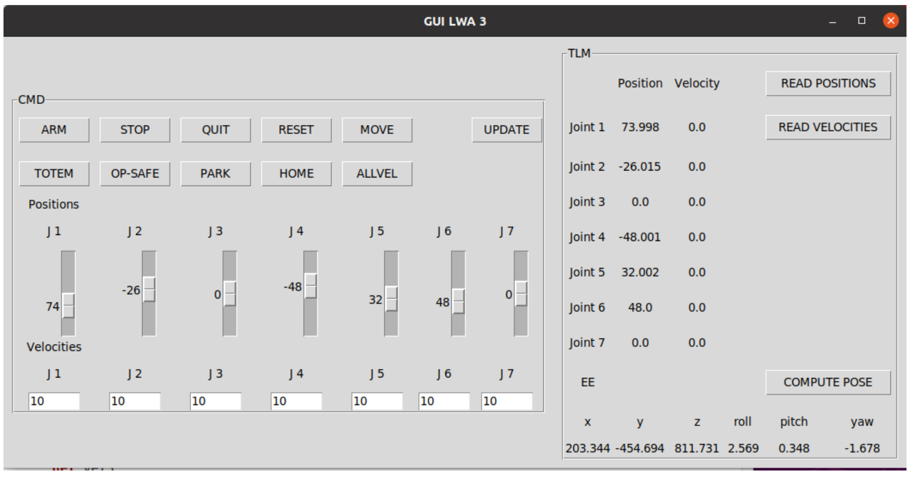


Figure 5.6: GUI window at power on

Here, it is possible to see that the main window is divided in two frames: the **CMD** on the left for sending commands to the arm, and the **TLM** on the right to analyse the telemetry. The first two rows of the CMD frame include buttons that launch the homonym commands. Two of those commands are linked to the third and fourth lines of the frame. The sliders in the third row are used to decide the movement of each joint. Their values are the input for the task performed by "MOVE", that indeed, is to move every module by the angle set on the respective slider. A similar task is carried out by the "ALLVEL" button, that takes the numbers entered in the boxes on the fourth row to set the velocity of the corresponding joint. These entry boxes, the buttons and the sliders are created using the homonym tkinter widgets.

The sliders have the same maximum and minimum limits as the positions that each joint can reach.



Positions

J 1	J 2	J 3	J 4	J 5	J 6	J 7
74	-26	0	-48	32	48	0

Velocities

J 1	J 2	J 3	J 4	J 5	J 6	J 7
10	10	10	10	10	10	10

	Position	Velocity
Joint 1	73.998	0.0
Joint 2	-26.015	0.0
Joint 3	0.0	0.0
Joint 4	-48.001	0.0
Joint 5	32.002	0.0
Joint 6	48.0	0.0
Joint 7	0.0	0.0

EE

x	y	z	roll	pitch	yaw
203.344	-454.694	811.731	2.569	0.348	-1.678

Figure 5.7: GUI window working

Figure 5.7 shows a screenshot of the GUI window where the sliders are moved from the zero position and there are values entered in the velocity boxes. The resulting configuration, obtained after clicking the button MOVE, is shown in figure 5.8.

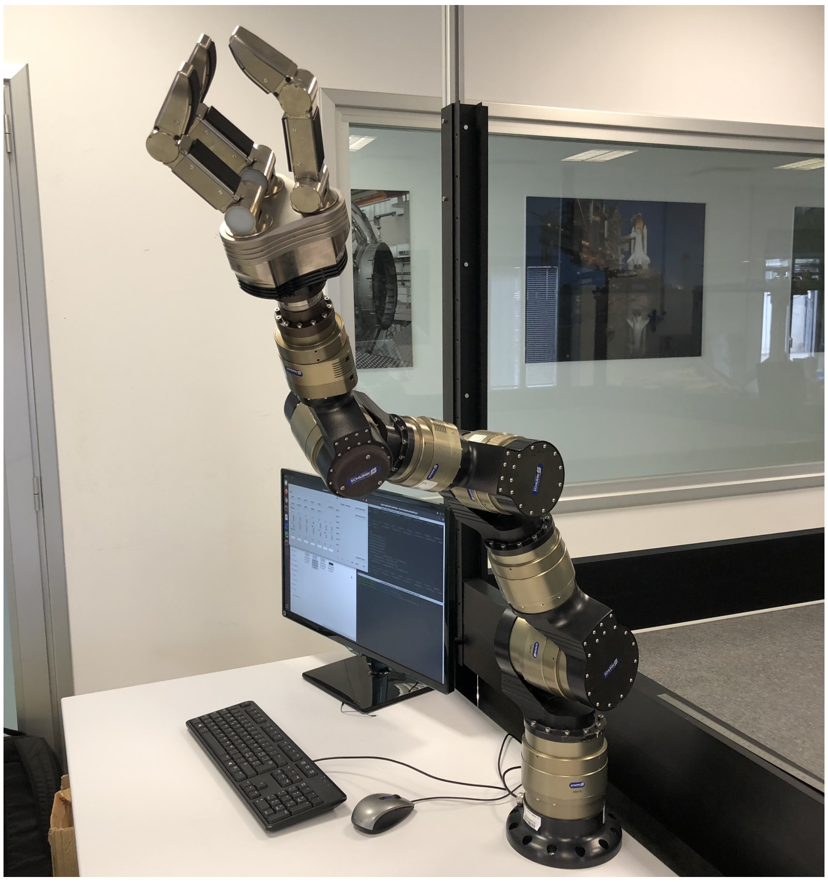


Figure 5.8: Position set through GUI

The screenshot in Figure 5.7 also shows the right frame running. In the TLM frame the joints' positions and velocities are labels, constantly updated in real time after the buttons "READ POSITIONS" and "READ VELOCITIES" are clicked once. On the bottom right, the "COMPUTE POSE" calculates position and orientation of the end-effector when clicked.

Chapter 6

Conclusions

The target of this thesis project was to design a suitable control system for a robotic manipulator that can be used during on-orbit satellite servicing missions. The first step was to study the state of the art of the present servicing technologies and on-ground simulation facilities, in order to create a solid knowledge about it. A deep study of the manipulator modelling and kinematics was conducted with a particular focus on the Denavit-Hartenberg representation and the coordinate frame assignment. The equations needed to compute the roto-translation matrix are presented as an essential part of the study of the forward kinematics of a 7-DoF robotic arm. Then, all the informations about the Schunk LWA 3 are collected, from the mechanical structure to the electrical components and connections to the power supply. The CAN bus protocol, used to communicate with the arm, is presented starting from the ISO 11898 and its layers. The standard data frame and the main features of the protocol are deeply illustrated to introduce and better understand how it was implemented for the Schunk LWA 3. Also important is the description of the structure of the CAN message and the work done with software that enables a simple communication through a USB-to-CAN interface. A description of the design and development of control system follows, with focus on the structure, the functions created and the tasks performed by the LWA 3. The tests of the manipulator's functionalities are reported, highlighting what obstacles had to be overcome, the solutions adopted and the different configuration of the arm that can be used in various situations. A brief description of the GUI developed for controlling the arm closes the chapter regarding the control system. After these conclution there will be two appendices: one for the LWA 3 user manual and the other for the code written during this thesis work.

6.1 Future developments

It is worth highlighting the future developments that can follow this project. As a matter of fact, there are some areas that the control system does not cover at the moment but could be added in the future. First of all, two types of control could be implemented to support the forward kinematics: the inverse kinematics (cartesian control) and the impedance control. The first would allow the user to specify a certain position in the base reference frame for the end-effector to reach and the control system would autonomously compute the movement for each joint in order to achieve such task. This would save a lot of time and computation on the user side. The impedance control, instead, would allow the user to manually move the arm applying a small force on an end-effector equipped with a force sensor. Also in this case, the computer would compute the movements for each joint to follow the directions imposed by the user. Finally, another area of improvement could be the extension of the control to the Schunk Dexterous Hand (SDH), the end-effector tool supplied at the Thales Alenia Space RV&D facility. The SDH, like the LWA 3, uses the CAN protocol for communication and could be included in the framework of the arm with the addition of suitable commands and function for its control.

Appendix A

User manual

This section contains the step-by-step procedure to use the Schunk LWA 3, from the power supply setup to the launch of the control interfaces. First of all, while the power supply is not connected to the terminal block, set the voltage at 24 VDC and the current at 2 A maximum. Then, link the cable from the terminal block to the power supply: two wires inside this cable must be connected to the 24 and 0 VDC pin of the terminal block on one side and to the + and - of the power supply on the other. In a similar way, the CAN cable inserted in the CAN port of the computer, must be connected, on the other end, to the CAN pin of the terminal block.

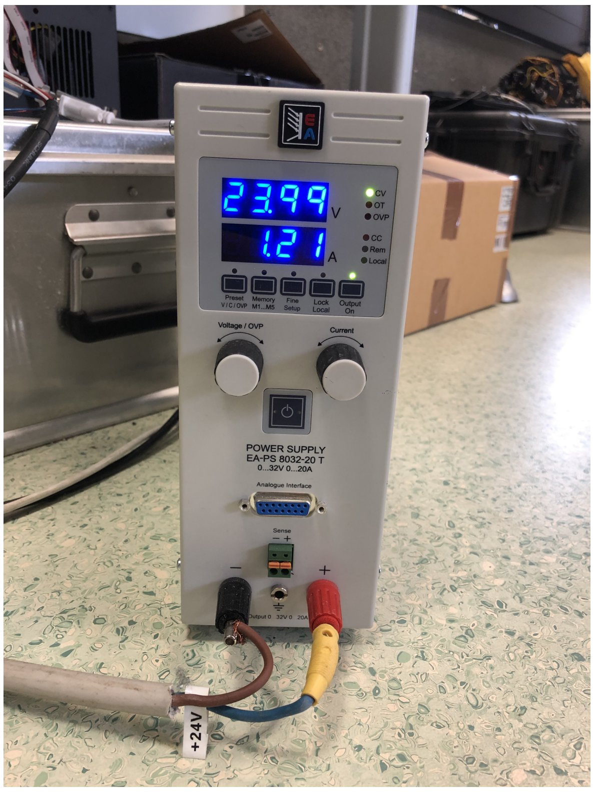


Figure A.1: Power supply

Before switching on the power supply and the computer, it is mandatory to connect the red push button to the emergency circuit on the terminal block.

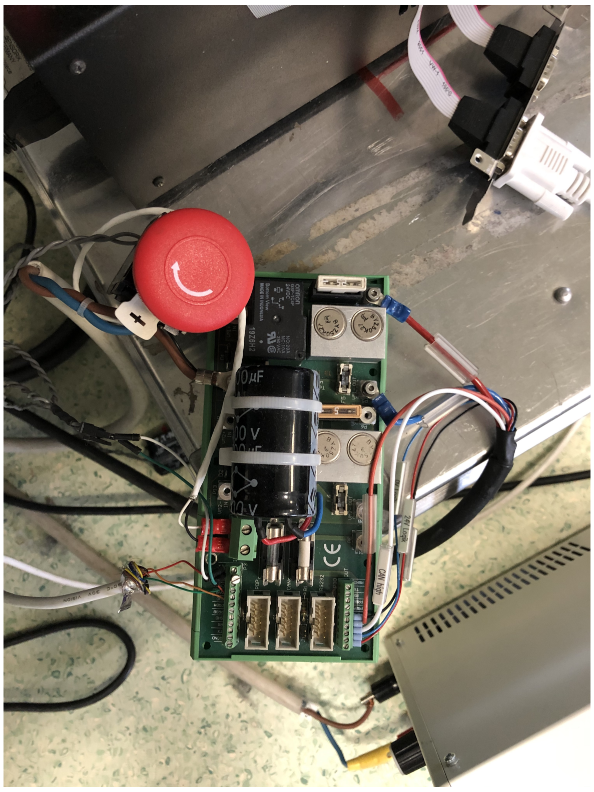


Figure A.2: Emergency push button and terminal block

Once all the cables are connected and the terminal block is linked via CAN to a port of the computer it is possible to power up both the power supply and the computer. If the SDH or any other end-effector are not connected, it is possible to measure the electric potential different at the pins visible on top of joint 7. Make sure that it is equal to 24 VDC before going on with the next steps. Then, right click on the folder "SchunkLWA3" and open it in the terminal. At the prompt, write `./init_can.sh` and press Enter to initialize the communication on the CAN port 1. Type the password and then write the name of the file that contains the command interface:

- `python dugtrio_arm_model.py` if the arm is going to be controlled through the command line;
- `python guirealtime.py` if the choice is to use the GUI.

No matter what file has been chosen, after it has loaded, the first command that must be launched is *Arm*: type the name "arm" at the prompt in the CLI or click the button in the GUI and wait for the LWA 3 to clear the error states and to set the maximum and minimum position for each joint. Once the Arm procedure is over, send the command *Home*, otherwise no motion will be allowed. When the motors' clicking has stopped, the manipulator is ready to receive any command. Now it is possible to start any procedure and send the commands described in Chapter 5 to make the LWA

3 carry out the desired tasks. If the arm is about to collide with any surface, object or instrumentation of the facility, press the red emergency push button or click the button "STOP" in the GUI (or type the command if using the CLI) to immediately stop the motion and avoid any dangerous event. After the robotic arm has been stopped, it is necessary to send the *Reset* command to clear the error state before resuming any procedure. If the push button was pressed, make sure to unlock it by turning it in the direction of the arrow. In this case it is not necessary to send the arm in the *Home* position, although very recommendable. If any sudden shutdown occurs, reset the home configuration by turning on the power supply, move the manipulator to the *Totem* position and turn it off. However, to avoid the loss of the chosen home, make sure to always type or click "QUIT" before turning on the power supply: this will send the *Totem* command to the arm before closing the GUI or terminal window.

Appendix B

Code

In this appendix is listed the code relative to the control of the Schunk LWA 3. This first part is the `dugtrio_arm_model.py`, that is the main file of the control system: it contains the initialization of the arm, its kinematic description through the Denavit-Hartenberg parameters, the computation of the roto-translation matrix and the call to the functions created to communicate and send tasks to the LWA 3.

```python
from numpy import *
import numpy as np
import GeneralCommands
import JointControl
import launch
import ParametersRW
import Telemetry
import Multiple
# import time


class Dugtrio:

    # Initialisation
    def __init__(self):
        self.joints = []
        self.__links = np.array([[0,        0,          133.1],
                                 [0,        -13.1,      -166.9],
                                 [0,        182.6,      13.1],
                                 [0,        -13.6,      -145.4],
                                 [0,        159.1,      13.6],
                                 [0.565,    -10.785,    -117.4],
                                 [-0.571,   141.6,      10.885],
                                 [0,        0,          -36.9]], dtype=float)
        self.read()
        self.rt = self.compute_rt()
        self.eps = 0.1

    def read(self):
        CAN_msg = [['0x0C' + str(i), '0x0A', '0x3C'] for i in range(1, 8) if i != 3]
        position = []
        for msg in CAN_msg:
            position.append(launch.launcher(msg))
        position.insert(2, 0)
        self.joints = position
        self.rt = self.compute_rt()
        print('rotation: ', self.rt[0:3, 0:3])
        print('translation: ', self.rt[0:3, 3])

    # Compute the roto-translation from base to end-effector
    def compute_rt(self):

        # From the inertial frame to the first fixed-joint
        t_01f = [[1, 0, 0, self.__links[0, 0]],
                 [0, -1, 0, self.__links[0, 1]],
                 [0, 0, -1, self.__links[0, 2]],
                 [0, 0, 0, 1]]

        # From the first fixed-joint to the first mobile-joint
        t_1f1 = [[cos(self.joints[0]), -sin(self.joints[0]), 0, 0],
                 [sin(self.joints[0]), cos(self.joints[0]), 0, 0],
                 [0, 0, 1, 0],
                 [0, 0, 0, 1]]
        # Iterate the matrix multiplication from joints couples (1,2) to (6, 7)
        from_v = True
        t_1to7 = [[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]]

        for link in range(0, len(self.joints) - 1):
            # From fixed to mobile
            t_ftm = [[cos(self.joints[link + 1]), -sin(self.joints[link + 1]), 0, 0],
                     [sin(self.joints[link + 1]), cos(self.joints[link + 1]), 0, 0],
                     [0, 0, 1, 0],
                     [0, 0, 0, 1]]

            # If we start from a vertical joint we adopt t_vh, otherwise t_hv
            if from_v:
                t_mtf = [[1, 0, 0, self.__links[link + 1, 0]],
                         [0, 0, 1, self.__links[link + 1, 1]],
                         [0, -1, 0, self.__links[link + 1, 2]],
                         [0, 0, 0, 1]]
            else:
                t_mtf = [[1, 0, 0, self.__links[link + 1, 0]],
                         [0, 0, -1, self.__links[link + 1, 1]],
                         [0, 1, 0, self.__links[link + 1, 2]],
                         [0, 0, 0, 1]]

            # Update the transformation chain
            t_mm = matmul(t_mtf, t_ftm)
            t_1to7 = matmul(t_1to7, t_mm)

            # Change the flag
            # print(from_v)
            from_v = not from_v

        # Define the last transformation from joint-7 to end-effector
        t_7e = [[1, 0, 0, self.__links[7, 0]],
                [0, 1, 0, self.__links[7, 1]],
                [0, 0, 1, self.__links[7, 2]],
                [0, 0, 0, 1]]

        # Compute the global roto-translation
        mtx = matmul(matmul(matmul(t_01f, t_1f1), t_1to7), t_7e)
        # print(type(mtx))
        return mtx

    # Command on joints
    def move(self):

        joint_space = input('insert joint angular displacements: ').split(' ')
        joint_space = [float(joint_space[i]) for i in range(len(joint_space))]

        # Bounds for variables
        down = [-180 for i in range(len(joint_space))]
        up = [180 for i in range(len(joint_space))]

        if not len(joint_space) != 7:

            if down < joint_space < up:
                joint_space[2] = 0
                # sending position to arm
                count = 0
                for joint in joint_space:
                  if count != 2:
                    CAN_msg = JointControl.PosMove(str(count + 1), float(joint))
                    launch.launcher(CAN_msg)
                  count += 1
                self.read()

            else:
                print("Wrong input: enter joints variables in [-180, 180].")
        else:
            print("Wrong input: joint space must have shape 7.")

    # Reset the encoders
    def reset(self):
        GeneralCommands.Reset()
        return True

    # Send one or more joints to their home positions
    def home(self):
        GeneralCommands.Home()
        while [0 - self.eps] * 7 < self.joints < [0 + self.eps] * 7:
            self.read()
            break
        return True

    # Immediately stops the movement of one or more joints
    def halt(self):
        GeneralCommands.Halt()
        return True

    # Resets the encoder and set the minimum an maximum position for each joint
    def arm(self):
        Multiple.Arm()

    # Bring the arm in vertical position
    def totem(self):
        Multiple.Totem()
        self.read()
        return True

    # Bring each joint to its encoder's zero position
    def park(self):
        Multiple.Park()
        self.read()
        return True

    # Pre-operation position
    def opsafe(self):
        Multiple.Opsafe()
        self.read()
        return True

    # Read actual position
    def actpos(self):
        CAN_msg = Telemetry.ActPos()
        response = launch.launcher(CAN_msg)
        return response

    # Read actual velocity
    def actvel(self):
        CAN_msg = Telemetry.ActVel()
        response = launch.launcher(CAN_msg)
        return response

    # Read or write the minimum position
    def minpos(self):
        CAN_msg = ParametersRW.MinPos()
        response = launch.launcher(CAN_msg)
        return response

    # Read or write the maximum position
    def maxpos(self):
        CAN_msg = ParametersRW.MaxPos()
        response = launch.launcher(CAN_msg)
        return response

    # Read or write the maximum velocity
    def maxvel(self):
        CAN_msg = ParametersRW.MaxVel()
        response = launch.launcher(CAN_msg)
        return response

    # Read or write the maximum acceleration
    def maxacc(self):
        CAN_msg = ParametersRW.MaxAcc()
        response = launch.launcher(CAN_msg)
        return response

    # Read or write the maximum current
    def maxcurr(self):
        CAN_msg = ParametersRW.MaxCurr()
        response = launch.launcher(CAN_msg)
        return response

    # Set the target position before sending a motion command
    def targetpos(self):
        CAN_msg = JointControl.TargetPos()
        response = launch.launcher(CAN_msg)
        return response

    # Set the target velocity before sending a motion command
    def targetvel(self):
        CAN_msg = JointControl.TargetVel()
        response = launch.launcher(CAN_msg)
        return response

    # Set the target acceleration before sending a motion command
    def targetacc(self):
        CAN_msg = JointControl.TargetAcc()
        response = launch.launcher(CAN_msg)
        return response

    # Send motion command to reach the set position
    def posmotion(self):
        CAN_msg = JointControl.PosMotion()
        response = launch.launcher(CAN_msg)
        self.read()
        return response

    # Send motion command to move with the set speed
    def velmotion(self):
        CAN_msg = JointControl.VelMotion()
        response = launch.launcher(CAN_msg)
        return response

    # Send position motion command to each joint (with a different
   angle for each one)
    ''' def allpos(self):
        CAN_msg = Multiple.AllPos()
        response = launch.launcher(CAN_msg)
        return response '''

    # Set the same velocity for all joints
    def allvel(self):
        Multiple.AllVel()


```

```

if __name__ == '__main__':
    arm = Dugtrio()
    arm.compute_rt()
    cmd_name = ['reset', 'home', 'halt', 'arm', 'totem', 'park', '
opsafe', 'actpos', 'actvel', 'minpos', 'maxpos',
                'maxvel', 'maxacc', 'maxcurr', 'targetpos', 'targetvel
', 'targetacc', 'posmotion', 'velmotion',
                'move', 'allvel', 'read', 'quit']
    cmd_func = [arm.reset, arm.home, arm.halt, arm.arm, arm.totem, arm
.park, arm.opsafe, arm.actpos, arm.actvel, arm.minpos,
                arm.maxpos, arm.maxvel, arm.maxacc, arm.maxcurr, arm.
targetpos, arm.targetvel, arm.targetacc,
                arm.posmotion, arm.velmotion, arm.move, arm.allvel,
arm.read]
    while True:
        cmd = input(f'call a function: \n{cmd_name} \n>> ')
        if cmd in cmd_name and cmd != 'quit':
            i = cmd_name.index(cmd)
            cmd_func[i]()

        elif cmd == 'quit':
            break

        else:
            print('command not found')
```

This second part of the code is the `guirealtime.py` file. Most of these lines are aimed at the creation of the elements that form the main window using the widget provided by tkinter (labels, entries, buttons and sliders). Some lines are the definition of the commands performed by the buttons when these are clicked. This file imports the previous one and makes use of some of its functions such as *read(self)* and *compute_rt(self)* for the computation of the end-effector's pose.

```
from tkinter import *
import Multiple
import GeneralCommands
import Telemetry
import launch
import AngleConversion
import math
from dugtrio_arm_model import Dugtrio


root = Tk()
root.title('GUI LWA 3')
dg = Dugtrio()
```

```python

cmd_frame = LabelFrame(root, text="CMD")
tlm_frame = LabelFrame(root, text="TLM")
cmd_frame.grid(row=0, column=0, padx=10, pady=10)
tlm_frame.grid(row=0, column=1, padx=10, pady=10)


def update():
    global entry1
    global vel1
    message = Telemetry.ActPos('1')
    if entry1 is not None:
        entry1.grid_forget()
        del entry1
    pos1 = launch.launcher(message)
    pos1deg = AngleConversion.RadToDeg(pos1)
    entry1 = Label(tlm_frame, text=str(round(pos1deg, 3)))
    entry1.grid(row=1, column=1, padx=5, pady=10)
    message = Telemetry.ActVel('1')
    if vel1 is not None:
        vel1.grid_forget()
        del vel1
    jv1 = launch.launcher(message)
    jv1deg = AngleConversion.RadToDeg(jv1)
    vel1 = Label(tlm_frame, text=str(round(jv1deg, 3)))
    vel1.grid(row=1, column=2, padx=5, pady=10)
    tlm_frame.after(237, update2)


def update2():
    global entry2
    global vel2
    message = Telemetry.ActPos('2')
    if entry2 is not None:
        entry2.grid_forget()
        del entry2
    pos2 = launch.launcher(message)
    pos2deg = AngleConversion.RadToDeg(pos2)
    entry2 = Label(tlm_frame, text=str(round(pos2deg, 3)))
    entry2.grid(row=2, column=1, padx=5, pady=10)
    entry3 = Label(tlm_frame, text="0.0")
    entry3.grid(row=3, column=1, padx=5, pady=10)
    message = Telemetry.ActVel('1')
    if vel2 is not None:
        vel2.grid_forget()
        del vel2
    jv2 = launch.launcher(message)
    jv2deg = AngleConversion.RadToDeg(jv2)
    vel2 = Label(tlm_frame, text=str(round(jv2deg, 3)))
    vel2.grid(row=2, column=2, padx=5, pady=10)
    vel3 = Label(tlm_frame, text="0.0")
    vel3.grid(row=3, column=2, padx=5, pady=10)
    tlm_frame.after(237, update4)


def update4():
    global entry4
    global vel4
    message = Telemetry.ActPos('4')
    if entry4 is not None:
        entry4.grid_forget()
        del entry4
    pos4 = launch.launcher(message)
    pos4deg = AngleConversion.RadToDeg(pos4)
    entry4 = Label(tlm_frame, text=str(round(pos4deg, 3)))
    entry4.grid(row=4, column=1, padx=5, pady=10)
    message = Telemetry.ActVel('1')
    if vel4 is not None:
        vel4.grid_forget()
        del vel4
    jv4 = launch.launcher(message)
    jv4deg = AngleConversion.RadToDeg(jv4)
    vel4 = Label(tlm_frame, text=str(round(jv4deg, 3)))
    vel4.grid(row=4, column=2, padx=5, pady=10)
    tlm_frame.after(237, update5)


def update5():
    global entry5
    global vel5
    message = Telemetry.ActPos('5')
    if entry5 is not None:
        entry5.grid_forget()
        del entry5
    pos5 = launch.launcher(message)
    pos5deg = AngleConversion.RadToDeg(pos5)
    entry5 = Label(tlm_frame, text=str(round(pos5deg, 3)))
    entry5.grid(row=5, column=1, padx=5, pady=10)
    message = Telemetry.ActVel('1')
    if vel5 is not None:
        vel5.grid_forget()
        del vel5
    jv5 = launch.launcher(message)
    jv5deg = AngleConversion.RadToDeg(jv5)
    vel5 = Label(tlm_frame, text=str(round(jv5deg, 3)))
    vel5.grid(row=5, column=2, padx=5, pady=10)
    tlm_frame.after(237, update6)


def update6():
    global entry6
    global vel6
    message = Telemetry.ActPos('6')
    if entry6 is not None:
        entry6.grid_forget()
        del entry6
    pos6 = launch.launcher(message)
    pos6deg = AngleConversion.RadToDeg(pos6)
    entry6 = Label(tlm_frame, text=str(round(pos6deg, 3)))
    entry6.grid(row=6, column=1, padx=5, pady=10)
    message = Telemetry.ActVel('1')
    if vel6 is not None:
        vel6.grid_forget()
        del vel6
    jv6 = launch.launcher(message)
    jv6deg = AngleConversion.RadToDeg(jv6)
    vel6 = Label(tlm_frame, text=str(round(jv6deg, 3)))
    vel6.grid(row=6, column=2, padx=5, pady=10)
    tlm_frame.after(237, update7)


def update7():
    global entry7
    global vel7
    message = Telemetry.ActPos('7')
    if entry7 is not None:
        entry7.grid_forget()
        del entry7
    pos7 = launch.launcher(message)
    pos7deg = AngleConversion.RadToDeg(pos7)
    entry7 = Label(tlm_frame, text=str(round(pos7deg, 3)))
    entry7.grid(row=7, column=1, padx=5, pady=10)
    message = Telemetry.ActVel('1')
    if vel7 is not None:
        vel7.grid_forget()
        del vel7
    jv7 = launch.launcher(message)
    jv7deg = AngleConversion.RadToDeg(jv7)
    vel7 = Label(tlm_frame, text=str(round(jv7deg, 3)))
    vel7.grid(row=7, column=2, padx=5, pady=10)
    tlm_frame.after(237, update)
```

```python


def move():
    angles = [sli1p.get(), sli2p.get(), sli3p.get(), sli4p.get(),
   sli5p.get(), sli6p.get(), sli7p.get()]
    Multiple.AllPos(angles)


def reset():
    module='100'
    GeneralCommands.Reset(module)


def home():
    module='100'
    GeneralCommands.Home(module)
    sli1p.set(value=0)
    sli2p.set(value=0)
    sli3p.set(value=0)
    sli4p.set(value=0)
    sli5p.set(value=0)
    sli6p.set(value=0)
    sli7p.set(value=0)


def stop():
    module='100'
    GeneralCommands.Halt(module)


def totem():
    Multiple.Totem()
    sli1p.set(value=0)
    sli2p.set(value=0)
    sli3p.set(value=0)
    sli4p.set(value=0)
    sli5p.set(value=0)
    sli6p.set(value=0)
    sli7p.set(value=0)


def opsafe():
    Multiple.Opsafe()
    sli1p.set(value=70)
    sli2p.set(value=30)
    sli3p.set(value=0)
    sli4p.set(value=70)
    sli5p.set(value=20)
    sli6p.set(value=50)
    sli7p.set(value=0)


def pose():
    global x_entry
    global y_entry
    global z_entry
    global roll_entry
    global pitch_entry
    global yaw_entry
    rtm = dg.read()
    # print(rtm)
    eex = rtm[0, 3]
    if x_entry is not None:
        x_entry.grid_forget()
        del x_entry
    x_entry = Label(tlm_frame, text=round(eex, 3))
    x_entry.grid(row=10, column=0)
    eey = rtm[1, 3]
    if y_entry is not None:
        y_entry.grid_forget()
        del y_entry
    y_entry = Label(tlm_frame, text=round(eey, 3))
    y_entry.grid(row=10, column=1)
    eez = rtm[2, 3]
    if z_entry is not None:
        z_entry.grid_forget()
        del z_entry
    z_entry = Label(tlm_frame, text=round(eez, 3))
    z_entry.grid(row=10, column=2)
    theta_x = math.atan2(rtm[2][1], rtm[2][2])
    cx = math.cos(theta_x)
    sx = math.sin(theta_x)
    theta_y = math.atan2(-rtm[2][0], sx * rtm[2][1] + cx * rtm[2][2])
    theta_z = math.atan2(-cx * rtm[0][1] + sx * rtm[0][2], cx * rtm
    [1][1] - sx * rtm[1][2])
    if roll_entry is not None:
        roll_entry.grid_forget()
        del roll_entry
    roll_entry = Label(tlm_frame, text=round(theta_x, 3))
    roll_entry.grid(row=10, column=3)
    if pitch_entry is not None:
        pitch_entry.grid_forget()
        del pitch_entry
    pitch_entry = Label(tlm_frame, text=round(theta_y, 3))
    pitch_entry.grid(row=10, column=4)
    if yaw_entry is not None:
        yaw_entry.grid_forget()
        del yaw_entry
    yaw_entry = Label(tlm_frame, text=round(theta_z, 3))
    yaw_entry.grid(row=10, column=5)


def allvel():
    angles = [en1v.get(), en2v.get(), en3v.get(), en4v.get(), en5v.get
    (), en6v.get(), en7v.get()]
    Multiple.AllVel(angles)

# Left frame --> COMMAND
arm_btn = Button(cmd_frame, text="ARM", width=7, command=Multiple.Arm)
arm_btn.grid(row=0, column=0, padx=5, pady=10)
stop_btn = Button(cmd_frame, text="STOP", width=7, command=stop)
stop_btn.grid(row=0, column=1, padx=5, pady=10)
quit_btn = Button(cmd_frame, text="QUIT", width=7)
quit_btn.grid(row=0, column=2, padx=5, pady=10)
reset_btn = Button(cmd_frame, text="RESET", width=7, command=reset)
reset_btn.grid(row=0, column=3, padx=5, pady=10)
move_btn = Button(cmd_frame, text="MOVE", width=7, command=move)
move_btn.grid(row=0, column=4, padx=5, pady=10)
totem_btn = Button(cmd_frame, text="TOTEM", width=7, command=totem)
totem_btn.grid(row=1, column=0, padx=5, pady=10)
opsafe_btn = Button(cmd_frame, text="OP-SAFE", width=7, command=opsafe
    )
opsafe_btn.grid(row=1, column=1, padx=5, pady=10)
park_btn = Button(cmd_frame, text="PARK", width=7)
park_btn.grid(row=1, column=2, padx=5, pady=10)
home_btn = Button(cmd_frame, text="HOME", width=7, command=home)
home_btn.grid(row=1, column=3, padx=5, pady=10)
allvel_btn = Button(cmd_frame, text="ALLVEL", width=7, command=allvel)
allvel_btn.grid(row=1, column=4, padx=5, pady=10)

update_btn = Button(cmd_frame, text="UPDATE", command=update, width=7)
update_btn.grid(row=0, column=6)

pos_label = Label(cmd_frame, text="Positions")
pos_label.grid(row=2, column=0)

jo1p = Label(cmd_frame, text="J 1")
jo1p.grid(row=3, column=0, padx=5, pady=10)
sli1p = Scale(cmd_frame, from_=-180, to=180)
sli1p.grid(row=4, column=0)
jo2p = Label(cmd_frame, text="J 2")
```

```
jo2p.grid(row=3, column=1, padx=5, pady=10)
sli2p = Scale(cmd_frame, from_=-180, to=180)
sli2p.grid(row=4, column=1)
jo3p = Label(cmd_frame, text="J 3")
jo3p.grid(row=3, column=2, padx=5, pady=10)
sli3p = Scale(cmd_frame, from_=-180, to=180)
sli3p.grid(row=4, column=2)
jo4p = Label(cmd_frame, text="J 4")
jo4p.grid(row=3, column=3, padx=5, pady=10)
sli4p = Scale(cmd_frame, from_=-180, to=180)
sli4p.grid(row=4, column=3)
jo5p = Label(cmd_frame, text="J 5")
jo5p.grid(row=3, column=4, padx=5, pady=10)
sli5p = Scale(cmd_frame, from_=-180, to=180)
sli5p.grid(row=4, column=4)
jo6p = Label(cmd_frame, text="J 6")
jo6p.grid(row=3, column=5, padx=5, pady=10)
sli6p = Scale(cmd_frame, from_=-180, to=180)
sli6p.grid(row=4, column=5)
jo7p = Label(cmd_frame, text="J 7")
jo7p.grid(row=3, column=6, padx=5, pady=10)
sli7p = Scale(cmd_frame, from_=-180, to=180)
sli7p.grid(row=4, column=6)

vel_label = Label(cmd_frame, text="Velocities")
vel_label.grid(row=5, column=0)

jo1v = Label(cmd_frame, text="J 1")
jo1v.grid(row=6, column=0, padx=5, pady=10)
en1v = Entry(cmd_frame, width=7)
en1v.grid(row=7, column=0)
jo2v = Label(cmd_frame, text="J 2")
jo2v.grid(row=6, column=1, padx=5, pady=10)
en2v = Entry(cmd_frame, width=7)
en2v.grid(row=7, column=1)
jo3v = Label(cmd_frame, text="J 3")
jo3v.grid(row=6, column=2, padx=5, pady=10)
en3v = Entry(cmd_frame, width=7)
en3v.grid(row=7, column=2)
jo4v = Label(cmd_frame, text="J 4")
jo4v.grid(row=6, column=3, padx=5, pady=10)
en4v = Entry(cmd_frame, width=7)
en4v.grid(row=7, column=3)
jo5v = Label(cmd_frame, text="J 5")
jo5v.grid(row=6, column=4, padx=5, pady=10)
en5v = Entry(cmd_frame, width=7)
en5v.grid(row=7, column=4)
```

```
jo6v = Label(cmd_frame, text="J 6")
jo6v.grid(row=6, column=5, padx=5, pady=10)
en6v = Entry(cmd_frame, width=7)
en6v.grid(row=7, column=5)
jo7v = Label(cmd_frame, text="J 7")
jo7v.grid(row=6, column=6, padx=5, pady=10)
en7v = Entry(cmd_frame, width=7)
en7v.grid(row=7, column=6)

# Right frame --> TELEMETRY
position_tlm = Label(tlm_frame, text="Position")
position_tlm.grid(row=0, column=1, padx=5, pady=10)
velocity_tlm = Label(tlm_frame, text="Velocity")
velocity_tlm.grid(row=0, column=2, padx=5, pady=10)

readpos_btn = Button(tlm_frame, text="READ POSITIONS", width=15,
    command=update)
readpos_btn.grid(row=0, column=4, padx=5, pady=10, columnspan=2)
readvel_btn = Button(tlm_frame, text="READ VELOCITIES", width=15)
readvel_btn.grid(row=1, column=4, padx=5, pady=10, columnspan=2)

joint1 = Label(tlm_frame, text="Joint 1")
joint1.grid(row=1, column=0, padx=5, pady=10)
# entry1 = Label(tlm_frame, text=sli7.get())
# entry1.grid(row=1, column=1, padx=5, pady=10)
joint2 = Label(tlm_frame, text="Joint 2")
joint2.grid(row=2, column=0, padx=5, pady=10)
# entry2 = Entry(tlm_frame, width=7)
# entry2.grid(row=2, column=1, padx=5, pady=10)
joint3 = Label(tlm_frame, text="Joint 3")
joint3.grid(row=3, column=0, padx=5, pady=10)
joint4 = Label(tlm_frame, text="Joint 4")
joint4.grid(row=4, column=0, padx=5, pady=10)
# entry4 = Entry(tlm_frame, width=7)
# entry4.grid(row=4, column=1, padx=5, pady=10)
joint5 = Label(tlm_frame, text="Joint 5")
joint5.grid(row=5, column=0, padx=5, pady=10)
# entry5 = Entry(tlm_frame, width=7)
# entry5.grid(row=5, column=1, padx=5, pady=10)
joint6 = Label(tlm_frame, text="Joint 6")
joint6.grid(row=6, column=0, padx=5, pady=10)
# entry6 = Entry(tlm_frame, width=7)
# entry6.grid(row=6, column=1, padx=5, pady=10)
joint7 = Label(tlm_frame, text="Joint 7")
joint7.grid(row=7, column=0, padx=5, pady=10)
# entry7 = Entry(tlm_frame, width=7)
# entry7.grid(row=7, column=1, padx=5, pady=10)
```

```

ee_label = Label(tlm_frame, text="EE")
ee_label.grid(row=8, column=0)

compute_pose = Button(tlm_frame, text="COMPUTE POSE", width=15,
    command=pose)
compute_pose.grid(row=8, column=4, padx=5, pady=10, columnspan=2)

x_label = Label(tlm_frame, text="x")
x_label.grid(row=9, column=0, padx=5, pady=10)
# x_entry = Entry(tlm_frame, width=7)
# x_entry.grid(row=10, column=0)
y_label = Label(tlm_frame, text="y")
y_label.grid(row=9, column=1, padx=5, pady=10)
# y_entry = Entry(tlm_frame, width=7)
# y_entry.grid(row=10, column=1)
z_label = Label(tlm_frame, text="z")
z_label.grid(row=9, column=2, padx=5, pady=10)
# z_entry = Entry(tlm_frame, width=7)
# z_entry.grid(row=10, column=2)
roll_label = Label(tlm_frame, text="roll")
roll_label.grid(row=9, column=3, padx=5, pady=10)
# roll_entry = Entry(tlm_frame, width=7)
# roll_entry.grid(row=10, column=3)
pitch_label = Label(tlm_frame, text="pitch")
pitch_label.grid(row=9, column=4, padx=5, pady=10)
# pitch_entry = Entry(tlm_frame, width=7)
# pitch_entry.grid(row=10, column=4)
yaw_label = Label(tlm_frame, text="yaw")
yaw_label.grid(row=9, column=5, padx=5, pady=10)
# yaw_entry = Entry(tlm_frame, width=7)
# yaw_entry.grid(row=10, column=5)


root.mainloop()
```

Bibliography

[1] J. Denavit. *A kinematic notation for lower-pair mechanisms based on matrices.* Journal of Applied Mechanics, 1955.

[2] *Modeling and control of Robot Manipulators.* McGraw-Hill International editions, Singapore, 1996.

[3] National Research Council. *Evaluation of the National Aerospace Initiatives.* National Academic Press, Washington, USA, 2004.

[4] Craig J. J. *Introduction to Robotics: Mechanics and Control.* Pearson Prendice Hall, Upper Saddle River, New Jersey, 2005.

[5] Lanzerotti L. J. *Assessment of Options for Extending the Life of the Hubble Space Telescope.* National Academic Press, Washington, USA, 2005.

[6] Texas Instruments. *Introduction to the Controller Area Network (CAN).* 2008.

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

[8] *Handbook of Space Technology.* John Wilex sons, Ltd, Unite Kingdom, 2009. Chap. 7.

[9] *EPOS-Using Robotics for RvD Simulation of On-Orbit Servicing Missions.* AIAA Guidance, Navigation and Control Conference, 2010.

[10] NASA. “On-Orbit Satellite Servicing Study: Project Report”. In: (2010).

[11] Britannica Educational Publishing. *Unmanned Space Missions.* Rosen Educational Services, New York, USA, 2010.

[12] Peng Zhang. *Advanced Industrial Control Technology.* 2010.

[13] National Research Council. *Limiting future collision Risk to Spacecraft: An Assessment of NASA's Orbital Debris programs.* National Academic Press, Washington, USA, 2011.

[14] Schunk GmbH. *Lightweight Arm LWA 3. Assembly and Operating manual.* 2011.

[15] Wolfhard Lawrenz. *CAN System Engineering. From Theory to Practical Application.* Springer, 2013.

[16] Amtec Robotics GmbH. *Data exchange with PowerCube. Description of PowerCube communication interfaces.*

[17] Northrop Grumman. *Mission Extension Vehicles Validate New Satellite Lifeline in Orbit.* URL: `https://www.nasaspaceflight.com/2020/07/mission-extension-vehicles-validate-lifeline/`.

[18] Northrop Grumman. *SpaceLogistics Life Extension Services.* URL: `https://www.northropgrumman.com/space/space-logistics-services/`.

[19] Sudhakar Maradana. *CAN Basics.* URL: `https://%20automotivetechis.wordpress.com/2012/06/01/can-basics-faq/`.

[20] Institute of Robotics and DLR Mechatronics. *OOS-SIM.* URL: `https://www.dlr.de/rm/en/desktopdefault.aspx/tabid-11675/#gallery/30051`.

Acknowledgements

At the end of this project, I'd like to take a moment to thank all the people who have played an important role during these months. First of all, I'd like to thank my supervisor, Prof. Chiaberge, for giving me the opportunity to carry out my thesis in such an important company.

Thanks to my co-supervisor, Andrea Merlo, for trusting me with this assignment and for letting me gain experience in an environment like the Robotics and Mechatronics group at TAS-I. I really appreciate the help and the support you gave me during these months.

Finally, I'd like to thank Ciro Napolitano and Genny Scalise: from the first day I came in, you have been welcoming and you have done more than I expected to help me. You have started as my tutors for this project but soon I started considering you my friends. Thanks to you and to all the friends at Building 77, coming to work every day was a pleasure.