FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Robot Controller for a Parallel Robot From conception to implementation

Thesis

Submitted to the Faculty of Fachhochschule Aachen
and Centro de Ingenieria y Desarrollo Industrial

BY

Eng. José Luis Germán-Félix

PRIMARY SUPERVISOR:

Gengis K. Toledo-Ramirez, Dr. Eng. M. Sc.

CO-SUPERVISOR:

José Antonio Estrada-Torres, PhD

In Partial Fulfillment of the requirements for the degree of Master of Science in Mechatronics

Santiago de Querétaro, Qro., Mexico, September 2018

Declaration of Autorship

I, José Luis Germán Félix, declare that this thesis, ”Robot Controller for a Parallel Robot, from conception to implementation”, is the result of my own work. Any part of this dissertation has not been previously submitted, in part or whole, to any university or institution for a degree or other qualification.

I confirm that all consulted work from others is attributed and the source is always given. Where the work is done with the help of others, these sources of help have been acknowledged.

Signed: ________________________________

Date: ________________________________

Acknowledgments

First of all, I want to thank my thesis supervisor Dr. Eng. Gengis K. Toledo-Ramirez, for giving me the opportunity to work in this project, and who, in addition to providing support, and guidance for the development of this thesis project, he helped me debug the software and provided with code libraries, which made it possible to develop this work.

Also, to my co-supervisor, Antonio Estrada-Torres, PhD, for its valuable advice and suggestions, during the development of this thesis.

To the Automated Systems Department in CIDESI, for providing a workplace and the equipment to develop this project, and also to M.C. Juán Noé Reyes Elías, who supported this project in many ways as division leader.

I am very grateful to CIDESI and CONACYT for providing me with the opportunity to study abroad.

Also, to Prof. Dr. Klaus-Peter Kämper and Prof. Dr. Ing. Jörg Wollert, for their help and support during my stay in Germany.

Thanks to Dipl.-Ing. Daniel Nottarp for the provided help and suggestions to this work.

To my parents, my sister, and my brother, for their unconditional love and support, and for motivating me to keep going in difficult times.

Finally, to all my friends from the Masters program, with whom I have created great memories.

Abstract

The field of robotics nowadays continues to grow and new applications are being developed. One limiting factor for the development of research of robotics is the closed architecture of robot systems. In particular, the closed architecture of robot controllers only allows basic levels of integration of robotic systems with external applications. In this thesis, it is developed a generic robot controller with an open architecture, and a modular approach. The methodology followed for this was the Systems Engineering's V Model. The Concept of Operations and system requirements are the starting point for the development work. The system is designed, considering available technologies, components, etc. In the implementation stage of the system, the hardware and software components are developed. Finally, testing and validation is performed, in order to ensure that the requirements are met. As an immediate result, it was achieved a controller that fulfills the requirements for a prototype with basic functionality. The open architecture of the model, and the modular design of the software makes it possible to extend and adapt the controller for future developments of the system, as well as for different applications.

Kurzfassung

Die Forschung dem Gebiet der Robotik wächst heute weiter und neue Anwendungen werden entwickelt. Ein limitierender Faktor für die Entwicklung der Robotik Forschung ist die geschlossene Architektur von Robotersystemen. Dies limitiert die Erweitung durch externen Zusatzfunktionen auf ein niedriges Integrationsniveau. In dieser Arbeit wird eine generische Robotersteuerung mit einer offenen Architektur und einem modularen Ansatz entwickelt. Die dafür verwendete Methodik ist das V-Modell aus dem Systems Engineering. Das Betriebskonzept und die Systemanforderungen sind der Ausgangspunkt für die Entwicklungsarbeit. Das System wurde unter Berücksichtigung der verfügbaren Technologien, Komponenten usw. entwickelt. In der Implementierungsphase des Systems wurden die Hardware- und Softwarekomponenten entwickelt. Schließlich wurden Validierungstests sowie Validierungen durchgeführt, um sicherzustellen, dass die Anforderungen erfüllt werden. Als unmittelbares Ergebnis wurde ein Controller entwickelt der die Anforderungen an einen Prototyp mit Basisfunktionalitäten zufrieden stellt. Die offene Architektur des Modells und der modulare Aufbau der Software ermöglichen es, den Controller für zukünftige Entwicklungen des Systems und für verschiedene Anwendungen zu erweitern und anzupassen.

Resumen

El campo de la robótica continúa creciendo y nuevas aplicaciones siguen siendo desarrolladas. Un factor limitante para el desarrollo de la investigación en la robótica es la arquitectura cerrada de los sistemas robóticos. En particular, la arquitectura cerrada de los controladores para robots sólo permiten niveles básicos de integración con aplicaciones externas. En este trabajo de tesis se desarrolla un controlador genérico para un robot con una arquitectura abierta, y un enfoque modular. Para ello, se siguió la metodología V-model de Sistemas de Ingeniería (Systems Engineering). Los conceptos de operación y requerimientos del sistema son el punto inicial para el desarrollo de este trabajo. El sistema fue diseñado, considerando tecnologías y componentes disponibles, etc. En la etapa de implementación del sistema, los componentes de hardware y software fueron desarrollados. Finalmente, se realizaron pruebas, y el sistema fue validado, para asegurar que los requerimientos hayan sido alcanzados. Como resultado inmediato, se logró desarrollar un controlador que cumple con los requerimientos para un prototipo con funcionalidad básica. La arquitectura abierta del sistema, y el diseño modular del software, hacen posible extender y adaptar el controlador para futuros desarrollos del sistema, así como para diferentes aplicaciones.

Contents

List of Figures

List of Tables

List of Abbreviations

AC	Alternating Current
AI	Artificial Intelligence
ALMA	Atacama Large Millimeter Array
CLI	Command Line Interface
ConOps	Concept of Operations
FPGA	Field Programmable Gate Array
GPOS	General Purpose Operating System
GUI	Graphical User Interface
IFR	International Federation of Robotics
MEMS	Microelectomechanical Systems
PC	Personal Computer
PI	Physik Instrumente
ROS	Robot Operating System
RTAI	RealTime Application Interface for Linux

Chapter 1

Introduction

Robotics nowadays continues to be an important developing field within engineering and related areas such as software for Artificial Intelligence (AI). It is true that industrial robotics has been used in many industries for the last three decades; nevertheless, new approaches for robotics are being developed, such as collaborative robotics, reconfigurable manufacturing systems, industry 4.0, among others.

The two most widely used types of industrial robotic systems are serial manipulators, such as robot arms, and parallel manipulators, such as delta robots. Parallel manipulators offer some advantages, compared with serial robots, such as higher load capacity, mechanical stiffness and precision, due to its parallel structure.

One important type of parallel manipulator is the Stewart-Gough platform, also called Hexapod. It consists of a fixed platform and a mobile platform, connected by six prismatic actuators. This configuration provides six degrees of freedom (three translational and three rotational) on its mobile platform.

Stewart-Gough platforms are finding an increasing number of applications, due to their low inertia and high precision. Some interesting examples for applications are:

- Process automation: Hexapods are used for aligning the smallest optical components in the latest Silicon Photonics production processes, as controlling automated labeling machines, and positioning entire body parts for automotive production.
- Manufacturing and Quality Assurance: Hexapods are used in micro assembly processes, where automated 6D alignment systems take care of important tasks during testing and manufacturing of accessories for Microelectomechanical Systems (MEMS) based camera sensors and photonics components, including the alignment of optical fibers and fiber arrays, for the production of optical lenses.
- Motion Simulators: Motion simulators make high demands on motion dynamics. They repeatedly perform defined motion cycles, for example, for quality assurance and function monitoring of products in mobile use. Typical examples for this include inspection systems for acceleration or gyroscopic sensors, like those used in smartphones, cell phones,

and cameras for detecting changes in position. They are tested by means of a specified motion pattern.

- Astronomy: A typical radio telescope antenna consists of a main reflector (the large dish) and a secondary, sub-reflector that needs to be aligned properly in regard to the main dish for maximum efficiency. This precision alignment task is carried out by special Hexapod 6-axis parallel positioning systems from PI in the Atacama Large Millimeter Array (ALMA) telescope [1].

An essential component of any robotic system is the robot controller. This is, the electrical, electronic, and software components that allow the robot to move and perform different actions. All of the industrial robot controllers have a closed architecture, i.e. their technical specifications are not made public, and any extension or modification of the system's functionalities are not allowed.

The aforementioned issues, combined with the high price of industrial robotic systems, become a challenge for researchers in the area of robotics, especially in developing countries.

One way to overcome these challenges is with the development of a generic controller with an open architecture, which can be modified and extended to suit many different applications. This way, affordable robotic systems could be used by researchers and institutions to further develop the area of robotics.

This thesis project consists in the design and implementation of a basic functional controller prototype, with an open architecture, for a Stewart-Gough robot, which can be used for demonstration and research purposes.

1.1 Motivation

Industrial robotics are designed and manufactured by a few world-wide companies, all of them provide high-tech and costly proprietary technology. It is true that such products are suitable for many industrial applications, especially those that involve repetitive, dangerous tasks or high loads. Nevertheless, for most applications with less stringent requirements (of accuracy, for example) these solutions result impractical because of their cost or the impossibility to be adapted to a specific need. For research and academy an industrial robot is suitable only for integration with other technologies, but almost impossible to modify or improve at the design level. Thus, the development of a robot system with own technology allows a wide range of possibilities for researchers, academics and others to adapt the system to many more applications, such as industry, instrumentation and many others. Moreover, this kind of developments impact immediately in engineering capabilities for those institutions and people involved in such projects.

For this reason, since the year 2015, it is being developed at CIDESI a research project, consisting on a Stewart-Gough platform made with self-developed technology that has an acceptable quality-price relation for national as well as industry applications. This project is called the CIDESI Hexapod and is currently in Phase 1 of development. In order to complete this phase, a software controller for the robot was required.

1.2 Objectives

This section presents the general and specific objectives of the thesis project.

1.2.1 General objective

Design and implement a functional controller prototype for the CIDESI Hexapod, capable of performing basic tasks of a robotic system, with an open architecture, using available open-source software, and with consideration on scalability to add more functionalities in future developments of the system.

1.2.2 Specific objectives

Several specific objectives are defined, in order to provide milestones that ultimately lead to the fulfillment of the general objective.

- Thesis protocol document
- Research of the state of the art and the technique
- Controller design
- Implement Controller Software
- Test software modules
- Pass the System validation
- Master Thesis
- Thesis dissertation

1.3 Hypothesis

A software controller for the CIDESI Hexapod robot with basic functionalities can be designed with available technology primarily for experimental, research and demonstration purposes, with an open architecture, and using open-source software development tools, able to perform basic tasks in the area of robotics.

1.4 Methodology

In order to ensure the quality and success of the project, the “CIDESI Systems Engineering’s V-model” was used. It is a system development process, which emphasizes the relationship between the design phases and the testing phases of the system. It is shown in Fig. 1.1.

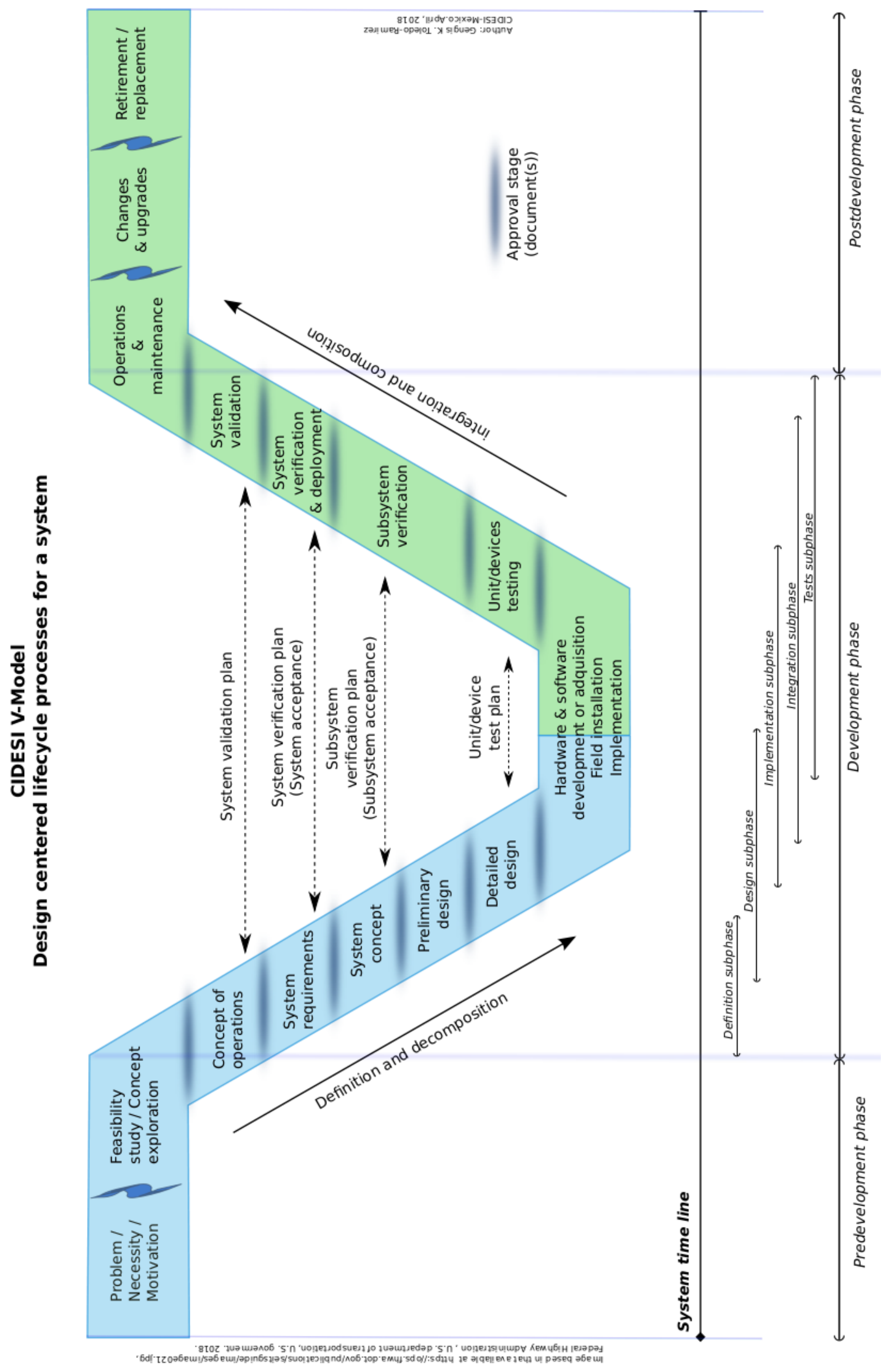

CIDESI V-Model
Design centered lifecycle processes for a system
Problem / Necessity / Motivation
Feasibility study / Concept exploration
Concept of operations
System requirements
System concept
Preliminary design
Detailed design
Hardware & software development or adquisition
Field installation
Implementation
Unit/devices testing
Subsystem verification
System verification & deployment
System validation
Operations & maintenance
Changes & upgrades
Retirement / replacement
System validation plan
System verification plan (System acceptance)
Subsystem verification plan (Subsystem acceptance)
Unit/device test plan
Definition and decomposition
Integration and composition
Approval stage (document(s))
Author: Gengis K. Toledo-Ramirez
CIDESI-Mexico.April, 2018
Image based in that available at https://ops.fhwa.dot.gov/publications/seitsguide/images/image021.jpg, Federal Highway Administration, U.S. department of transportation, U.S. goverment. 2018.
System time line
Definition subphase
Design subphase
Implementation subphase
Integration subphase
Tests subphase
Predevelopment phase
Development phase
Postdevelopment phase

Figure 1.1: CIDESI Systems Engineering's V-Model [2].

First, the Concept of Operations (ConOps) and system's requirements were established. They describe the expected functionalities for the system in terms of its users. After this, ConOps and requirements were analyzed in order to develop a concept design. In this stage, some important decisions had to be made, such as the selection of hardware components, programming languages, software development tools, etc.

The analysis of requirements, as well as available technologies and constrains led to the design of the system, taking into account hardware components, communication interfaces and software architecture.

The software architecture was designed as a set of individual software modules, and its development was the bulk of this project. Each software module was written and tested separately (unit testing). Once a logical or related group of software modules had been written, integration tests were performed: the software modules and their mutual interaction were tested and validated together. Since a small number of modules was being tested at once, the testing process was automated, using a unit test framework. Once the integration testing was successful, the system verification was performed, in order to assure that the system requirements were met.

Finally, the project was documented. Two types of documentation were created: one is the documentation for the user and operator of the robot, in the form of a user manual. The other type of documentation is for future developers of the software. This is the documentation of the source code, and it was done as a browser-based documentation.

Chapter 2

Theoretical background

This section presents an introduction to the topics of Parallel robots, the Stewart-Gough platform, and robot controllers, followed by a literature review of scientific publications and related works.

2.1 Parallel robots

Parallel robots, also sometimes called hexapods or Parallel Kinematic Machines, are closed loop mechanisms presenting very good performances in terms of accuracy, rigidity and ability to manipulate large loads. They have been used in a large number of applications ranging from astronomy to flight simulators, and are becoming increasingly popular in the machine-tool industry.

Formally, parallel robots have been defined as follows:

A parallel robot is made up of an end-effector with n degrees of freedom, and of a fixed base, linked together by at least two kinematic chains. Actuation takes place through n simple actuators [3].

Some examples for parallel robots include:

- FlexPicker robot from ABB
- Adept Quattro robot from Omron
- F-200iB robot from FANUC
- Agile Eye from from Laboratoire de robotique, Université LAVAL

2.2 The Stewart-Gough Platform

The Stewart-Gough Platform is a spatial 6-degree of freedom parallel manipulator. A general Stewart-Gough platform is shown in Fig. 2.1. Six identical limbs connect the moving platform to the fixed base by spherical joints at points B_i and A_i, $i = 1, 2, ..., 6$, respectively.

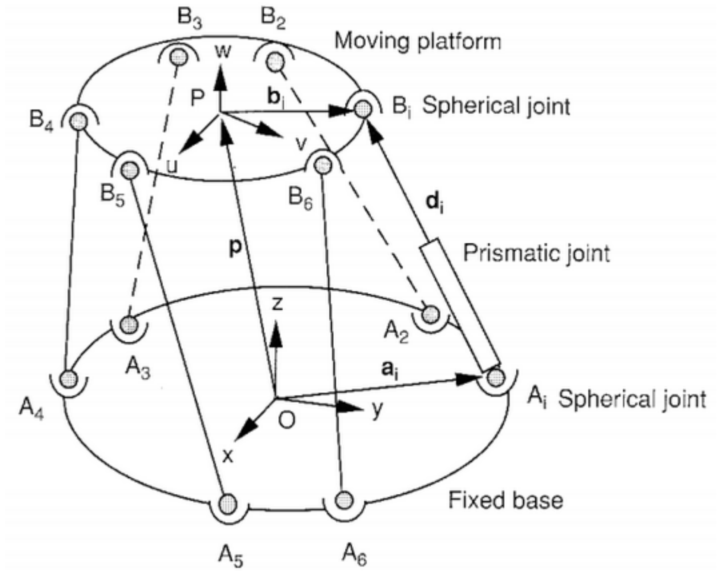


Figure 2.1: General Stewart-Gough Platform, taken from [4].

2.3 Inverse Kinematics of the Stewart-Gough Platform

The inverse Kinematics consists in establishing the value of the joint coordinates corresponding to the end-effector configuration. This relation is usually simple for parallel robots.

In Fig.2.1, two coordinate systems, frames $A(x, y, z)$ and $B(u, v, w)$, are attached to the fixed base and moving platform, respectively. The transformation from the moving platform to the fixed base can be described by the position vector p and the rotation matrix R_B^A of the moving platform. Let a_i and b_i be the position vectors of points A_i and B_i in the coordinate frames A and B, respectively. These vectors are defined by the geometry of the Stewart-Gough platform. We can write a vector loop equation for the ith limb of the manipulator as follows:

$$\overline{A_iB_i} = p + R_B^A b_i - a_i$$

The position vector p and rotation matrix R_B^A of frame B with respect to A are given (this is the desired pose) and the limb lengths correspond to the actuator lengths.

2.4 Review of Literature

This section presents the literature review of scientific publications. First, it was researched topics related to the Stewart-Gough Platform, robot controllers, and their architectures. It was found that many of these applications have real-time constraints. As a consequence, many publications focus on the development of real-time controllers for robotics applications. This led to an additional research on the topics of real-time controllers and real-time operating systems.

2.4.1 A platform with six degrees of freedom

In 1965, D. Stewart [5] described a platform with six degrees of freedom, controlled by six independent motors. Stewart proposed that this platform could be used for flight simulation, as a basis of design for a machine tool, and as a platform for training helicopter pilots.

Since the publication of Stewart's paper, the Stewart-Gough platform has been a very popular research topic, and has been applied in various fields, such as flight simulation, mechanical testing, telescopes, machine tool technology, and orthopedic surgery.

2.4.2 Position control of a Stewart platform using inverse dynamics control with approximate dynamics

Se-Han Lee et al. [6], realized the position control based on the inverse dynamics controller. Based on the characteristics of small motion range, the motion matrices can be approximated to constant ones. The modeling errors caused by such approximation are compensated by H_∞ controller that treats the error as disturbance.

The results in this paper show that the inverse dynamics control with approximate dynamics combined with the H_∞ controller provide better results than that with a full dynamics and a simple PID controller.

2.4.3 A Real-Time Control System For Industrial Robots And Control Applications Based On Real-Time Linux

In [7], a real-time control system for an industrial manipulator is presented. This control system was developed under the RealTime Application Interface for Linux (RTAI).The main goal was to build a flexible and highly configurable robotic setup to quickly test new control algorithms and to easily (and safely) use the robot in different manipulation tasks. The software developed for the control of the robot was divided into two modules: a real-time module, which executes the control algorithms and communicates directly with the plant (robot), and a non real-time application that provided a user interface for the real-time module.

2.4.4 Real-time control systems: a tutorial

In [8], Gambier presents a general introduction to the design of real-time control systems, and highlights important issues about real-time systems that should be taken into account at the moment to implement digital control. Gambier provides a definition for a real-time system:

A real-time system is one in which the correctness of a result not only depends on the logical correctness of the calculation but also upon the time at which the result is made available.

Other important topics are discussed, such as real-time operating systems, schedulers, and common mistakes when programming real-time operating systems.

2.4.5 A real-time walking robot control system based on Linux RTAI

Deng Liming [9] developed a real-time control system for a walking robot. The Linux RTAI was selected as the operating system. The control system consisted of a motion controller, connected with an industrial Personal Computer (PC), and several amplifiers and motors.

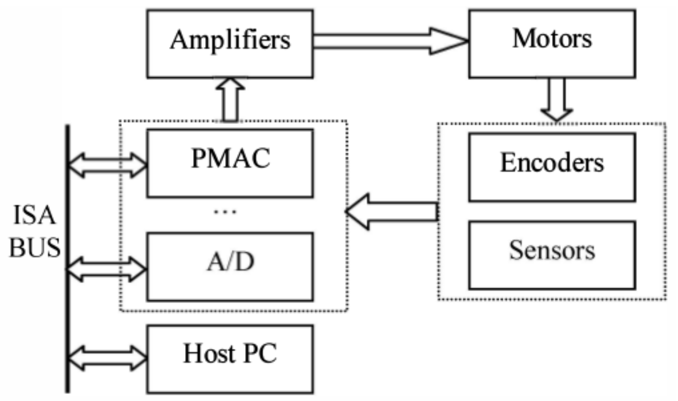


Figure 2.2: The architecture of the control system developed by Deng Liming, taken from [9].

The industrial PC responds to the high level planning and calculation, such as trajectory generation, status monitoring and process sensor information.

The real-time control software consists of four kinds of threads. The main thread is responsible for initialization and deletion of all other threads. The signal gathering threads try to gather signals from the sensors at a fixed frequency. The trajectory planning thread is the core of the control software. It incorporates all the relative sensors' signals, processing them as desired, and it also receives and responds to commands from a remote GUI.

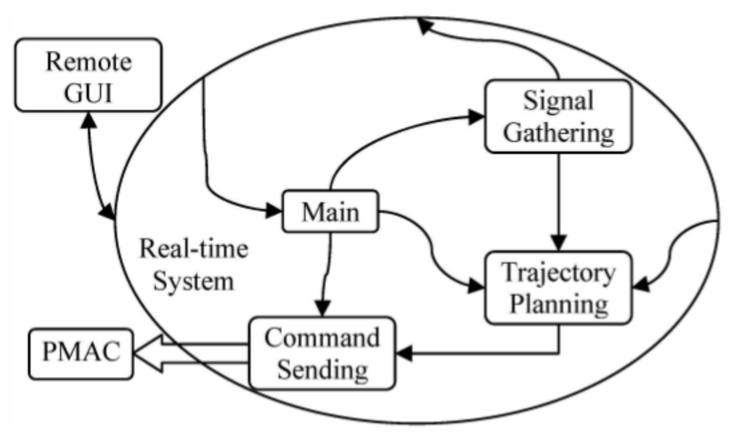


Figure 2.3: The architecture of the control software developed by Deng Liming, taken from [9].

2.4.6 Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application

In [10], A. Barbalace et al reported a set of performance measurements executed on VMEbus MVME5500 boards equipped with MPC7455 PowerPC processor, running four different operating systems: Wind River VxWorks, Linux, RTAI, and Xenomai. Interrupt latency, rescheduling and inter-process communication times are compared in the framework of a sample real-time application.

For evaluating the performance, they used a test program, which acquires 64 channels from an ADC converter via the VME bus and produces a single output on a DAC board. This simple program allows testing of interrupt latency and rescheduling time, since the ADC generates an interrupt when a new set of digitized data is ready. By measuring with an oscilloscope the delay between an input signal and the output of the system, it is possible to perform a non-perturbative measurement of the overall system performance.

Overall the performance measurements are similar with about 5% difference between the fastest (VxWorks) and the slowest (Xenomai). Results show that both RTAI and Xenomai represent valid alternatives to VxWorks for the implementation of real-time control systems.

2.4.7 RT-ROS: A real-time ROS architecture on multi-core processors

In [11] it is presented a real-time software architecture based on the Robot Operating System (ROS) called RT-ROS on multi-core processors. RT-ROS provides an integrated real-time/non-real-time task execution environment so that real-time and non-real-time ROS nodes can be separately run on a real-time OS and Linux, respectively, with different processor cores. This way, real-time tasks can be supported by real-time ROS nodes on a real-time OS, while non-real-time ROS nodes on Linux can provide other functions to ROS.

Nuttx was used as the real-time platform, and Linux as the general-purpose OS. The architecture of RT-ROS is shown in Fig. 2.4.

At the application layer, ROS nodes are divided into real-time and non-real-time nodes. ROS nodes communicate with each other via shared memory. At the hardrware level, hardware resources are divided so non-real-time ROS nodes and real-time ROS nodes are separately run based on General Purpose Operating System (GPOS) or RTOS, respectively.

RT-ROS was applied to control a 6-DOF modular manipulator. The experimental results show that RT-ROS can effectively provide real-time support for the ROS platform with high performance.

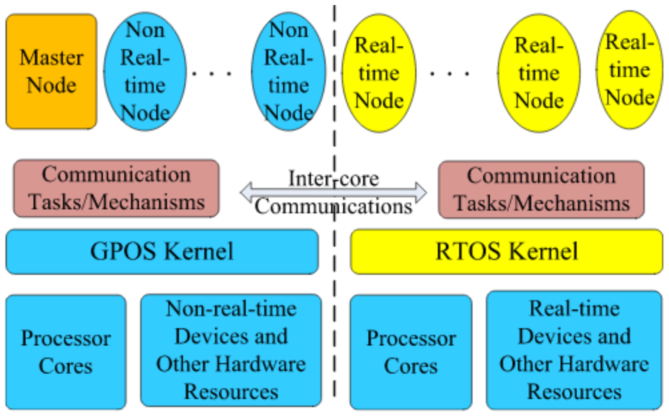

Master Node
Non Real-time Node
. . .
Non Real-time Node
Real-time Node
. . .
Real-time Node
Real-time Node
Communication Tasks/Mechanisms
Inter-core Communications
Communication Tasks/Mechanisms
GPOS Kernel
RTOS Kernel
Processor Cores
Non-real-time Devices and Other Hardware Resources
Processor Cores
Real-time Devices and Other Hardware Resources

Figure 2.4: The system architecture of RT-ROS, taken from [11].

2.4.8 Control System Based on FreeRTOS for Data Acquisition and Distribution on Swarm Robotics Platform

In Docekal's [12], the real-time operating system FreeRTOS is implemented into a 32 bit ARM microcontroller for the control system of a swarm robotics platform. The main task is to ensure the communication with different types of sensors, data processing, and also motion control of the robot. The system architecture is shown in Fig. 2.5.

Hardware description. The low level control, interfacing with sensors and actuators in the system was realized with the use of the 32 bit microcontroller STM32F4 with core ARM Cortex M4. The high level processing of the system, which includes swarm algorithm and path planning, was formed by the Raspberry Pi 3. The Raspberry Pi 3 includes the WiFi and Bluetooth modules. An external camera module with a resolution of 8 Megapixel was connected to the Raspberry Pi.

Firmware description. The FreeRTOS was selected for the programming of the microcontroller in order to ensure timely execution of all tasks. The interaction between tasks was realized using the facilities provided by FreeRTOS: queues, mutexes, and shared memory, etc. Each task was assigned different priority. There are different threads related to interfacing with the sensors.

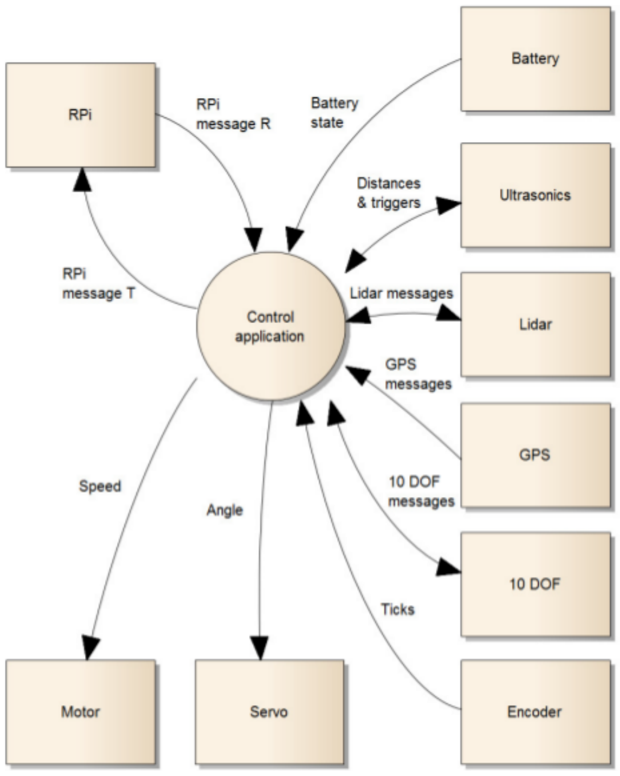
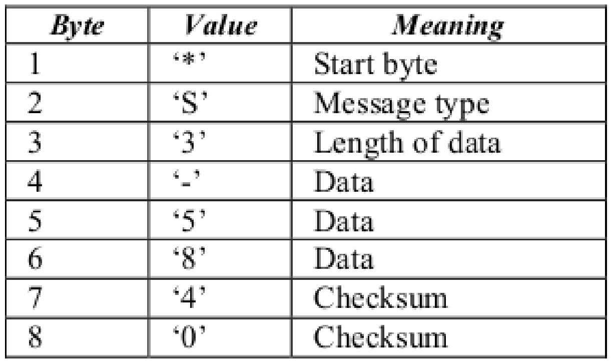


Figure 2.5: Docekal's system architecture, taken from [12].

These threads write the sensor information to a shared memory, and another thread can read this memory and send it to the Raspberry Pi.

Communication with Raspberry Pi. The data exchange between the microcontroller and the Raspberry Pi was realized using the UART periphery. When a new character (byte) is received, a hardware interrupt is triggered, and the character is saved into a frame buffer. The message frame is described in Fig. 2.6.



Byte	***Value***	***Meaning***
1	'*'	Start byte
2	'S'	Message type
3	'3'	Length of data
4	'-'	Data
5	'5'	Data
6	'8'	Data
7	'4'	Checksum
8	'0'	Checksum

Figure 2.6: Message Frame description for communication, taken from [12].

Additionally, an application for smartphones was used for data visualization and for robot control.

2.4.9 An FPGA-Based Open Architecture Industrial Robot Controller

In [13], a field programmable gate array-based motion control system is developed as part of an open-architecture and vendor-independent control system.

The proposed robot control architecture comprises a standard PC and a Field Programmable Gate Array (FPGA) based interface which involves a lower cost. Additionally, the Linux OS in addition with the real-time application interface (RTAI) is used.

The FPGA is responsible for decoding feedback position data, computing the control output, and communication signals, and transmitting such data to the actuators, whereas the PC interpolates movements, computes kinematics solutions, and transmits the position reference for each joint while it receives its current position. The complete system is shown in Fig. 2.7.

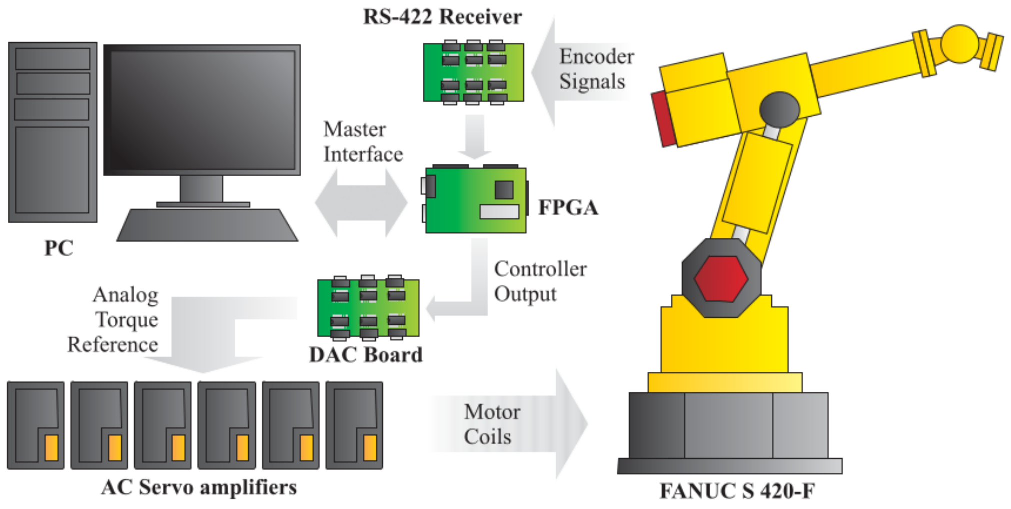


Figure 2.7: Robot controller hardware setup developed by Martínez-Prado, taken from [13].

2.5 Commercial Study on the field of Stewart Gough Platform

This section covers a commercial study on robotics and Stewart Gough Platforms, as well as different applications for this kind of robots.

2.5.1 Robot sales and market forecast

According to the International Federation of Robotics (IFR), the automotive industry is the major customer of industrial robots with a share of 35% of the total supply in 2016. It is forecasted that 521,000 industrial robot units will be sold world-wide in 2020, as shown in Fig. 2.8.

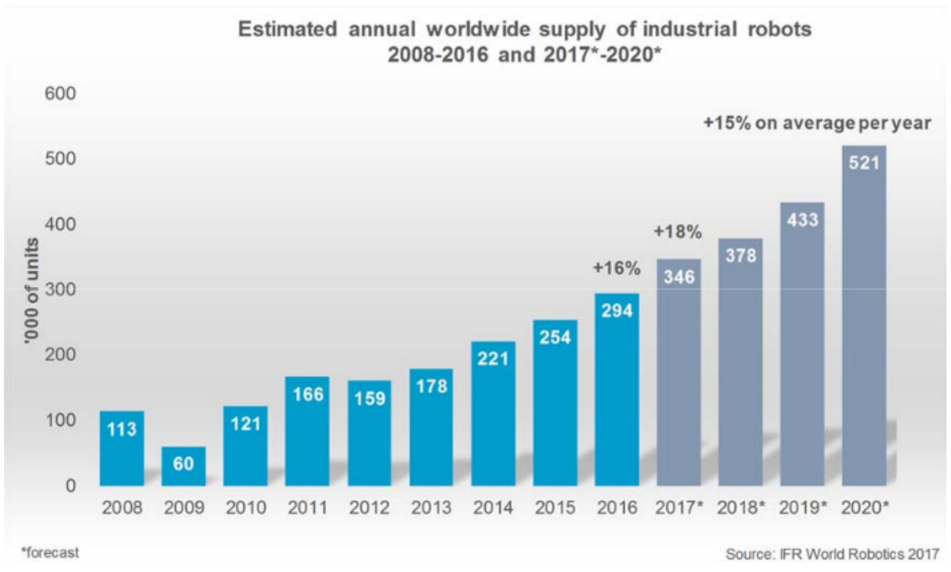


Figure 2.8: Estimated annual worldwide supply of industrial robots 2008-2016 and 2017-2020 [14].

2.5.2 Commercially available Stewart Gough Platforms

This section describes commercially available Stewart Gough Platforms from two large companies. These have been used in many different industrial, scientific, research, and medical fields, as will be described next.

2.5.2.1 Physik Instrumente

Physik Instrumente (PI) is one of the largest producers of Hexapods in the world. Their hexapods solutions range from tiny hexapods with workspace dimensions of 40 x 40 x 13 mm and a load capacity of 5 kg, up to hexapods with a load capacity larger than 1000 kg [15].

2.5.2.2 FANUC F-200iB

The F-200iB is a six degrees of freedom servo-driven parallel link robot designed for use in a variety of manufacturing and automotive assembly processes.

This robot has six degrees of freedom, powered by six Alternating Current (AC) servo motors and precision gear drives. It contains a fail-safe brake on each leg. It has a mechanical weight of 190 kg and a payload of 100 kg, allowing a motion speed of 300 mm/s in the vertical direction and 1500 mm/s in horizontal (X & Y) direction. It can be mounted in floor or inverted (upside-down) positions [16].

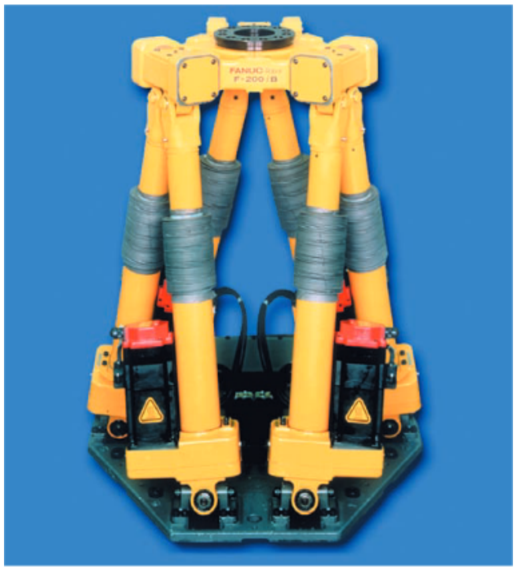


Figure 2.9: The FANUC F-200iB robot [16].

This chapter presented the theoretical background of the project followed by the review of literature. For the review of literature, it was searched for relevant papers and information related to the development of robot controllers, in particular for the Stewart-Gough platform, including the operating systems, software architecture structures, and hardware platforms used for the system. It was found that there are almost no publications related to the internal structures of industrial robot controllers. In contrast, a lot of the research is related to the implementation of real-time operating systems, their use in embedded devices, and different control techniques and algorithms for robotics applications.

Chapter 3

CIDESI Hexapod project

This chapter presents the CIDESI Hexapod project, the different phases of development, the previous work that has been done by members of the Automated Systems Department at CIDESI, and finally the concept of operations and requirements for the development of this thesis project.

3.1 The CIDESI Hexapod robotic system

The CIDESI Hexapod robotic system is a project that aims to develop an industrial grade robotic system that provides functionalities similar to those usually found in an industrial robot.

Similar to most commercial robotic systems, the CIDESI Hexapod is designed to be operated by a human operator (the User), or by another system or device (the Application) like, e.g., a vision system, as shown in Fig. 3.1.

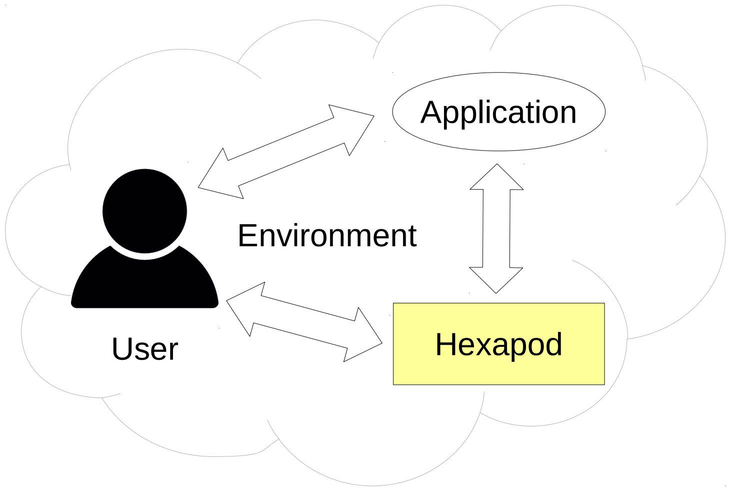


Figure 3.1: The CIDESI Hexapod system

The Hexapod robot system consists of the Robot (mechanical structure), the controller, a Teach

Pendant, and a user Documentation, as shown in Fig. 3.2. These are the basic components that are generally included in a commercial robotic system.

The preliminary as well as detailed mechanical design of the robot has already been developed, and in order to achieve a functional prototype, a robot controller is needed. This thesis project involves the development of the robot controller (and its corresponding Documentation) for the Phase I of the CIDESI Hexapod. It includes the hardware elements of the controller and the controller software, which runs in a PC.

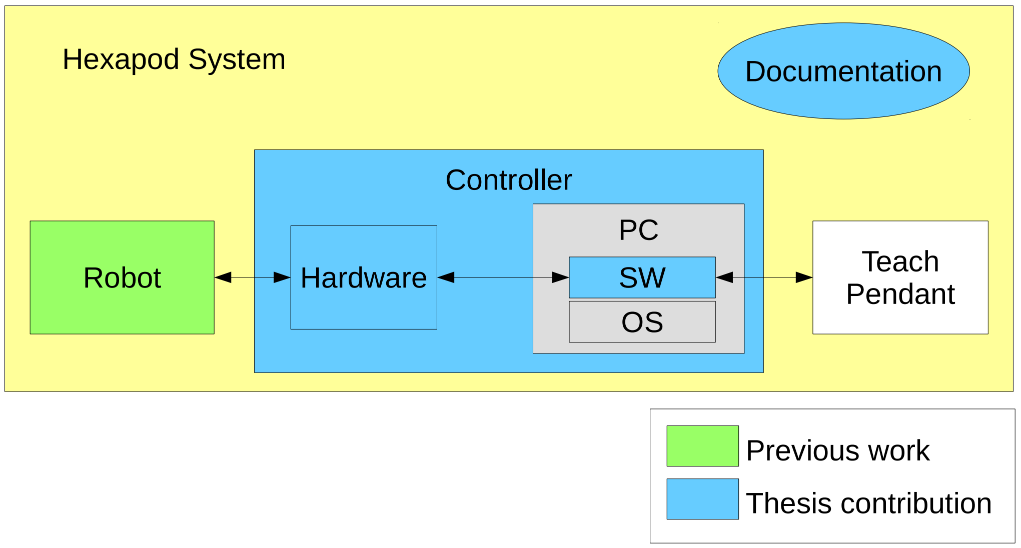


Figure 3.2: The components included in the Hexapod System

The robot component (mechanical structure and electrical wiring) is a Stewart-Gough Platform, consisting of a fixed base and a movable platform, see Fig. 3.3. The base and platform are connected by six linear actuators.

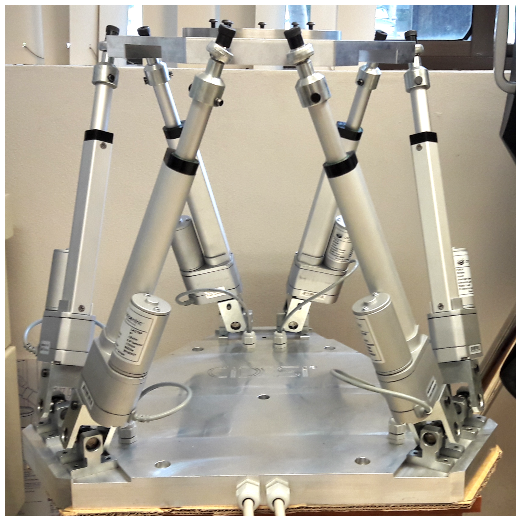


Figure 3.3: CIDESI Hexapod Robot.

3.2 CIDESI Hexapod development phases

The CIDESI Hexapod is a large project, and for its development, it is divided in four phases, as shown in Table 3.1.

	Phases			
Parameter	I	II	III	IV
Accuracy	1mm	100um	1um	
Payload	10Kg	100Kg	500Kg	
Actuator	GlideForce (LACTP6P-12V-20)	To be Decided	CIDESI Linear Actuator	
Estimated cost (MXN)	<80,000	<300,000	<900,000	
Objective	Experimental prototype	Low capabilities app.	Normal app.	High capabilities app.

Table 3.1: Development phases for the CIDESI Hexapod project, adapted from [17].

In the Phase I, the robot is mainly intended as an experimental prototype, and the use of actuators GlideForce LACTP6P-12V-20 is one of the requirements for this project. In future developments of the system, it is planned to replace these actuators with CIDESI's own linear actuators, as well as the development of a Teach Pendant.

3.3 Previous work on the CIDESI Hexapod System

This section describes the research and development work that existed on the CIDESI Hexapod system previous to the development of this thesis project.

3.3.1 Industrial Robot project proposal

In 2012, Gengis K. Toledo-Ramirez proposed the design of a six degree-of-freedom industrial robot at CIDESI Nuevo León, Mexico. The goal was to aid national industry with the reduction of manufacturing costs and by increasing the quality and efficiency. In the year 2015, he also proposed a research project named "Development of technologies for a Stewart-Gough Platform", to be developed by the Research and Development group of the Automated Systems Division at CIDESI Querétaro. The goal was to develop a robotic platform at an experimental level with enough efficiency to be of interest to the research, academic and industrial sectors.

The Automated Systems Division director, Noé Reyes, has supported this project since.

3.3.2 Development of a Demo for a Hexapod robot

This work was done by Ernesto Salazar Joya [18]. The main activity was to develop the programming of three demos(programs) for the FANUC F-200ib robot. In the first program, the objective was to visualize the maximal positions of displacement and rotation of the robot, as a first demonstration. In the second program, it is shown how to program the robot to perform industrial tasks, using simulated digital inputs. Finally, the third program consisted in the communication of a robot with an industrial vision system.

3.3.3 Hexapod system for motion simulator

Víctor David Acosta-Abraham [19] developed a study and preliminar draft for a truck simulator mechanism. This mechanism is based on a Stewart-Gough platform and its purpose is to train truck drivers in a simulated environment that recreates the forces to which a truck is subject when operating. A prototype using servo-motors and an Arduino was developed. A LabVIEW interface was used to solve the inverse kinematics equations and communicating the data to the Arduino to drive the servomotors of the prototype.

3.3.4 Development of an alpha prototype for a Hexapod robot

Gerson Andrés Díaz López [20] developed a concept design for the Hexapod system. This concept design included ideas for the mechanical structure of the robot, the control system for the robot and communication possibilities for the system.

3.3.5 Preliminary Design of a Stewart-Gough platform

Rubén Ordaz Madrigal [21] developed the preliminary design for the mechanical structure of the CIDESI Hexapod system. In this work, the mechanical parts were selected, including the joints and actuators, and a preliminary design was developed with these components.

3.3.6 Detailed Design of a Stewart-Gough platform

José Ramón Barajas-Fletes [22], based on the works by Ordaz and guided by Noé Reyes, developed the detailed design for the CIDESI Hexapod. In his design, some parts were changed, and the CAD model was refined. Additionally, motion simulations were performed in order to determine the workspace capabilities of the robot.

3.3.7 Robot assembly

The robot components were fabricated according to the detailed design by Barajas. The robot assembly, as well as the main mechanical parts, are shown in Fig. 3.4. Once all components were ready, the robot was assembled.

First, the mechanical components of the robot were assembled: the parts that form the universal joints were assembled and fixed to the base platform of the robot. Then the actuators were assembled to the universal joints and secured in place, and the head couplings were installed on the top of the shaft of the actuators. After this, the spherical joints were screwed in the mobile platform and secured with bolts, and the coupling disc was assembled to the mobile platform. Finally, all the actuators were assembled to the spherical joints and secured in place with screws.

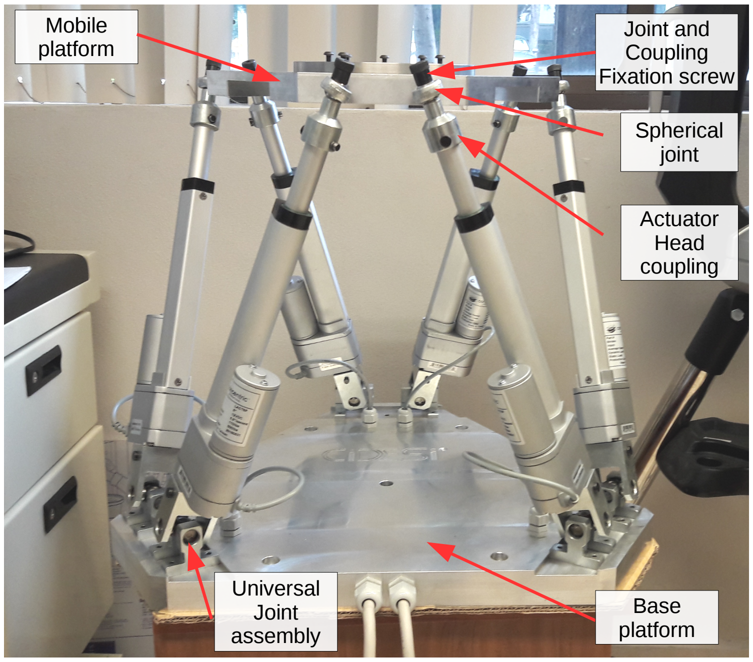


Figure 3.4: Hexapod assembly

3.4 Concept of Operations

In this section, the system's ConOps are presented. They describe the expected functionalities and operation policies for a system in terms of its users. The complete ConOps were developed in [17]. Here, only the concepts relevant to the robot controller are retrieved.

3.4.1 Operation policies and constrains

The system, in phase I, is not intended to be operated continuously, but for short time cycles, executing sample tasks and never continuous production cycles.

3.4.2 Usage

The prototype will be used in Laboratory or similar environments for demonstration, experimentation and testing purposes. The robot software and its programming tools must be simple but at the same time provide programming power on different levels, from high level (user or application level), intermediate level, and low level. It is desired to integrate as user interfaces, augmented reality systems, virtual reality, cyber-physical systems, Industry 4.0, Internet of Things, as well as applications for the Android operating system.

3.4.3 Final users and related persons

Operators. The direct users of the robot are its operators. They are capable of starting, moving and programming the robot by means of a "Teach Pendant", and perform basic maintenance tasks. They are required to have technical knowledge in robotics and specific training.

Technicians. In addition to operating the robot, a technician is capable of mounting applications for the robot; program the robot via a programming language; dismount the robot for transportation and mount it again; disassembly and reassembly of the robot; part replacing. For this, it is required robotics knowledge at the graduate level as well as training and the corresponding documentation.

Administrators. They may or may not be operators or technicians, but they have the responsibility of the robot, and therefore they can and must allow other users to work on the robot.

3.5 System Requirements

This section presents the requirements for the CIDESI Hexapod controller. These requirements were the main input and starting point for the thesis project.

According to the Systems Engineering's V-model, one of the initial phases in the lifecycle process for a system is the establishment of the system requirements.

The system requirements are formal statements that establish the characteristics of a system, and what the system must be able to accomplish. The system requirements were established in [17] and, together with an analysis of possible solutions and available technologies and resources, they lead to a concept design.

3.5.1 List of controller-related requirements

[Req 6.3.2] The system must be able to move each of its actuators independently within the maximum specified limits.

[Req 6.3.4] The system must be able to execute translational movements on its mobile base in the three coordinated axis within its specified limits.

[Req 6.3.5] The system must be able to execute rotational movements on its mobile base with respect to the three coordinated axis within its specified limits.

[Req 6.3.6] The system must have two working modes: 1) Manual mode and 2) Automatic Mode. In the "Manual mode" the prototype will receive direct commands from the user and execute them immediately. In "Automatic mode" the prototype will execute a previously programmed routine.

[Req 6.3.7] The prototype must indicate at least visually the following states: 1) On, 2) Moving, 3) Alarmed.

[Req 6.3.8] The prototype shall have six degrees of freedom (three translational and three rotational) within its workspace and mechanical limits.

[Req 6.3.9] Every valid point in the workspace shall be defined by six coordinates (three translational and three rotational).

[Req 6.3.10] Translational dimensions shall be given in millimeters (mm) and rotational dimensions shall be given in degrees (°).

[Req 6.3.11] The actuators shall be moved individually by absolute or relative values (millimeters for linear actuators).

[Req 6.3.12] It shall be defined a Universal-HOME position which provides repeatability, the most stiffness, stability, safety, and makes the system as compact as possible. This position shall not be changed by any user, only by developers.

[Req 6.3.13] Before being turned off or de-energized the prototype must be moved to its Universal-HOME position. Once it is turned on, and before any other movement, during its power up procedure and initial configuration, the prototype should be in its HOME position. If the system is not already in this position, it must automatically move to it before finishing its start and configuration procedure.

[Req 6.3.14] The user shall be able to define a User-HOME position within the mechanical limits of the prototype. If this position is not defined, it shall be equal to the Universal-HOME position. The User-HOME position shall be lost when de-energizing the prototype.

[Req 6.3.15] There shall exist a universal coordinate system (UCS), which cannot be changed by any user, only the developers of the system. This system shall comply with the following: 1) Its XY plane shall be parallel to the lower platform's plane, 2) Its positive Z axis must point upwards (opposite to the gravity vector), 3) Its origin must be at the center of the prototype (seen from above), either over or near its fixed platform.

[Req 6.3.16] The user shall be able to define one or more coordinate systems (CS) within the mechanical limits of the prototype.

[Req 6.3.17] The user shall be able to activate one coordinate system among the ones that have been previously defined (including the universal). When no coordinate system has been defined, the universal coordinate system shall be used.

[Req 6.3.18] The system must allow relative movements, which are movements that add or subtract to the actual position. These must be preceded by a double plus or minus sign (++/--).

[Req 6.3.19] The system shall allow four types of movement:

1. Absolute: These movements will be applied directly with relation to the current coordinate system.
2. Relative: These movements will be applied with respect to the current position.
3. Combined: They are a combination between absolute and relative movements. This means that some coordinates are indicated as absolute while others as relative.
4. Labeled: A labeled move is any of the previous moves, but its definition has been stored with a label, which is used to refer to the position instead of its coordinates.

[Req 6.3.20] Every absolute movement of the robot shall be performed with respect to the active coordinate system. In order to perform movements with respect to another coordinate system, it shall be activated first.

[Req 6.3.21] Additional to the type of movement of the robot, it shall allow two modes:

1. Linear mode: The movement between two points is executed in two stages: first, the robot moves in a straight line following only translational motion, and when it reaches the translational position, the rotational movements are executed.
2. Actuator mode: In this mode, the translational and rotational movements are performed simultaneously without any trajectory (the final positions for every actuator will be calculated and these will follow the positions at the same time).

[Req 6.3.22] The following basic functions shall be available in Manual mode as well as automatic mode: 1) Label and store the robot's current position, 2) Move the robot to a stored position, 3) Move each of the six actuators independently, 4) Define, edit, or delete coordinate systems, 5) Define, edit or delete the user-HOME.

[Req 6.3.24] The actuators shall be able to move in three ways: 1) Absolute, 2) Relative, 3) Percentage. The absolute form shall receive a scalar parameter (in millimeters) and the actuator will move with respect to its minimal absolute value. The relative form, is the same concept as for moving the robot, this is a parameter that adds or subtracts from the actuator's actual value. The percentage is a value between 0 and 100, and must be followed by a percentage sign "%", (e.g. 30%) which indicates a percent value with respect to the total allowed stroke of the actuator.

[Req 6.3.25] The system shall have protection in order not to exceed the recommended work cycles.

[Req 6.3.26] The system must register all important events with a timestamp. Examples: on-procedure, off-procedure, alarms, mode changes and state changes.

[Req 6.3.27] The prototype should contemplate digital and analog input/output, at least preparation for these in phase I. They are aimed for communication of external applications or other systems with the prototype and they should be configurable by the user.

[Req 6.3.28] The prototype shall contemplate preparation to add communication ports as those in an industrial robot, such as ethernet, CAN, USB, etc. Its type and quantity shall be defined according to an analysis of similar products, ConOps and all requirements.

[Req 6.3.29] The prototype shall incorporate interfaces, or at least preparation to work with concepts such as Industry 4.0, Internet of Things (IoT), Cyber-physical systems, Virtual Reality and Augmented Reality.

[Req 6.3.30] An Android device should be able to function as teach pendant, by way of an android application running on a tablet or cellphone.

[Req 6.3.31] The robot controller shall be as compact and modular as possible.

[Req 6.3.32] The robot software should be based on free software in order to lower costs and have an open architecture. The cases in which this is not possible should be minimal and justified.

This chapter introduced the CIDESI Hexapod project, its development phases, the previous developments related to the system, the ConOps and the system requirements. These two latter concepts are crucial in the development of this thesis project, since the design of the system was based on the fulfillment of the ConOps and the system requirements.

Chapter 4

Controller concept design

In this chapter it is described the concept design of the controller. This involves all the activities and design decisions that led to a viable design. First, it is described the PC and the hardware components of the system. Then, the system functionality from the users' point of view is given. Finally, the design decisions about the selection of the operating system and programming language are presented.

The concept design for the controller is shown in Fig. 4.1. The Controller is mainly divided in a PC, and the Hardware components. They are described next.

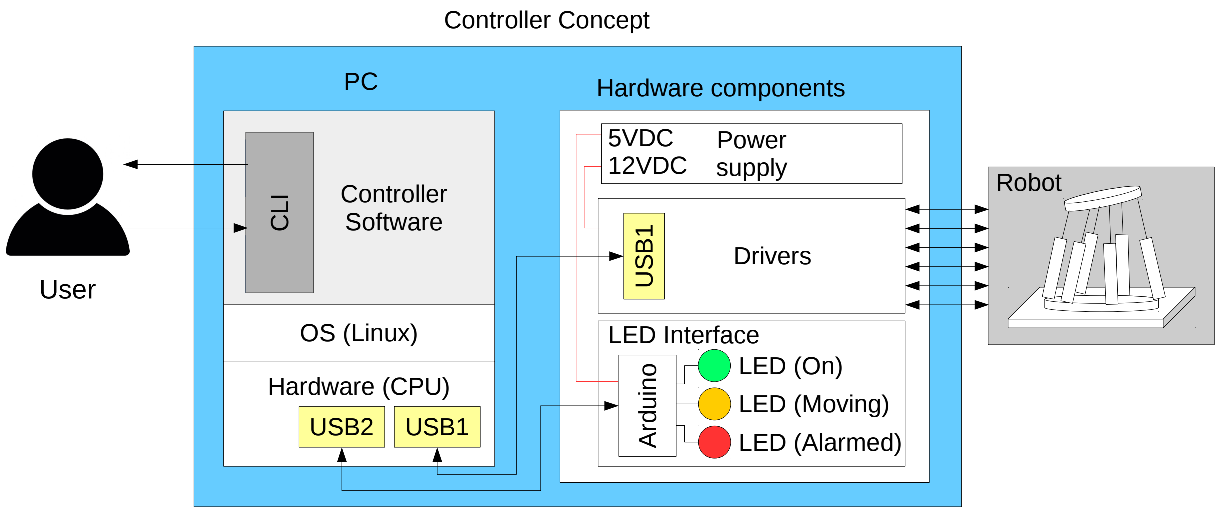


Figure 4.1: Concept design for the Controller.

4.1 PC

The PC is the main platform on which the robot software runs. The hardware components include the peripherals, CPU, and communication ports. On top of the hardware, the operating system runs. The selection of the operating system was an important design decision, and will be described in more detail in the following sections.

For this project, it was used a PC with the following characteristics: Intel i5 processor, 64-bit architecture. The default operating system on the PC is Windows 10. However, it is possible to install other operating systems in additional partitions on the hard drive of the PC.

The controller software is the program responsible for receiving input commands from the user or system, displaying log information, and sending motor commands to the drivers via a serial interface, as well as operating the LED Interface. For communicating with the user, a Command Line Interface (CLI) is provided. In order for the user to interact with the software via the CLI, a keyboard, mouse and monitor are used.

4.2 Hardware components

The "Hardware components" is divided in three parts: the *Power supply*, the *Drivers*, and the *LED Interface*.

4.2.1 Power supply

The power supply should be able to power the LED Interface, and the drivers. The PC is powered separately. In order to provide current to the actuators and drivers, it is required a DC power supply able to deliver at least 18 Amp at 12V. The Arduino only requires a 5V supply, and its current consumption is negligible.

4.2.2 Drivers

For phase I of the project, each linear actuator is activated by a Pololu jrk21v3 driver. The jrk21v3 is a general-purpose motor controller, which supports various interfaces, including USB. It is shown in Fig. 4.2.

This driver works with supply voltage in the range 5V to 28V, and 3 Amp maximum continuous current (5 A peak). It also provides a lot of features, such as driver shutdown on under-voltage, over-current, and other conditions.

The following communication options are provided:

- USB interface for direct connection to a PC.
- Full-duplex, asynchronous serial interface for direct connection to microcontroller or other embedded devices.
- Hobby radio control (RC) pulse width modulated (PWM) interface for direct connection to an RC receiver or RC controller.

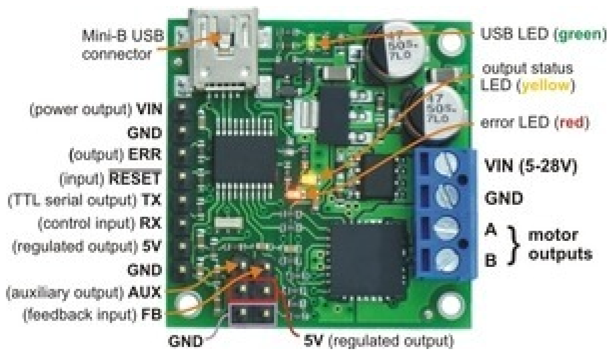


Figure 4.2: The jrk21v3 driver.

More information on the communication interfaces is provided in section 4.3.

4.2.3 LED Interface

This component displays visual information about the robot current state. This is done with three LEDs, powered by an Arduino. The Arduino receives commands from the Controller software via a USB-Serial interface, and turns the LEDs On and Off, accordingly.

4.3 Communication Interface

In order to send commands to and receive information from the driver, the jrk has three serial interfaces: USB Dual Port, USB Chained, and UART. In order to drive the six actuators using only one communication port, the modes USB Chained and UART are required. They are described next.

4.3.0.1 USB Chained connection mode

In this mode, the Command Port is used to both transmit bytes on the TX line and send commands to the jrk. The jrk's responses to those commands will be sent to the Command Port but not the TX line. In this mode, TTL Port is not used. This mode allows a single COM port on a computer to control multiple jrks.

4.3.0.2 UART connection mode

In this mode, the TX and RX lines can be used to send commands to the jrk and receive responses from it. Any byte received on RX will be sent to the Command Port, but bytes sent from the Command Port will be ignored. The TTL Port is not used. This mode allows a jrk to be controlled using a microcontroller or other TTL serial device.

4.3.1 Connection of the drivers

In order to control multiple jrks with a single COM port, the jrks must be connected forming a network of one master device and multiple slave devices:

- Every jrk must be assigned a different device number so that they can be individually addressed. This can be done using the jrk configuration utility.
- The master device is the one connected to the PC via USB cable and is configured in USB Chained mode.
- All slave devices are configured in UART connection mode.
- The TX line of the master device is connected to the RX line of all slave devices.
- When using more than one slave device in the network, an AND gate is used to join all the TX lines of the slave devices. The output of the AND gate is connected to the RX line of the master device.

The connections for the serial communication is shown in Fig. 4.3.

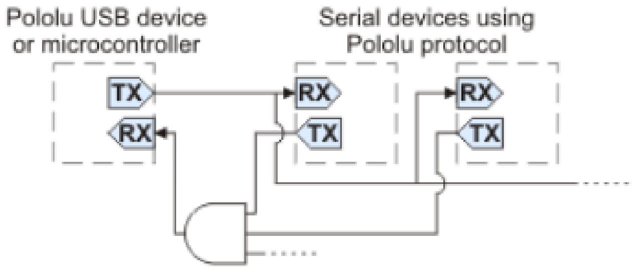


Figure 4.3: Connection of multiple jrks [23].

4.4 System functionality

In the system's requirements analysis a set of desired functions was established. Additionally, the ConOps describes the desired way in which the robot should be operated, the persons involved with the robot and their responsibilities.

All of these elements lead to the development of a general use case diagram of the system, shown in Fig. 4.4. In this diagram, the Hexapod system is divided into three subsystems: the *software subsystem*, the *drivers subsystem*, and the *Mechanical structure*. The mechanical structure includes the *actuators*.

There are two actors in the system, one is the operator, who is responsible for operating the robot through the software. The operator is restricted only to use the robot via the software.The other actor is the technician. Only he is allowed to make modifications to the controller, configure the drivers, assemble or disassemble the robot.

This is the way in which the CIDESI Hexapod system is intended to be operated.

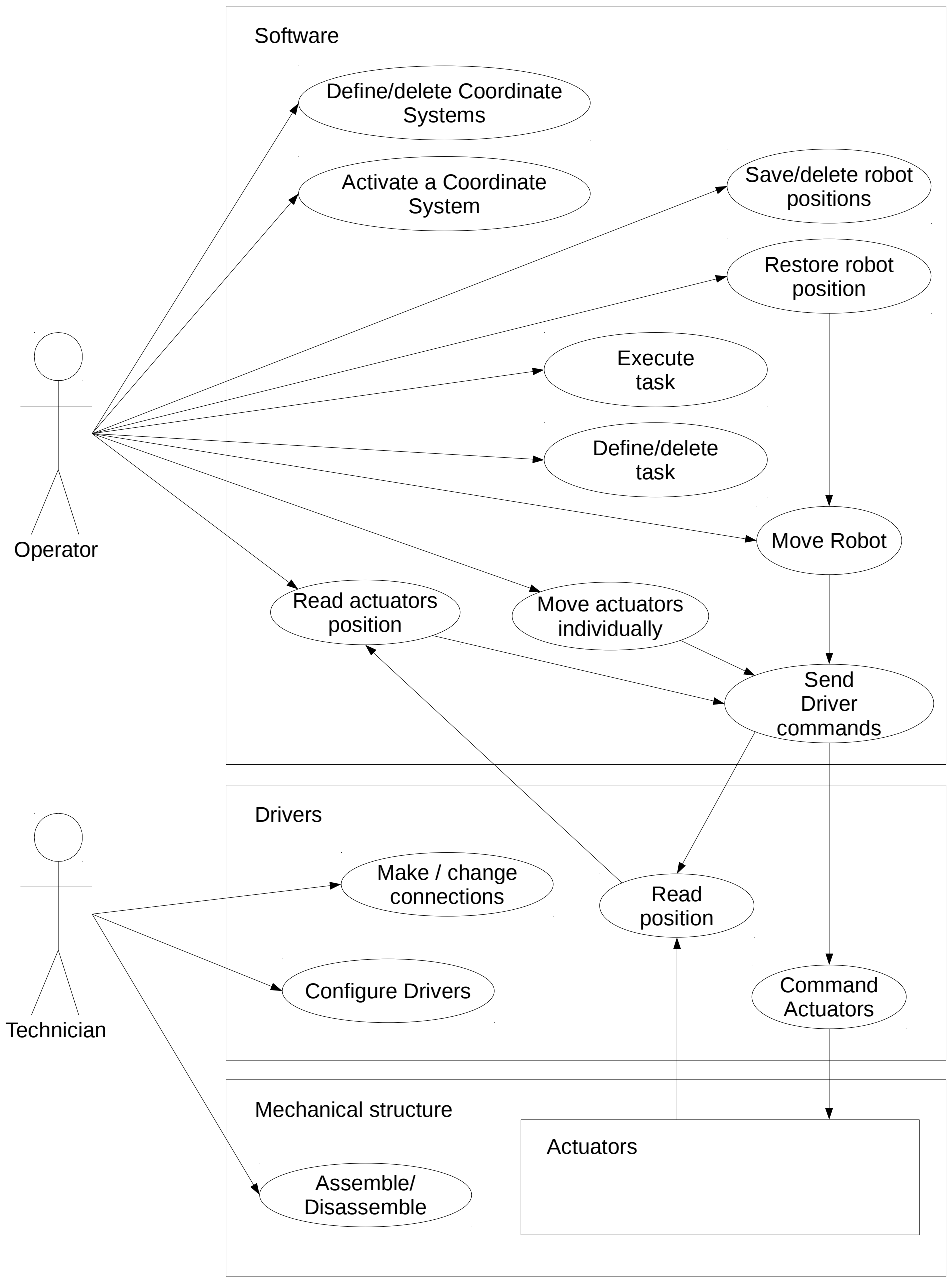


Figure 4.4: Functions to implement in Phase I of the project

4.5 Selection of the operating system

According to requirement **[Req 6.3.32]**, it is desired to use free software for building the project. For this reason, the Ubuntu Linux operating system was selected as the operating system. In addition to being an open source platform, the use of Linux has additional advantages:

- It is possible to apply a patch to the Linux Kernel, in order to implement hard real-time modules, e.g. RTAI.
- It comes with compilers and interpreters for different languages pre-installed.
- Many Robotics middlewares are based on Linux, e.g. ROS.

Therefore, the use of Linux will be useful in future development of the system, if real-time support, or integration with ROS is required.

4.6 Programming language selection

The main criteria for selection of a programming language were:

- Object orientation: The ability to implement classes allows for a modular design, where each module can be tested separately before integration.
- Portability: It is desired to use a language that can be supported by many different hardware platforms.

The C++ programming language was selected because it is an object oriented programming language, many microcontroller and other embedded devices provide a C++ compiler to program them.

4.7 The Command Line Interface

The user interface is based on a command line approach. This interface was selected for the following reasons:

- It provides a generic interface. It can be used by humans and external systems.
- The lower resource usage of a CLI allows it to be embedded in a microcontroller or microcomputer and is more portable than a Graphical User Interface (GUI).
- It is faster to implement a CLI than a GUI.

Chapter 5

Controller detailed design

This chapter presents the detailed design of the CIDESI Hexapod controller. In this stage of development, the concept design was refined and all the subsystems (hardware, software), were designed in detail. In particular, defining the hardware setup, and communication protocols was crucial to design the software. Therefore, the connection diagram and hardware setup is presented first.

Next, the detailed software architecture is presented, which takes into consideration the programming language(s) capabilities, the hardware, communication interface, and the interaction with the operating system, decisions that were made in the concept design.

Finally, the design of the user interface is presented, as well as the set of commands available to the user.

5.1 Hardware setup

The hardware components are connected as shown in Fig. 5.3. It shows the connections of the actuators (1), drivers (2), an AND gate circuit (3), power supply (4), and PC (5). This setup allows communication with the drivers via serial commands, coming from a PC connected to one of the drivers, via a USB connection.

As shown in Fig. 5.3, the driver connected to the PC is configured as a master device, while the other drivers are configured as slave devices. This is achieved by connecting the RX line of all slave devices to the TX line of the master device, so that all slave devices receive messages from the master device. Additionally, when the devices are not transmitting data, their TX line is driven HIGH. For this reason, in order to connect the TX lines of the slave devices to the RX line of the master device, an AND circuit must be used, as shown Fig. 5.1.

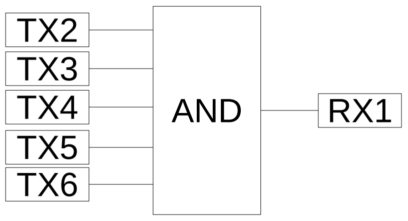


Figure 5.1: Logic circuit for data communication.

The integrated circuit 74LS08 was selected for this purpose. It provides four 2-input AND gates, which allows the implementation of an equivalent logic circuit as shown in Fig. 5.2.

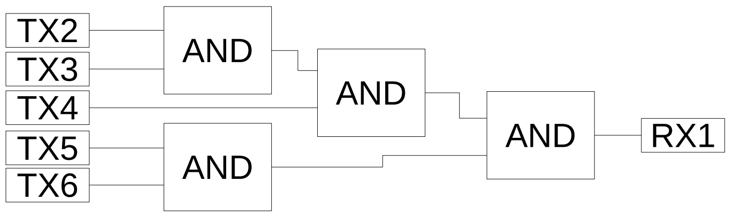


Figure 5.2: Equivalent logic circuit using four 2-input AND gates.

All drivers are connected to a 12 VDC line for providing current to the motors.

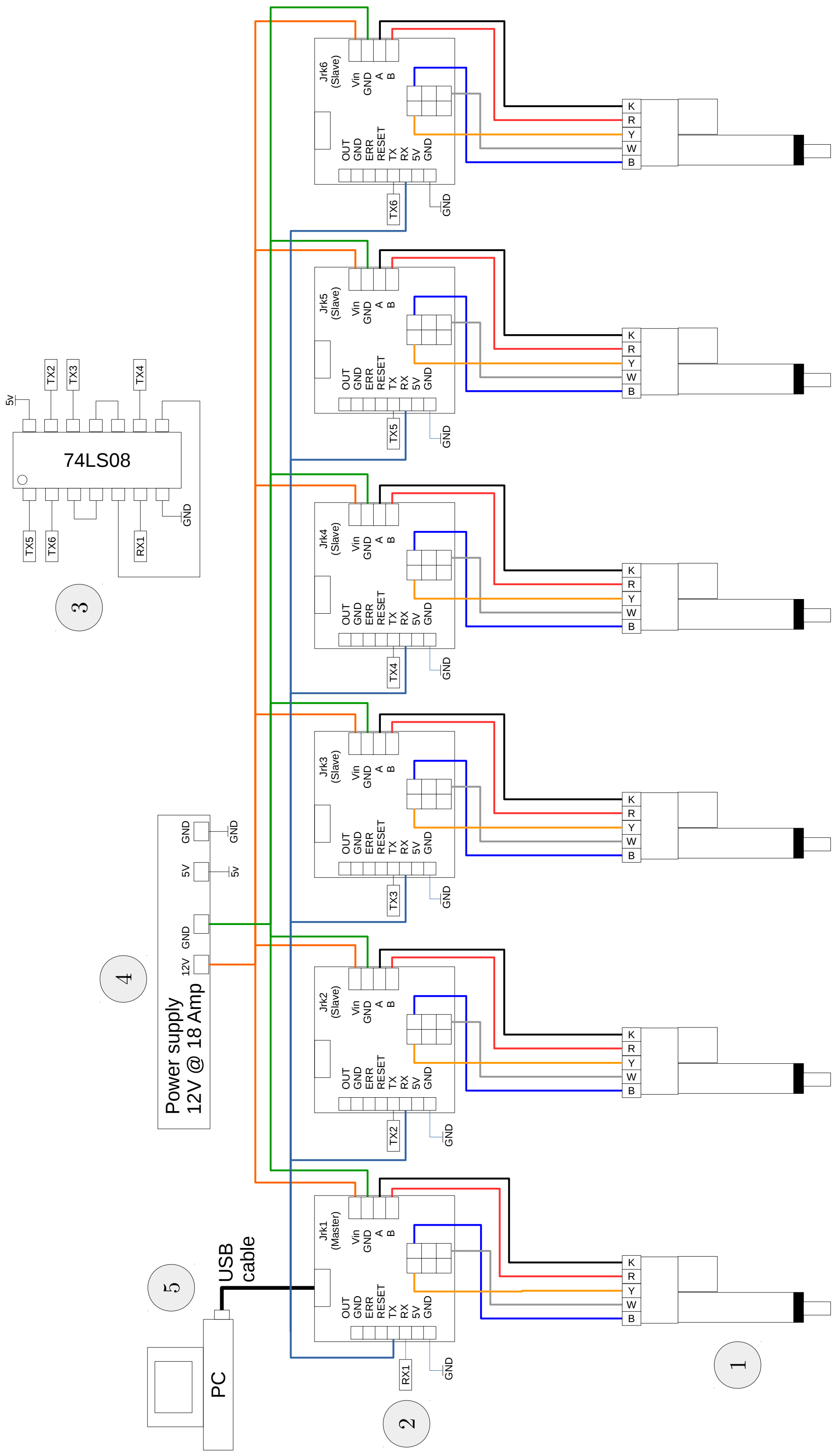


Figure 5.3: Hardware setup connection diagram: (1) Actuators, (2) Drivers, (3) AND circuit, (4) Power supply, (5) PC.

5.2 Software detailed design

This section presents the detailed design of the software for the system.

5.2.1 Software architecture

It is one of the most important elements of the system. In addition to satisfying all requirements established in chapter 3.5, the architecture is designed based on the following criteria:

- Modular design. The software was divided in modules, in order to allow unit testing on each of its modules.
- Reutilization of existing code base and knowledge in order to increase development speed.
- Ability to easily extend/replace software modules.

The software architecture is shown in Fig. 5.4. This figure shows the way in which information is provided to the program in the *User interface*, the way in which information is exchanged and processed in the *Controller Core*, and finally, the way information is sent to the drivers in the *Robot interface*.

The software is divided into several modules, which are described next:

- **Command Control Module**: This module has the task of taking commands from the user as input, process them, and perform the appropriate action, depending on the contents of the command. Depending on the command, one of the many functions are called. Most of the software functionality is provided to the user by way of one of the functions in this module.
- **CS_Manager**: The transformations between different coordinate systems are handled by this module. The user can define different coordinate systems with the function define_CS. These coordinate systems will be stored in a file, and the user can activate one of the previously defined coordinate systems by calling the function activate_CS. When a move command is issued by the user, the provided coordinates will be transformed to the current coordinate system before being sent to the Motion Control Module.
- **Motion Control Module**: This module is in charge of providing high level control for moving the individual actuators. The functions it provides allow to send commands to move the robot, and also to monitor if the actuators have reached their target positions. Only this module has access to the drivers and the serial port.
- **Inverse Kinematics**: This module calculates the inverse kinematics of the robot, with the function poseToAct. It requires the robot's geometric parameters, as well as the actuators' limits.
- **Drivers [1...6]**: The MotionControlUnit contains an array of six instances of the LinearDriver class. The LinearDriver class implements the communication to the drivers, via the serial port, and the six classes are managed by the MotionControlUnit.

- **TaskStorage**: This module will provide methods for creating, searching, and adding command to tasks. The tasks will be stored in a file, called the Task database.
- **PositionStorage**: Similar to TaskStorage, the PositionStorage module will allow the user to work with robot positions, which will be stored in a file.
- **Utility functions**: This module contains some helper functions that can be used by all other modules, such as parsing numbers from strings, or logging information into a log file.
- The **Industrial Interfaces** module is not implemented for this project, but it is proposed for implementing I/O and communication of the controller software with other devices via different communication protocols.

The arrows that connect the modules indicate the different function calls from one module to another, and the information that is transmitted between modules.

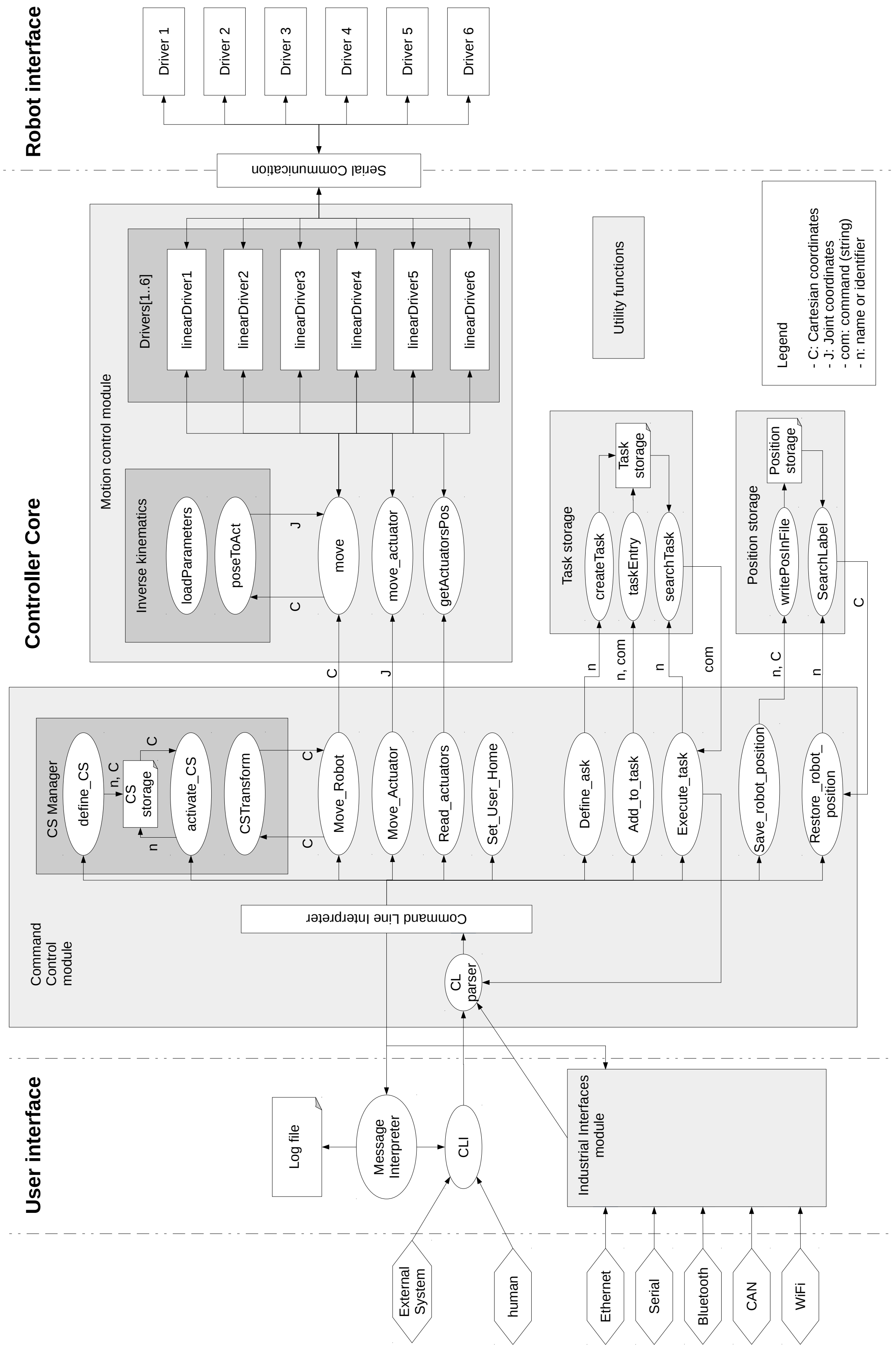


Figure 5.4: Detailed software architecture.

5.3 Use cases

This section provides a detailed description of the use-cases, which were introduced in Fig. 4.4. Each use-case is described with a table, that shows the following information:

- Use Case: The name of the use case that is being described.
- Goal in Context: A brief description of the main goal of the use case.
- Preconditions: A list with activities that must take place, of conditions that must be satisfied, before the use-case can be started.
- Success End Condition: Describes the state of the system after the use-case was executed in the way that it was intended (No errors).
- Failed End Condition: Describes the state of the system after the use-case execution, in case that an error is produced, or an unexpected situation is encountered.

In addition to the use-case table, a sequence diagram showing the flow of events and execution is presented for each use-case. In the sequence diagram, the main components (software, hardware, and human) involved in the use-case are shown.

5.3.1 Description of Use Case “Read actuators position”

Use Case	Read actuators position
Goal in Context	Display the position of all actuators.
Preconditions	The robot and controller must be connected and powered. The communication port must be open.
Success End Condition	The actuators positions are displayed on the monitor.
Failed End Condition	The actuators positions are not displayed on the monitor and a non-success message should be returned.

Table 5.1: Use case “Read actuators positions”.

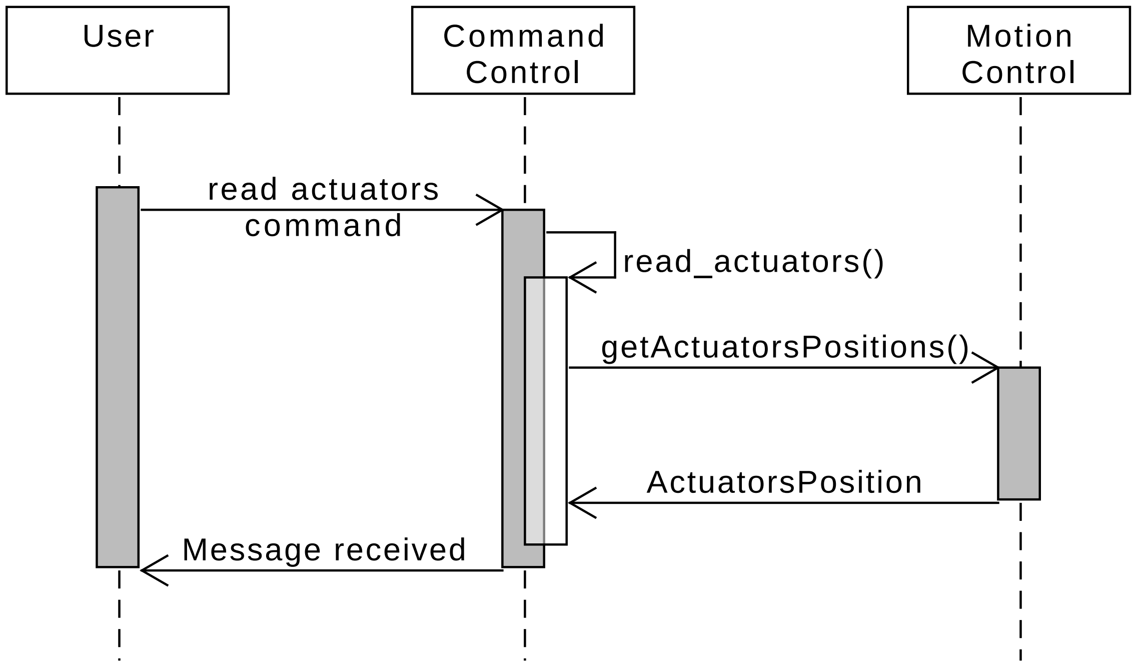


Figure 5.5: Sequence diagram for “Read actuators positions”.

5.3.2 Description of Use Case "Move Robot"

Use Case	Move robot
Goal in Context	Move the robot to a specified position.
Preconditions	The robot and controller must be connected and powered. The robot is not moving.
Success End Condition	The robot reaches the specified position.
Failed End Condition	The robot does not reach the specified position and a non-success message should be returned.

Table 5.2: Use case "Move robot".

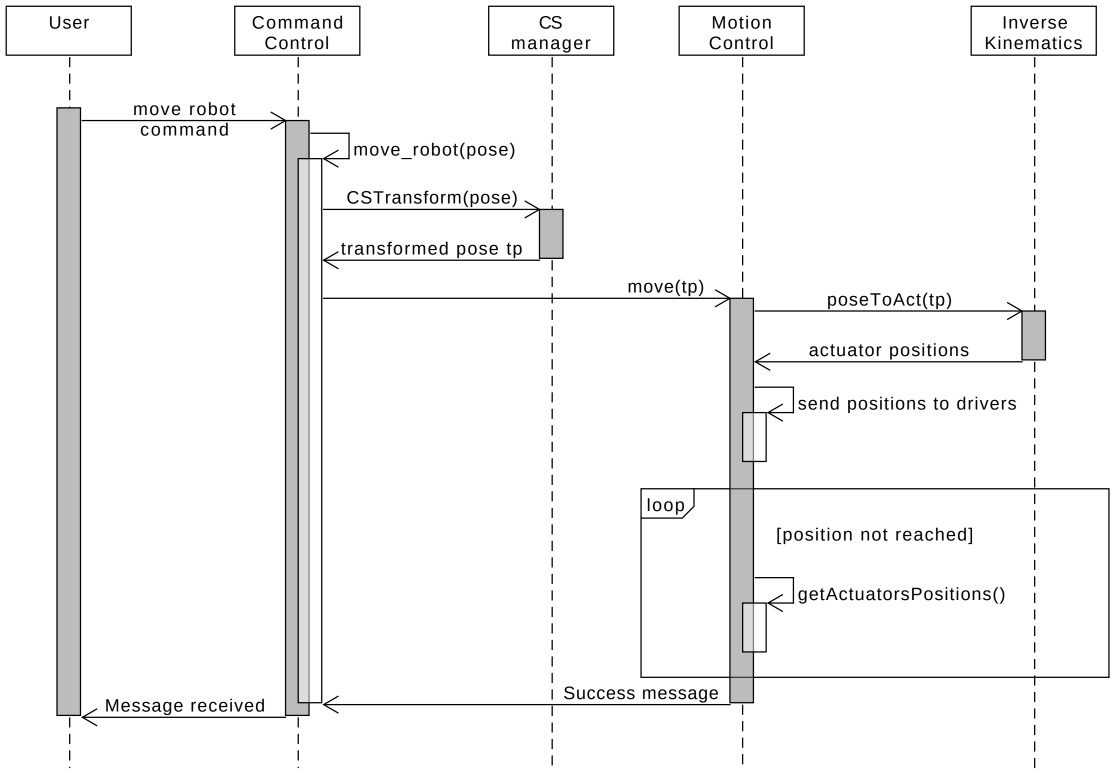


Figure 5.6: Sequence diagram for "Move robot".

5.3.3 Description of Use Case “Move Actuator”

Use Case	Move Actuator
Goal in Context	Move an individual actuator to a specified position.
Preconditions	The robot and controller must be connected and powered. The robot is not moving.
Success End Condition	The actuator reaches the specified position.
Failed End Condition	The actuator does not reach the specified position and a non-success message should be returned.

Table 5.3: Use case “Move actuator”.

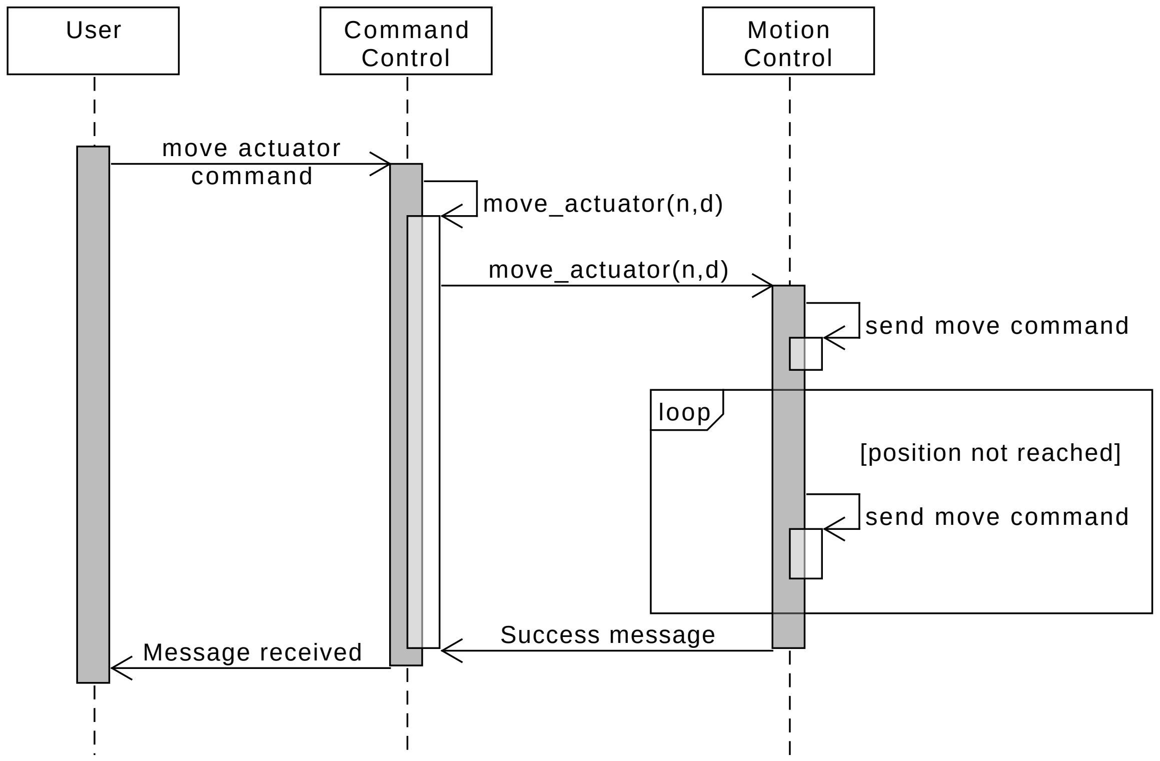


Figure 5.7: Sequence diagram for “Move actuator”.

5.3.4 Description of Use Case “Save robot position”

Use Case	Save robot position
Goal in Context	Save current robot position and store it in memory
Preconditions	The robot and controller must be connected and powered. The robot is not moving.
Success End Condition	The position is stored.
Failed End Condition	The position is not stored and a non-success message should be returned.

Table 5.4: Use case “Save robot position”.

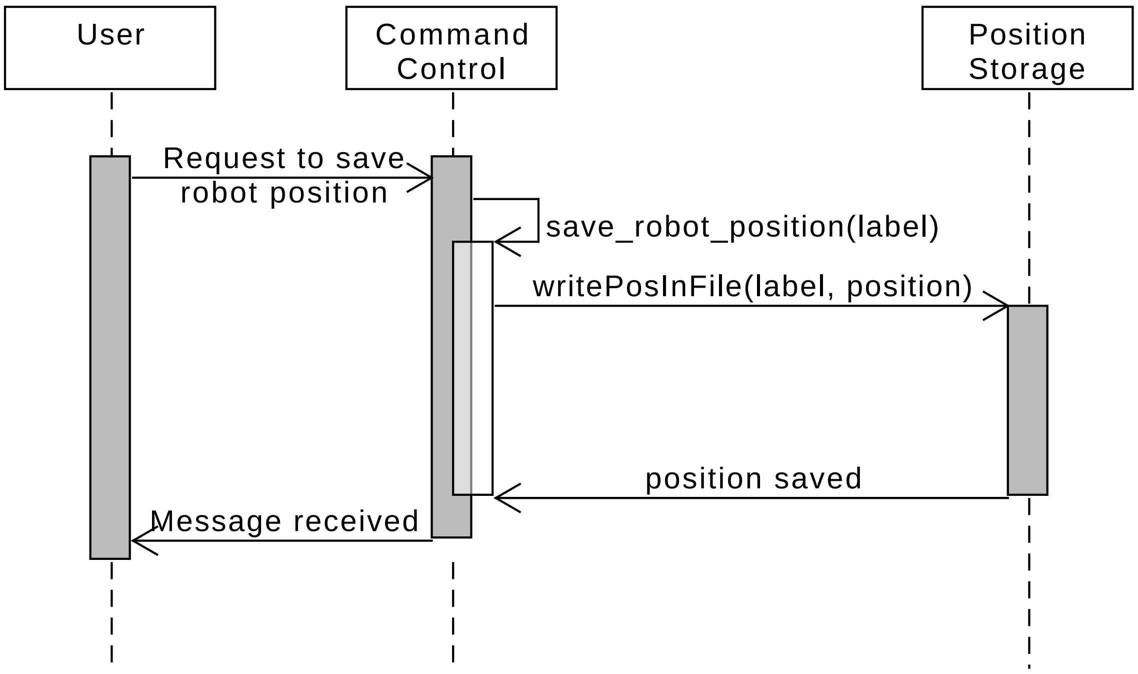

User
Command Control
Position Storage
Request to save robot position
save_robot_position(label)
writePosInFile(label, position)
position saved
Message received

Figure 5.8: Sequence diagram for “Save robot position”.

5.3.5 Description of Use Case “Restore robot position”

Use Case	Restore robot position
Goal in Context	A previous robot position of the robot is restored.
Preconditions	The robot and controller must be connected and powered. The communication port is open.
Success End Condition	The position is restored.
Failed End Condition	The position is not restored and a non-success message should be returned.

Table 5.5: Use case “Restore robot position”.

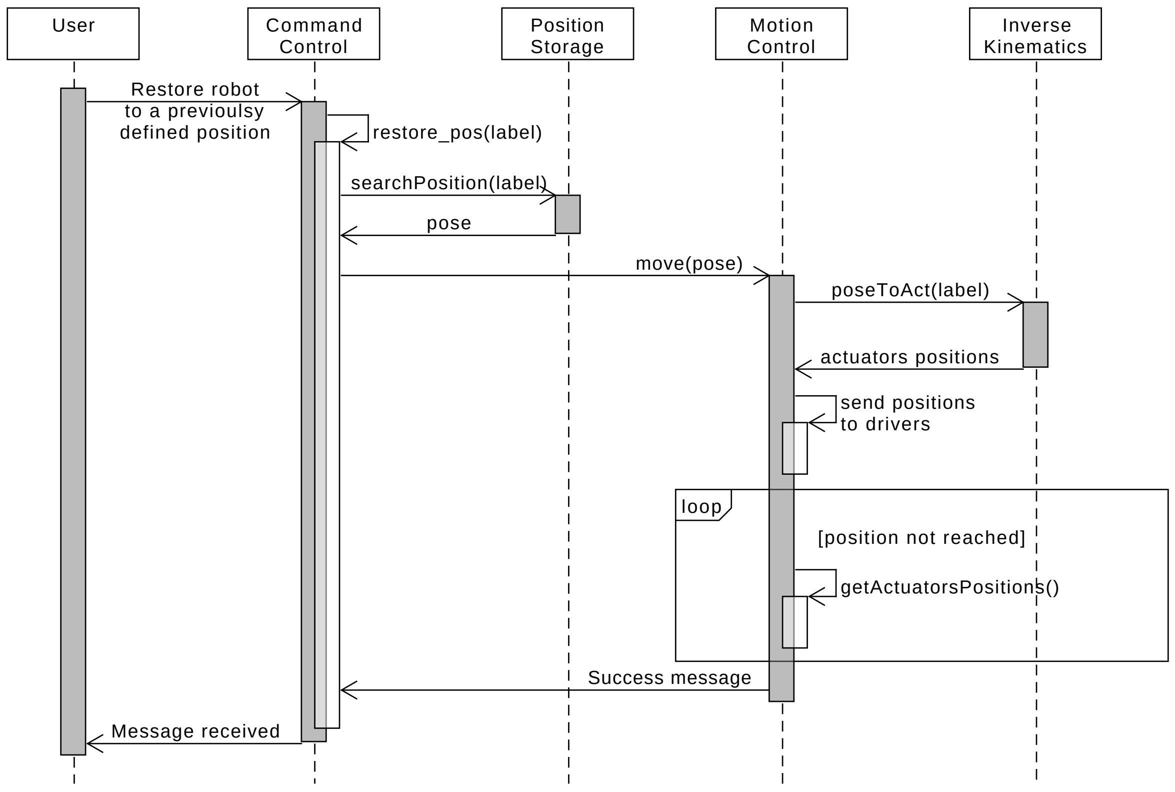


Figure 5.9: Sequence diagram for “Restore robot position”.

5.3.6 Description of Use Case “Define task”

Use Case	Define a new task
Goal in Context	A robot task is defined and stored in memory.
Preconditions	None.
Success End Condition	The task is defined and stored in memory.
Failed End Condition	The task is not defined and a non-success message is returned.

Table 5.6: Use case “Define task”.

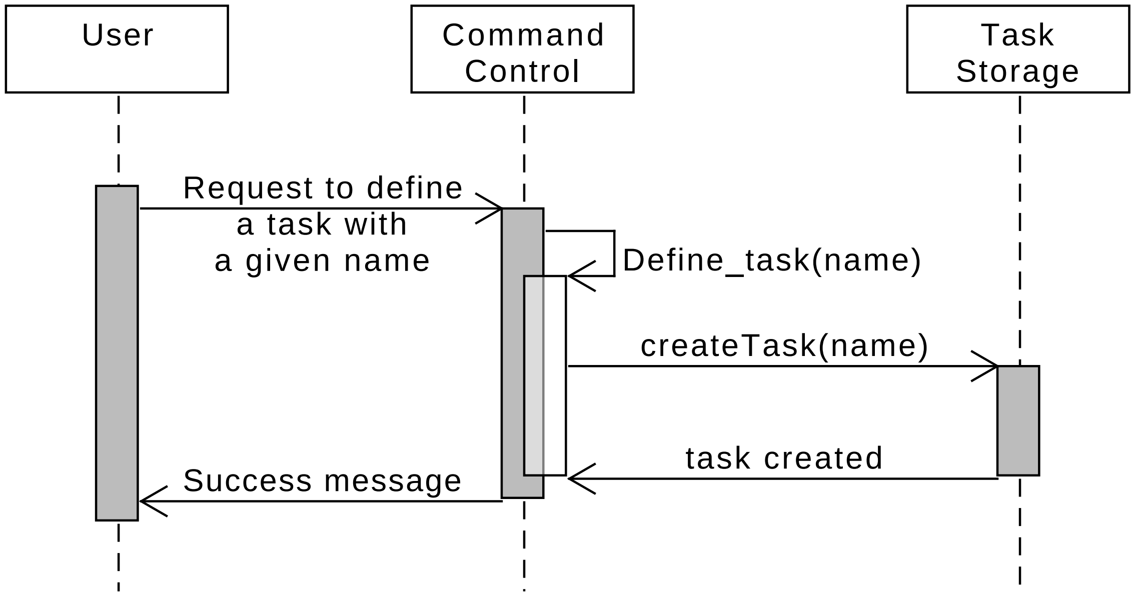


Figure 5.10: Sequence diagram for “Define task”.

5.3.7 Description of Use Case “Execute task”

Use Case	Execute task
Goal in Context	A preprogrammed task is executed.
Preconditions	The robot and driver should be already connected and powered. The communication port is open. The given task is already defined.
Success End Condition	The task is executed completely.
Failed End Condition	The task is not executed completely and a non-success message should be returned.

Table 5.7: Use case “Execute task”.

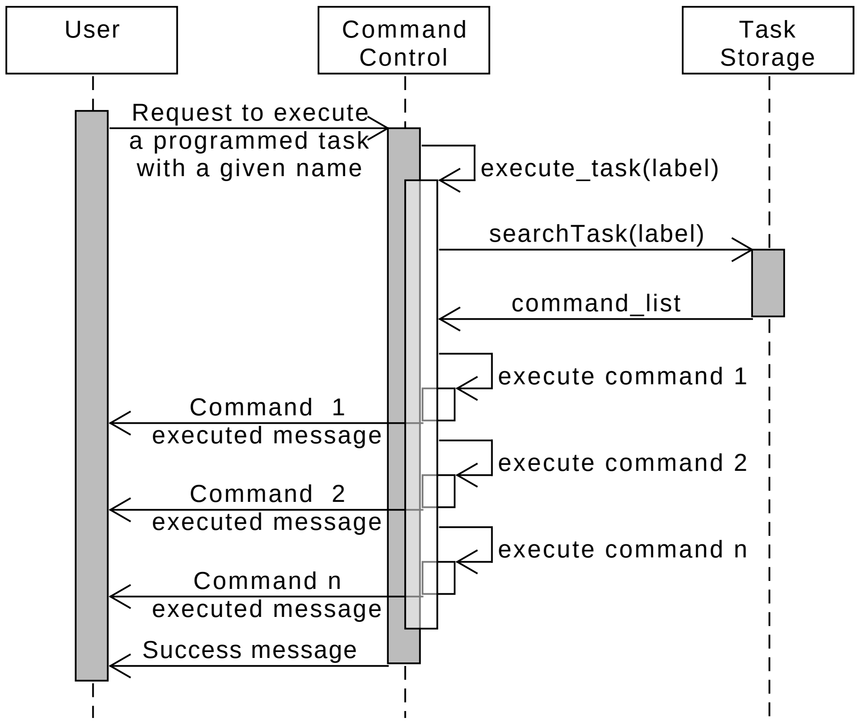


Figure 5.11: Sequence diagram for “Execute task”.

5.3.8 Description of Use Case “Define coordinate system”

Use Case	Define coordinate system
Goal in Context	A coordinate system is defined.
Preconditions	The robot and driver should be already connected and powered. The communication port is open. The robot is not moving.
Success End Condition	The coordinate system is defined and stored in memory.
Failed End Condition	The coordinate system is not defined and a non-success message should be returned.

Table 5.8: Use case “Define coordinate system”.

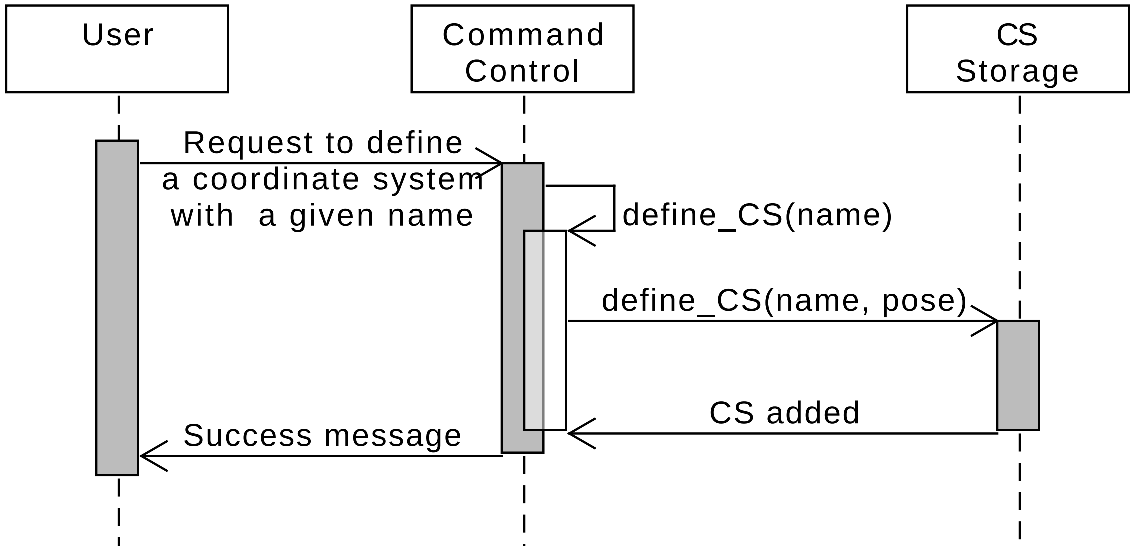


Figure 5.12: Sequence diagram for “Define coordinate system”.

5.3.9 Description of Use Case “Activate coordinate system”

Use Case	Activate coordinate system
Goal in Context	A previously defined coordinate system is activated.
Preconditions	The robot and driver should be already connected and powered. The communication port is open. The robot is not moving. The coordinate system must be previously defined.
Success End Condition	The coordinate system is activated.
Failed End Condition	The coordinate system is not activated and a non-success message should be returned.

Table 5.9: Use case “Activate coordinate system”.

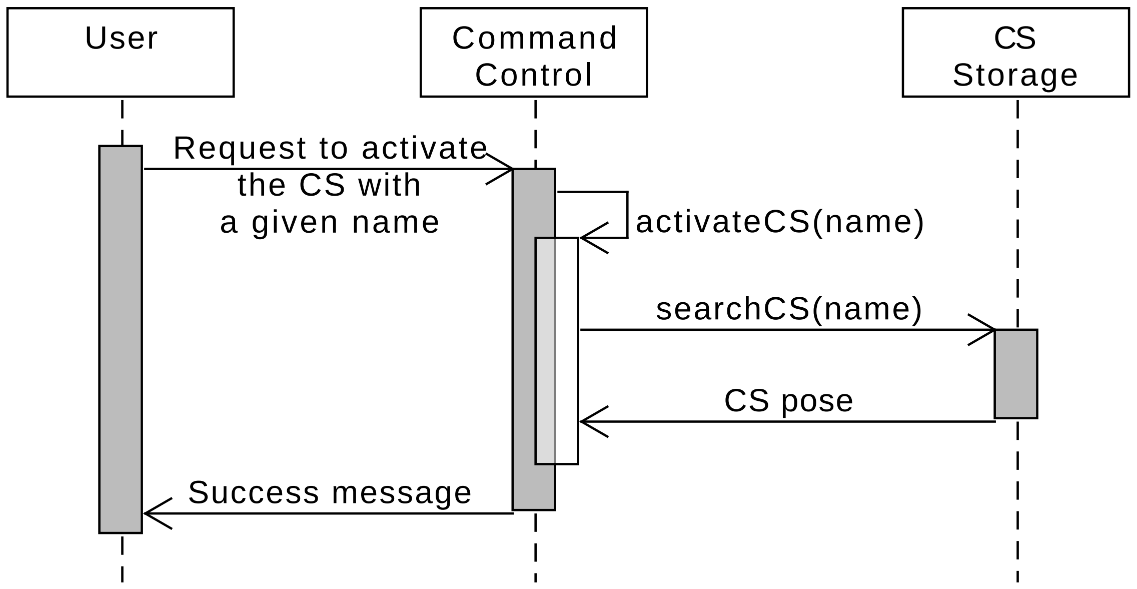


Figure 5.13: Sequence diagram for “Activate coordinate system”.

5.3.10 Description of Use Case “Define user home position”

Use Case	Define user home position
Goal in Context	The current position of the robot is defined as the user home position.
Preconditions	The robot and driver should be already connected and powered. The communication port is open. The robot is not moving.
Success End Condition	The user home position is defined.
Failed End Condition	The user home position is not defined and a non-success message should be returned.

Table 5.10: Use case “Define user home position”.

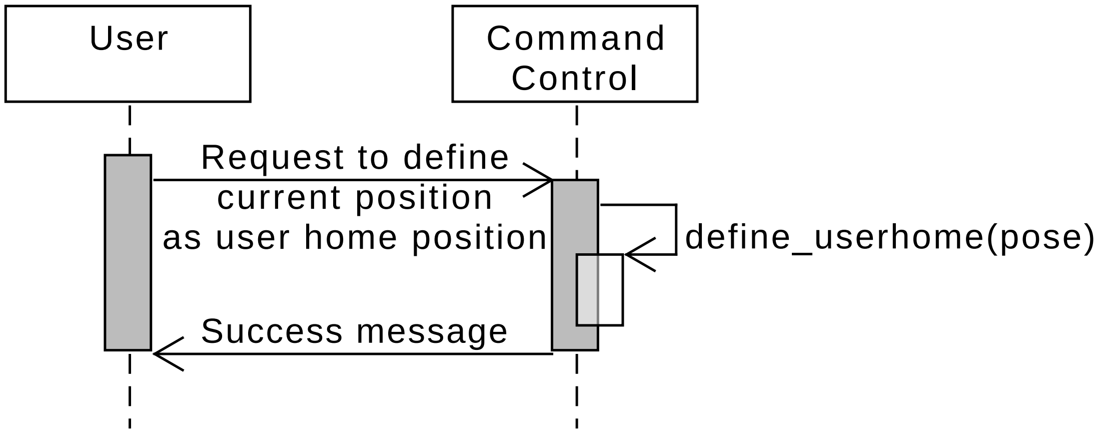


Figure 5.14: Sequence diagram for “Define user home position”.

5.4 User Interface Concept

This section describes the user interface provided for stage 1 of the CIDESI Hexapod robot. It is a CLI and its characteristics are presented next.

5.4.1 Elements of the User Interface

The principal elements in the CLI are the commands and their arguments. With a command, the user indicates the action that must be executed by the controller. The arguments specify how the command should be executed.

5.4.2 Structure of command

A command is a string of text, which contains tokens separated by spaces. A token can be the name of a command, a number, an identifier, an option, etc.

The command structure is as follows:

<command_name> <argument_1> <argument_2> ... <argument_n>

The first token in every command must always be the name of the command. All remaining tokens are arguments of a command. Some commands take no arguments, while other commands, may require up to six different arguments. All the commands available to the user are described in section 5.5.

5.4.3 Design of dialogues

The controller communicates the results of all operations, errors, warnings, failure conditions, etc., through text messages (displayed on the terminal) or log files when the program is running.

5.5 Controller commands

The functions provided to the user are called via commands. The commands are divided into logical groups, and each command is described in detail, explaining the command's function, it's input parameters. Because the command will be entered as a text string by the user or external system, a command parser able to define multiple parameters and flags is considered.

5.5.1 List of user commands

Table 5.11 shows the commands that were implemented in this project, as well as the requirement they fulfill. Some commands are not related to a particular requirement, they were added to the system in order to allow for testing, measurement and calibration of the robot and the actuators.

Table 5.11: List of Controller Commands

Requirement	Command name and description
	Information and help commands
	help - Print help
	info - Print info
6.3.6	**Manual mode commands**
6.3.3, 6.3.4, 6.3.5, 6.3.8, 6.3.9,	**move** - Move robot
6.3.1, 6.3.11, 6.3.24	**move-actuator** - Move individual actuator
	move-csv - Move to positions indicated in CSV file.
	move-lengths - Move actuators to specified lengths.
	Robot positions commands
	read-actuator - Read actuator values
6.3.19, 6.3.22	**save-position** - Save robot position
6.3.19, 6.3.22	**restore-position** - Restore robot position
6.3.22	**delete-pos** - Delete position from database
6.3.19	**list-saved-pos** - List saved robot positions
	Home position commands
6.3.12	**move-uhome** - Move robot to user-defined HOME position
6.3.14	**move-home** - Move robot to universal HOME position
6.3.14	**save-position-home** - Define current position as user-HOME position
6.3.6	**Automatic mode commands**
6.3.6	**task-define** - Define task with a given name
6.3.6	**task-execute** - Execute task
6.3.6	**task-list** - List declared tasks
	Coordinate systems commands
6.3.16	**cs-define** - Define coordinate system

Continued on next page

6.3.17, 6.3.20	**cs-set** - Activate coordinate system
6.3.16	**cs-list** - List coordinate systems
6.3.16	**cs-delete** - Delete coordinate system
	Additional commands
6.3.13	**exit** - Exit (Power off procedure)
	sleep - Wait for specified amount of milliseconds.
	driver-info - Display information of a driver device.

This chapter presented the design stage of the controller. This included hardware, software, and the design of the system operation, by means of use-cases and a set of software commands. After the design phase was completed, the implementation phase is presented in the next chapter.

Chapter 6

Controller implementation

This chapter describes the implementation of the system. First, it is described the wiring and connection of hardware components. Then, the software implementation is presented. It is given a detailed description of the classes that were developed, as well as additional tools for generating the source code.

6.1 Controller electrical connections and wiring

In this section it is described the controller board prototype that was developed, and its related connections with the power supply, and the robot.

6.1.1 Controller board connections

The connection diagram of the actuators, drivers, power supply, and PC was presented in section 4.2. In order to connect and wire all these components, an experimental prototype was developed. It is shown in Fig. 6.1.

A plastic plate was used to fix all components. This material was used because, unlike wood, acrylic, or metal, it is easy and fast to cut and make holes to fix components to it. The larger components were placed and fixed to the plate, making sure that there is enough space for wires and that the USB cable can be plugged to every driver, for configuration of the drivers.

For the connection of the power supply channels, the communication lines of the jrk drivers, and the AND gate circuit, three protoboards were used.

Every actuator requires five wires, which must come from the drivers in the controller board. The controller board and the robot are joined together by two 20-wire cables. In order to connect the actuators with the cable, and the controller to the cable, six 12-point terminals were used: three on the robot side, and three on the controller side. This provides 36 connections, but only 30

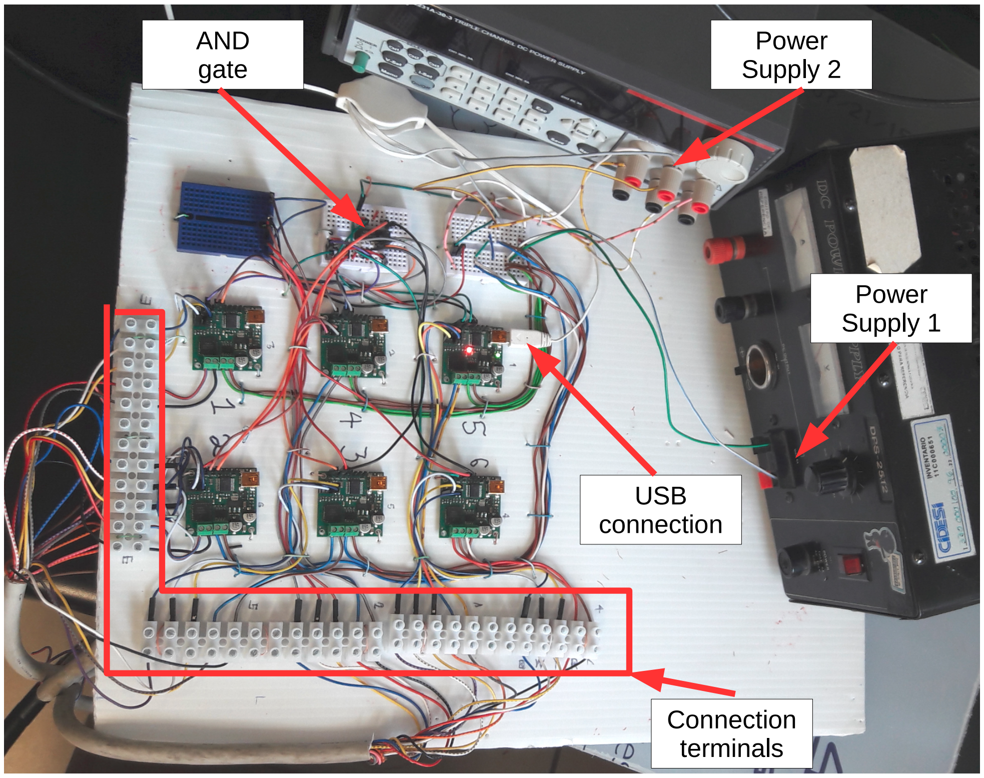


Figure 6.1: Preliminary assembly of critical components of the CIDESI Hexapod controller.

are needed for the six actuators. The remaining connections can be used for other purposes in future development of the system.

6.1.2 Power supply connections

At the time of testing the actuators and testing the robot, a power supply able to deliver the current for all actuators at maximum load was not available. Therefore, two Laboratory DC power supplies were used together to power the actuators. Together, they are able to supply 12 Amp at 12 V, which is enough for the purposes of basic testing of the system without load.

6.1.3 Robot electrical connections

The bottom part of the robot's base frame is designed to contain the electrical connections that will go from the actuators to the controller board. In order to connect the actuators to the 20-wire cables, three 12-point terminals were used, as shown in Fig. 6.2. One cable connects actuators 1,2, and 3, and the other cable connects actuators 4, 5, and 6. The remaining cables were not cut, because they may be used in future development of the system.

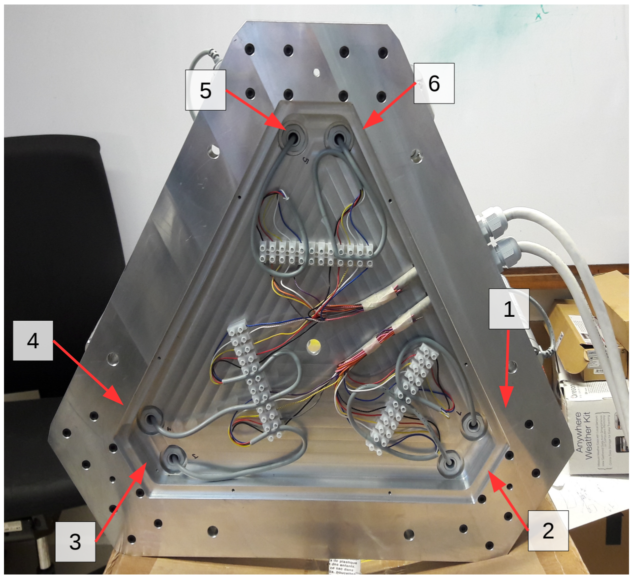


Figure 6.2: Robot electrical connections

6.2 Software implementation

In this section it is presented the implementation of the software modules for this thesis project. These modules were implemented as C++ classes, whose internal parameters can be read from a configuration file and initialized inside a class constructor. Communication between classes is achieved by allowing class methods to call other methods from other classes.

Additionally, there are other software modules which only contain helper functions, such as those needed for parsing strings, or logging events and errors in a log file. The functions provided by these modules are available to all classes. The software modules are:

- CommandControlModule
- CS_Manager
- TaskStorage
- PositionStorage
- InverseKinematics
- MotionControlUnit
- Parameter
- Driver family:
  - DeviceDriver
  - AxisDriver

  - LinearDriver
- arduinoDriver
- rotatorDriver (additional module not used in this work)

The next section presents a class diagram of each module, as well as a description of the most important class members. Methods such as getters/setters or debugging information methods are not shown.

6.2.1 Software module “CommandControlModule”

This module implements all the commands available to the user. As its name suggests, its function is to receive commands from the user, process them, and perform the appropriate functions.

This module provides the function CLInterpreter, which receives commands in the form of text strings. The command is analyzed, and depending on its contents, one of the different available functions is called. Most of the functions return a message code, which informs the Command Line Interpreter if the command was executed successfully, or if there was a problem or error. This message code is then taken by the Message Interpreter, which writes detailed information about the program execution to a Log File, and displays the information back to the user’s terminal.

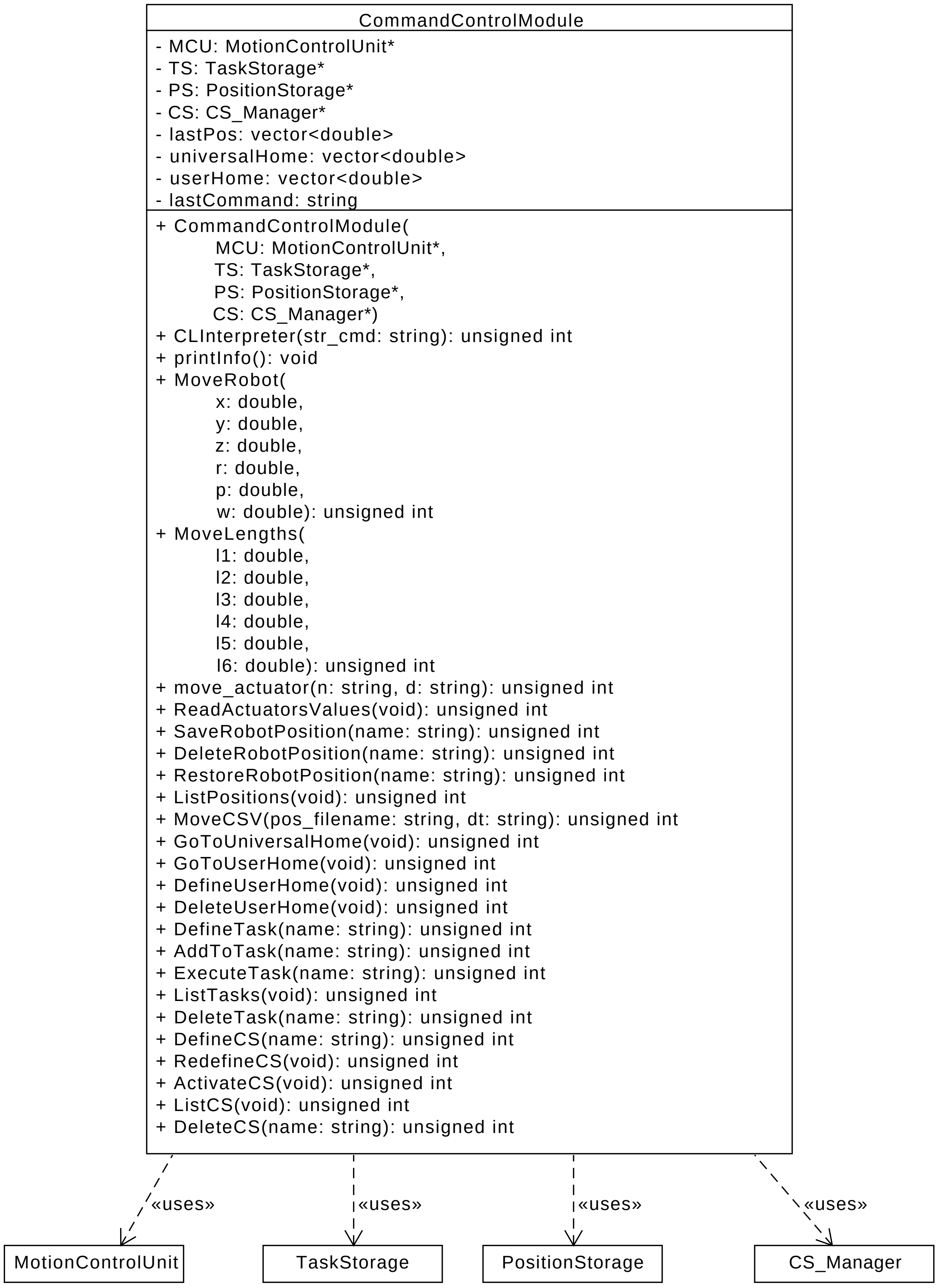

CommandControlModule
- MCU: MotionControlUnit*
- TS: TaskStorage*
- PS: PositionStorage*
- CS: CS_Manager*
- lastPos: vector<double>
- universalHome: vector<double>
- userHome: vector<double>
- lastCommand: string
+ CommandControlModule(MCU: MotionControlUnit*, TS: TaskStorage*, PS: PositionStorage*, CS: CS_Manager*)
+ CLInterpreter(str_cmd: string): unsigned int
+ printInfo(): void
+ MoveRobot(x: double, y: double, z: double, r: double, p: double, w: double): unsigned int
+ MoveLengths(l1: double, l2: double, l3: double, l4: double, l5: double, l6: double): unsigned int
+ move_actuator(n: string, d: string): unsigned int
+ ReadActuatorsValues(void): unsigned int
+ SaveRobotPosition(name: string): unsigned int
+ DeleteRobotPosition(name: string): unsigned int
+ RestoreRobotPosition(name: string): unsigned int
+ ListPositions(void): unsigned int
+ MoveCSV(pos_filename: string, dt: string): unsigned int
+ GoToUniversalHome(void): unsigned int
+ GoToUserHome(void): unsigned int
+ DefineUserHome(void): unsigned int
+ DeleteUserHome(void): unsigned int
+ DefineTask(name: string): unsigned int
+ AddToTask(name: string): unsigned int
+ ExecuteTask(name: string): unsigned int
+ ListTasks(void): unsigned int
+ DeleteTask(name: string): unsigned int
+ DefineCS(name: string): unsigned int
+ RedefineCS(void): unsigned int
+ ActivateCS(void): unsigned int
+ ListCS(void): unsigned int
+ DeleteCS(name: string): unsigned int
«uses»
«uses»
«uses»
«uses»
MotionControlUnit
TaskStorage
PositionStorage
CS_Manager

Figure 6.3: Class diagram for module CommandControlModule

6.2.2 Software module “CS_Manager”

The Coordinate System Manager transforms the coordinates given by the user in the active coordinate system to absolute or world coordinates. Internally, this module stores a coordinate system, called the *active coordinate system*. The private field active_cs_name contains the name of the active coordinate system, and the coordinates of the coordinate system are stored as a vector of real numbers. This module provides the three important functions:

- define_CS: The user calls this command providing a name, and the current pose of the robot is stored as a coordinate system with the given name as identifier in the Coordinate System Database file.
- activate_CS: The user calls this command providing the name of a previously defined coordinate system. The coordinate system is searched in the Coordinate System Database, and if it is found, it is set as the active coordinate system.
- CSTransform: This function takes a pose object in the active coordinate system, and transforms it to a pose in absolute coordinates.

The coordinate systems are stored in a file, whose contents are read when the class is constructed, with the method read_CS_file. This method stores the contents of the file in the private member w, as a 2D vector of strings. Other methods for getting the active CS and deleting CS from the file are also provided.

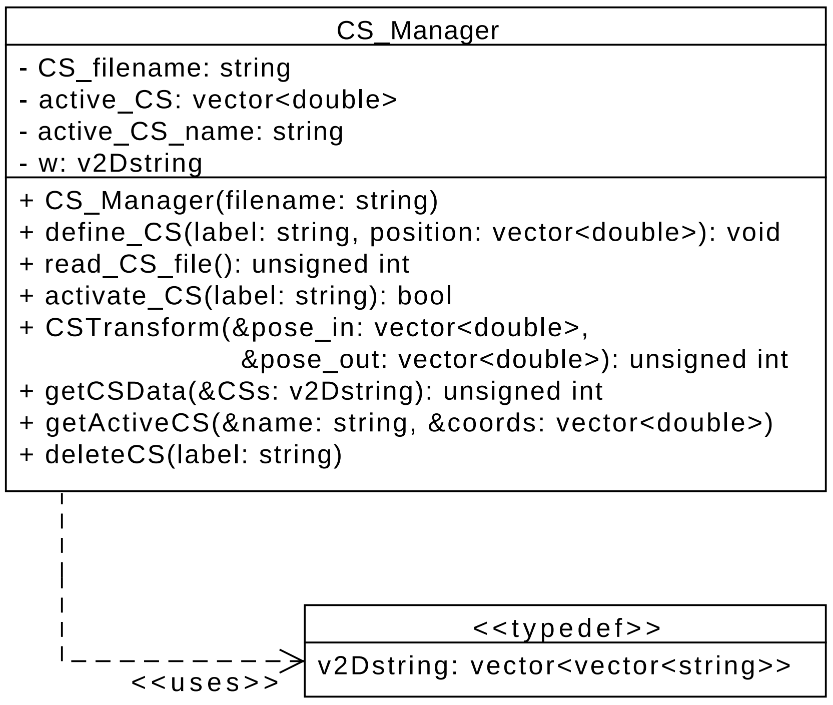

CS_Manager
- CS_filename: string
- active_CS: vector<double>
- active_CS_name: string
- w: v2Dstring
+ CS_Manager(filename: string)
+ define_CS(label: string, position: vector<double>): void
+ read_CS_file(): unsigned int
+ activate_CS(label: string): bool
+ CSTransform(&pose_in: vector<double>, &pose_out: vector<double>): unsigned int
+ getCSData(&CSs: v2Dstring): unsigned int
+ getActiveCS(&name: string, &coords: vector<double>)
+ deleteCS(label: string)
<<uses>>
<<typedef>>
v2Dstring: vector<vector<string>>

Figure 6.4: Class diagram for module CS_Manager.

6.2.3 Software module "TaskStorage"

This class provides the facilities for defining tasks, executing tasks, and adding commands to tasks. This was accomplished by using the class TaskStorage and providing three basic functions to the user:

- CreateTask: This function takes a name as argument. A new task is saved in the Task Database with the given name as the name of the task.
- TaskEntry: After executing a command, such as moving the robot or activating a coordinate system, the user may call this function, providing the name of a previously defined task. This will add the last executed command as an entry to the body of the given task.
- SearchTask: The user provides the name of a previously defined task, and the function returns all the commands that have been stored for this task.

The tasks are stored in a file, whose contents are read and loaded into the private member fileContents when the constructor of the class is called. There are methods for creating, deleting, and searching tasks. The method taskEntry allows to associate string commands under a given task.

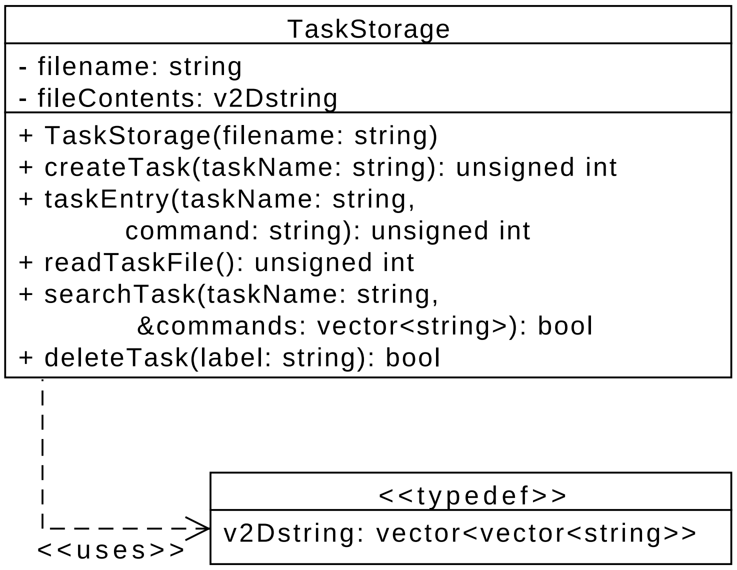

TaskStorage
- filename: string
- fileContents: v2Dstring
+ TaskStorage(filename: string)
+ createTask(taskName: string): unsigned int
+ taskEntry(taskName: string, command: string): unsigned int
+ readTaskFile(): unsigned int
+ searchTask(taskName: string, &commands: vector<string>): bool
+ deleteTask(label: string): bool
<<typedef>>
v2Dstring: vector<vector<string>>
<<uses>>

Figure 6.5: Class diagram for module TaskStorage.

6.2.4 Software module “PositionStorage”

This module provides functions for storing and searching robot positions.

Similar to the TaskStorage class, its contents are loaded from a file and stored in the private variable w.

The PositionStorage class supports addition, search and deletion of positions from the database, with the public methods writePosInFile, searchLabel, and deletePosition, respectively.

Before a position is added to the file, it is first searched, and if it is found, an error is reported. If the position is not found, only then it is added to the file. In order to delete a position, it is searched by iterating the w vector, not the file. If it is found in the w vector, it is removed from there, and then the new vector without this position is stored in the file.

The method readCSVPos was not considered in the original software architecture nor in the requirements, but it was added for testing purposes. This method allows reading positions from a file as comma-separated-values (csv).

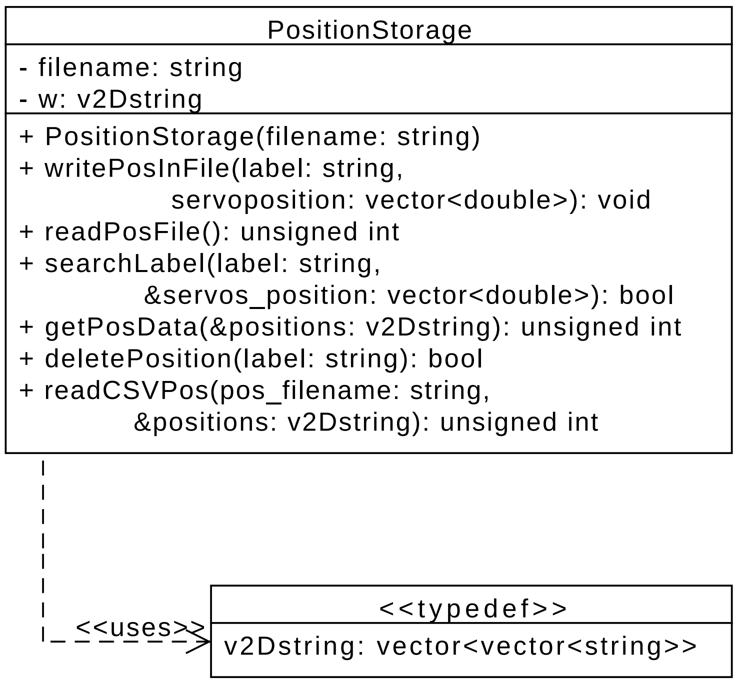

PositionStorage
- filename: string
- w: v2Dstring
+ PositionStorage(filename: string)
+ writePosInFile(label: string, servoposition: vector<double>): void
+ readPosFile(): unsigned int
+ searchLabel(label: string, &servos_position: vector<double>): bool
+ getPosData(&positions: v2Dstring): unsigned int
+ deletePosition(label: string): bool
+ readCSVPos(pos_filename: string, &positions: v2Dstring): unsigned int
<<uses>>
<<typedef>>
v2Dstring: vector<vector<string>>

Figure 6.6: Class diagram for module PositionStorage.

6.2.5 Software module “InverseKinematics”

This module applies the inverse kinematic algorithm in order to calculate the required actuator lengths for a desired robot pose. To do this, the module requires the geometry of the robot and other data. This information can be loaded to the class with the public method loadParameters, or with the provided constructor.

The function poseToAct converts a pose of the robot (in cartesian coordinates) to the required actuator lengths. The implementation of this function is described in more detail in section 6.2.11.

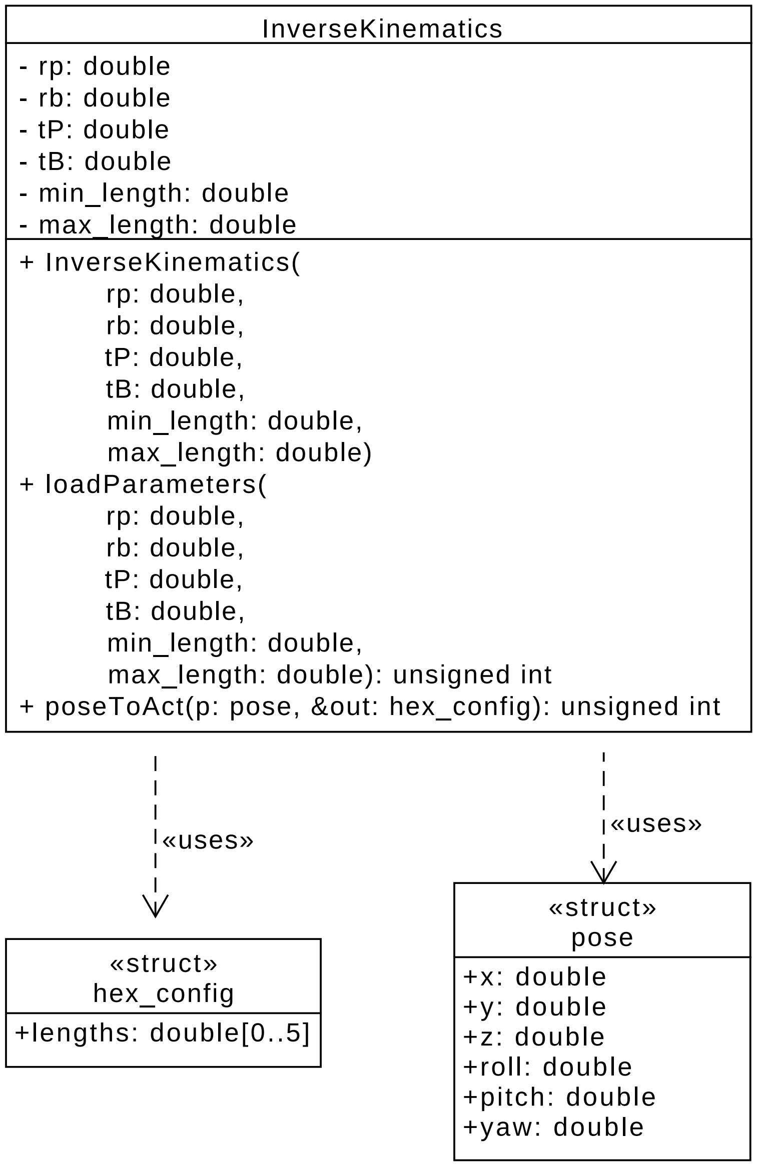

InverseKinematics
- rp: double
- rb: double
- tP: double
- tB: double
- min_length: double
- max_length: double
+ InverseKinematics(rp: double, rb: double, tP: double, tB: double, min_length: double, max_length: double)
+ loadParameters(rp: double, rb: double, tP: double, tB: double, min_length: double, max_length: double): unsigned int
+ poseToAct(p: pose, &out: hex_config): unsigned int
«uses»
«uses»
«struct» hex_config
+lengths: double[0..5]
«struct» pose
+x: double
+y: double
+z: double
+roll: double
+pitch: double
+yaw: double

Figure 6.7: Class diagram for module InverseKinematics

6.2.6 Software module “MotionControlUnit”

This module provides an interface between high level commands, such as move, and the required low level implementation of the drivers. This is done by creating one instance of the *linearDriver* class for each actuator, and storing a reference to each instance in an array, so that the drivers can be accessed with a loop. This class calls the poseToAct method from the inverse kinematics module in order to convert the pose to actuator lengths. For this reason, a reference to the InverseKinematics class must be defined.

The interface consists of six public methods:

- getActuatorsPositions: This method reads the position of the six drivers, and then returns this information.
- move_check: This function receives a pose object in cartesian coordinates, and calls the poseToAct method to calculate the required actuator lengths. After obtaining the lengths, this method will make the appropriate low level driver calls to ensure that the robot has moved to the desired position, before returning.
- move_noCheck: This function works similar to move_check. The only difference is that after sending the drivers their target positions, the function returns immediately and does not check if the actuators reached their goal positions.
- move_actuator: This function takes as arguments the name of an actuator (1 to 6) and a distance in millimeters. Then the robot moves the actuator to the specified position, and waits until the actuators reached their destination.
- displayDriverErrors: This function reads the error flags on the drivers, in order to provide the user with diagnostic information in the case of errors.
- move_l: Moves the robot providing the actuators lengths instead of cartesian coordinates. This function was added to the system for testing purposes.

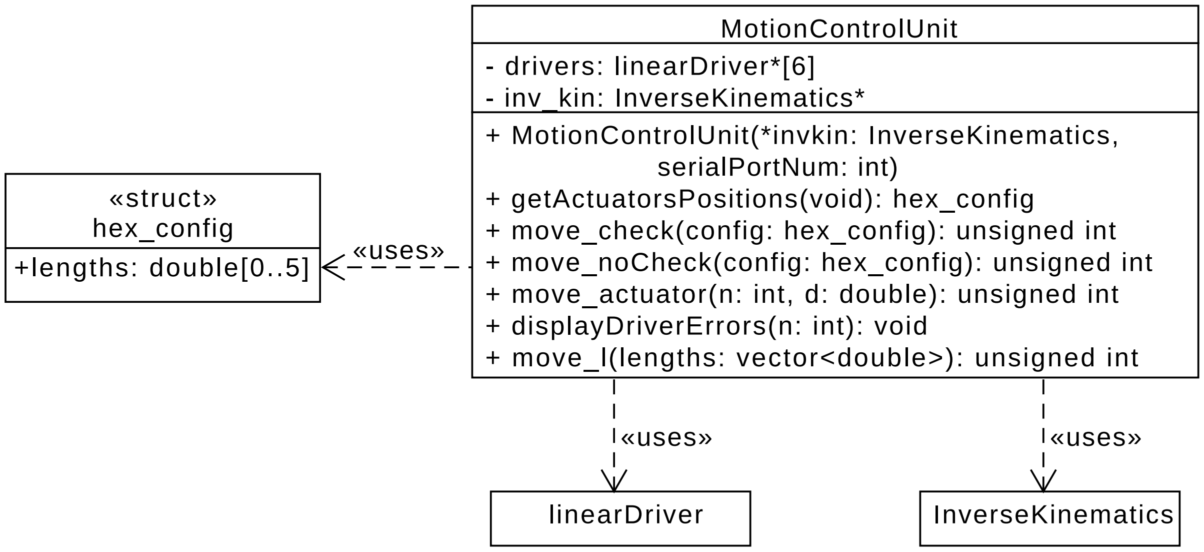

MotionControlUnit
- drivers: linearDriver*[6]
- inv_kin: InverseKinematics*
+ MotionControlUnit(*invkin: InverseKinematics, serialPortNum: int)
+ getActuatorsPositions(void): hex_config
+ move_check(config: hex_config): unsigned int
+ move_noCheck(config: hex_config): unsigned int
+ move_actuator(n: int, d: double): unsigned int
+ displayDriverErrors(n: int): void
+ move_l(lengths: vector<double>): unsigned int
«struct»
hex_config
+lengths: double[0..5]
«uses»
«uses»
«uses»
linearDriver
InverseKinematics

Figure 6.8: Class diagram for module MotionControlUnit.

6.2.7 Software module “Parameter”

This module allows parsing numerical parameter values from a file. The way it works is by calling its public method readList, which accepts the name of a file containing parameters and their respective values. When called, readList creates and initializes one instance of the parameter class for each parameter in the given file. Each instance of this class contains the private members name and value. These are the name of the parameter, and its associated numerical value. These instances are then pushed into the static vector list.

A static private iterator allows the methods of the class to iterate over the class instances (where each instance is one parameter). In order to get the value of a particular parameter, the public method getValOf is provided. The method show_params displays all the parameters that have been pushed into the list vector.

For this class, the only constructor of the class is declared private. This means that this class cannot be instantiated from outside of itself. Only its static method readList can create instances of this class.

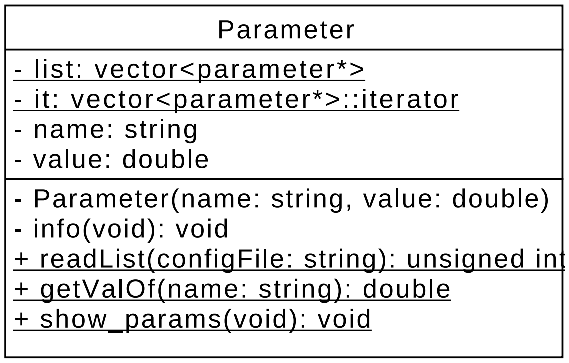

Parameter
- list: vector<parameter*>
- it: vector<parameter*>::iterator
- name: string
- value: double
- Parameter(name: string, value: double)
- info(void): void
+ readList(configFile: string): unsigned in
+ getValOf(name: string): double
+ show_params(void): void

Figure 6.9: Class diagram for module Parameter.

6.2.8 Software module "LinearDriver" and its superclasses

The current phase in the development of the controller software only requires one serial connection to the jrk drivers. However, the complexity of the system is expected to increase as the project develops, with the addition of multiple devices, and communication interfaces. In order to manage the code for different devices, different classes were developed in a family of classes that consists of three classes: DeviceDriver, AxisDriver, and LinearDriver. This family is shown in Fig. 6.10.

The class that implements communication with the jrk drivers is the class LinearDriver. It is a derived class from AxisDriver, which at the same time is a derived class from DeviceDriver.

This family of classes has been designed as part of the CIDESI Hexapod project by the research and development group at CIDESI. The development of this family of classes is intended for future development of the system as well as integration with other devices.

6.2.8.1 DeviceDriver class

The base class, DeviceDriver, has the purpose to manage hardware devices. It provides methods which are general to any type of device, such as installation, uninstallation, isAvailable, etc. It also provides protected members in order to restrict access to them only to itself and its derived classes. Since the implementation of these general methods is still specific to each device, they are declared virtual, which means that the class DeviceDriver expects to find implementations of these methods in its derived classes.

6.2.8.2 AxisDriver class

This class is responsible for managing specifically axis type devices, e.g. linear actuators, motors, etc. Therefore, it provides methods that are specific to axis devices, such as those for obtaining position readings, and conversions from digital readings to actual position interpretation of those readings. Similar to DeviceDriver, the actual implementations of these methods are specific to the axis device, and for these reason, these methods are declared virtual, and their implementations are delegated its derived classes.

6.2.8.3 LinearDriver class

This class contains the implementations of all methods that are specific to the jrk21v3 drivers. It inherits all fields and methods from its parent classes, and is responsible for implementing virtual methods that were declared in its super classes.

One important aspect of this class is that, while six instances of this class are required for the six actuators, only one serial port is used between them. Moreover, installation and uninstallation procedures require the port to be open if at least one actuator is installed, and closed when no more actuators are installed. This means that some fields and methods are not specific to one instance of the LinearDriver class, but rather are common to all instances. These class members are declared static, which will allow to have only one copy of them, shared among all instances of the class.

Some public methods are provided by this class, such as servo2axisLowLevel, moveServo, etc. However, for the actual use of the drivers, it is more convenient to call the methods of its super classes AxisDriver and DeviceDriver: move, getPos, isMoving, On, Off, installation, etc.

The private methods jrkGetVariable, jrkGetInput, etc., are low level functions that write to and read from the serial port.

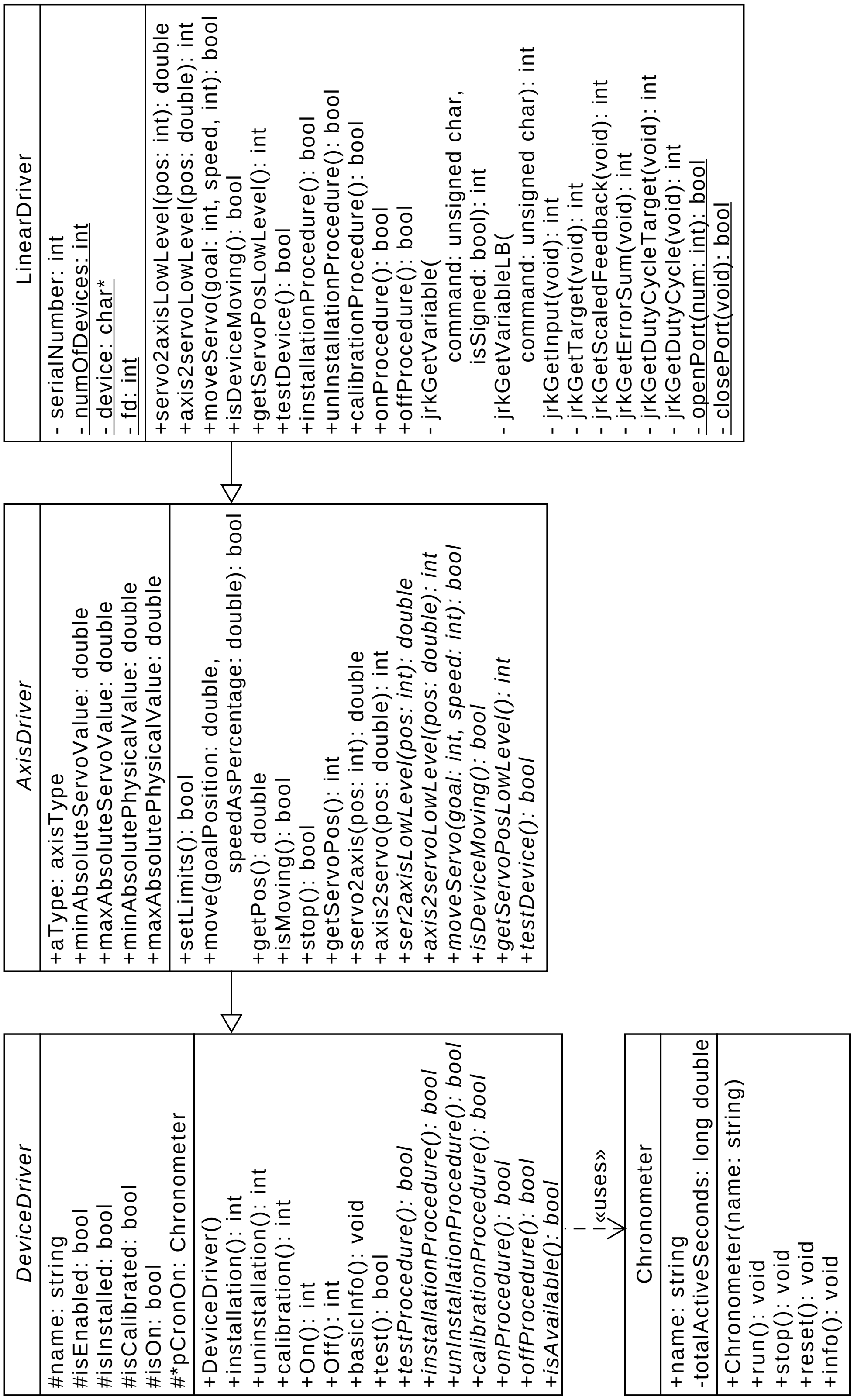

LinearDriver
- serialNumber: int
- numOfDevices: int
- device: char*
- fd: int
+servo2axisLowLevel(pos: int): double
+axis2servoLowLevel(pos: double): int
+moveServo(goal: int, speed, int): bool
+isDeviceMoving(): bool
+getServoPosLowLevel(): int
+testDevice(): bool
+installationProcedure(): bool
+unInstallationProcedure(): bool
+calibrationProcedure(): bool
+onProcedure(): bool
+offProcedure(): bool
- jrkGetVariable(
command: unsigned char,
isSigned: bool): int
- jrkGetVariableLB(
command: unsigned char): int
- jrkGetInput(void): int
- jrkGetTarget(void): int
- jrkGetScaledFeedback(void): int
- jrkGetErrorSum(void): int
- jrkGetDutyCycleTarget(void): int
- jrkGetDutyCycle(void): int
- openPort(num: int): bool
- closePort(void): bool
AxisDriver
+aType: axisType
+minAbsoluteServoValue: double
+maxAbsoluteServoValue: double
+minAbsolutePhysicalValue: double
+maxAbsolutePhysicalValue: double
+setLimits(): bool
+move(goalPosition: double,
speedAsPercentage: double): bool
+getPos(): double
+isMoving(): bool
+stop(): bool
+getServoPos(): int
+servo2axis(pos: int): double
+axis2servo(pos: double): int
+ser2axisLowLevel(pos: int): double
+axis2servoLowLevel(pos: double): int
+moveServo(goal: int, speed: int): bool
+isDeviceMoving(): bool
+getServoPosLowLevel(): int
+testDevice(): bool
DeviceDriver
#name: string
#isEnabled: bool
#isInstalled: bool
#isCalibrated: bool
#isOn: bool
#*pCronOn: Chronometer
+DeviceDriver()
+installation(): int
+uninstallation(): int
+calibration(): int
+On(): int
+Off(): int
+basicInfo(): void
+test(): bool
+testProcedure(): bool
+installationProcedure(): bool
+unInstallationProcedure(): bool
+calibrationProcedure(): bool
+onProcedure(): bool
+offProcedure(): bool
+isAvailable(): bool
«uses»
Chronometer
+name: string
-totalActiveSeconds: long double
+Chronometer(name: string)
+run(): void
+stop(): void
+reset(): void
+info(): void

Figure 6.10: The driver class hierarchy.

6.2.9 Software module “arduinoDriver”

This class is aimed to manage an Arduino board as LED panel. It is a derived class from DeviceDriver. In addition to the implementation of the virtual methods declared in its parent class, the arduinoDriver class provides one additional method called dwrite, which allows to write digital values to the ports of the Arduino.

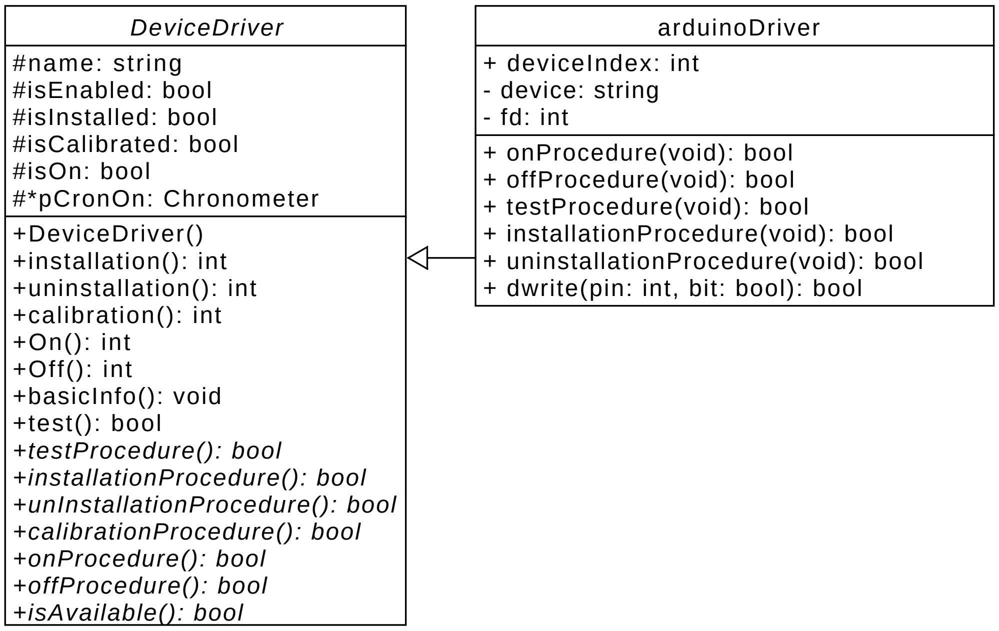


Figure 6.11: Class diagram for module arduinoDriver

6.2.10 Additional class “rotatorDriver”

This class was not used in this project, but it has been used in other projects that use devices with rotary movements. It can be suitable for future robotics projects such as robot arms or mobile robots.

The class rotatorDriver is a derived class from DeviceDriver.

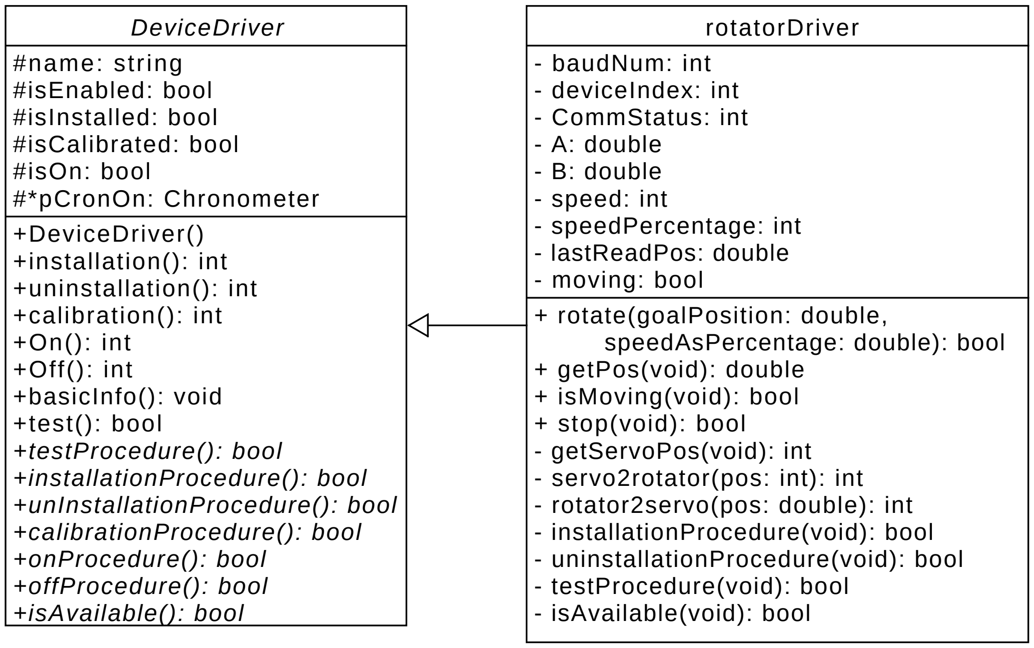


Figure 6.12: Additional class rotatorDriver

6.2.11 Automatic C/C++ Code Generation with MatLab Coder

Some of the controller's software modules, like the inverse kinematics, require the use of advanced mathematical functions, such as matrix multiplication, norm and vector product. While there exist libraries for C++ that provide functions to perform these operations, their use is somewhat difficult, and their syntax is complicated.

MatLab has a tool called Coder, which allows the generation of C/C++ source code from standard .m files. Coder takes a matlab function as input, and produces a header file with the function declaration, a source file with the function definition, a sample test program, and other header files with data types definitions, for use in different platforms.

The function that calculates the inverse kinematics was generated using this approach. The inverse kinematics is programmed as a function in an m-file. The inputs for this function are the parameters of the robot and the desired position. The outputs of the function are the actuators' lengths.

Some considerations must be taken into account when generating source code from .m files:

- There are some MatLab functions for which code generation is not supported.
- The function must be programmed in a particular way to let MatLab infer the data types for the function arguments and local variables used in the code. For instance, matrices and vectors with constant dimensions should be initialized (pre-allocated) inside the function before being used.

- Arrays and matrices that change their size dynamically can be used. However, the resulting code is more complicated and, where possible, should be replaced by fixed size arrays and matrices.
- If the function returns more than one value or number, the generated C function will be of type void and the return values will be listed as input values passed by reference.

After running Coder, the resulting header file was integrated in the inverse kinematics class header file (inverse_kinematics.h), and the implementation of the function was added to the source file (inverse_kinematics.cc). The MatLab funtion that implements the inverse kinematics algorithm is shown in Appendix B.

In this chapter, the implementation of the controller was presented. The mechanical structure was assembled, and an experimental connection board was developed. The software modules had the required functions that were presented in the software architecture, plus additional methods that were necessary for initialization, testing, etc. Finally it was described the code generation for the inverse kinematics function, using MatLab Coder.

Chapter 7

System testing and documentation

This chapter describes the testing and documentation phases of the project, as well as the tools that were used to accomplish these tasks.

7.1 Testing of the source code

Following the V Model for the project development, when the architecture has been designed, it is time to move to writing the code for the software. One of the most important activities in the later phases of development is performing tests on the hardware and software.

Testing consists of conducting specific actions in order to verify the operability, supportability, or performance capability of an item when subjected to real or simulated controlled conditions. The results obtained are then compared with the anticipated or expected ones [24].

7.1.1 Testing levels

Software development involves different abstraction levels: *modules* or *components* which may be developed for the specific system or reused from previous systems or libraries, *subsystems*, which are obtained by integrating sets of related components, and the final *system*, which is obtained by assembling components and subsystems in an application that satisfies initial requirements. Testing is performed at different levels [25]. The following testing levels are distinguished:

1. Module (unit) testing: Modules or components are first verified in isolation usually by the developers who check that the single modules behave as expected.
2. Integration testing: It may happen that subtle failures are caused by unexpected interactions among well-designed modules. Integration testing deals with many communicating modules. Most integration testing strategies suggest testing integrated modules incrementally.

3. System testing: Module and integration testing can provide confidence on the quality of the single modules and on their interactions, but not about the behavior of the overall system. Therefore, it is needed to complete the verification and validation process by testing the overall system against its specifications and requirements. System testing verifies the correspondence between the overall system and its specifications.

4. System Validation: Also called Acceptance testing. While system testing verifies the correspondence between the overall system and its specifications, the System Validation verifies the correspondence between the system and the user's expectations.

System and acceptance testing consider the behavior of the overall system in its functional and non-functional aspects. Module and integration testing are largely based on the internals of the software, which is hardly accessible to the user.

7.1.2 Unit and integration tests framework

Google Test is a testing library for the C++ programming language. Some of the features of Google Test include a rich set of assertions, ability to run some or all tests at once, fatal and non-fatal failures, and XML test report generation.

This framework was used to perform unit and integration tests for the different software modules.

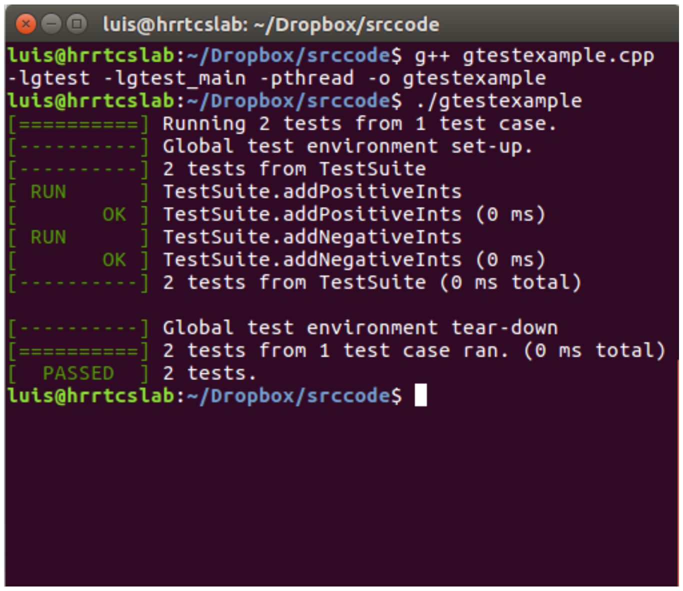
7.1.2.1 An example of a test program

The following example demonstrates how to write a test program using the Google Test framework.

```cpp
// foo.h provides function addInts.
#include "foo.h"
#include <gtest/gtest.h>

// Declare a test
TEST(TestSuite, addPositiveInts) {
    ASSERT_EQ(5, addInts(2, 3));
}
// Declare another test
TEST(TestSuite, addNegativeInts) {
    ASSERT_EQ(-2, addInts(-2, 0));
}

// Run all tests declared with TEST()
int main(int argc, char ** argv) {
    testing::InitGoogleTest(&argc,argv);
    return RUN_ALL_TESTS();
}
```

Listing 7.1: Example code for writing test with Google Test

After compiling and running the program, we get the output shown in Fig. 7.1:



```
luis@hrrtcslab:~/Dropbox/srccode$ g++ gtestexample.cpp
-lgtest -lgtest_main -pthread -o gtestexample
luis@hrrtcslab:~/Dropbox/srccode$ ./gtestexample
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from TestSuite
[ RUN      ] TestSuite.addPositiveInts
[       OK ] TestSuite.addPositiveInts (0 ms)
[ RUN      ] TestSuite.addNegativeInts
[       OK ] TestSuite.addNegativeInts (0 ms)
[----------] 2 tests from TestSuite (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 2 tests.
luis@hrrtcslab:~/Dropbox/srccode$
```

Figure 7.1: Results of running a test program

Google Test runs all the test cases, and summarizes the results of all tests. The advantage of Google Test is that whenever a change is made to the source code, or a new feature is added, the tests can be executed again and if one or more tests fail, this means that an error was introduced in the code. The process of unit testing the software can be automated by writing a test program for every software module (unit testing), and rerunning all tests whenever the source code is recompiled.

7.1.2.2 Automated tests for the project

Table 7.1 describes all the automated tests that were written for this project using Google Test. Most modules were tested with different types of valid and invalid input, and under many different conditions, such as attempting to open non-existent files.

The module LinearDriver was not tested with Google Test, because its functions are closely related to the communication interface, serial port, and the jrk21v3 drivers. Instead, it was tested manually.

Table 7.1: Automated tests for the software modules

Function name	**Test summary**
Module: PositionStorage	
PositionStorage	Invalid constructor parameters.
writePosInFile	Write to an existent file.
readPosFile	Read from existing file.
searchLabel	Search an existing label.
searchLabel	Search a non-existing label.
getPosDB	load the contents of the database to RAM.
deletePosition	Delete an existing position from the DB.
deletePosition	Attempt to delete a non existing position from the DB.
readCSVPos	read robots positions contained in a CSV file.
Module: TaskStorage	
TaskStorage	Empty constructor.
TaskStorage	Non-empty constructor.
readTaskFile	Read the contents of a file and load them to the objects list.
searchTask	Search an existing task.
searchTask	Search a task that does not exist in the task DB.
createTask	Create a new task.
Stringlist.TruncateToFile	Method added to add commands to tasks.
deleteTask	Delete a task and all its contents.
Module: InverseKinematics	
InverseKinematics	Default constructor.
InverseKinematics	Non-empty constructor.
loadParameters	Set parameters after construction.
poseToAct	Inverse kinematics algorithm calculation. Results are compared with MatLab script results.
poseToAct	Calculating the inverse kinematics with an unreachable position.
Module: parameter	
readList	Read an existing configuration file.
readList	Read a non-existing configuration file.
readList	Reading an existing file more than once.
readList	Reading a file with the wrong format.
getValOf	Attempt to find the value of a parameter that was not defined.
show_params	Display parameters and their values.
Module: CommandControlModule	
CLInterpreter	Testing command with no arguments.
CLInterpreter	Testing command with more than once arguments.
CLinterpreter	Testing appropriate call to move robot function.
CLinterpreter	Testing command with wrong number of arguments.
CLInterpreter	Testing command with correct number of arguments.
Module: CS_Manager	
CS_Manager	Non-empty constructor

Continued on next page

define_CS	Define correct CS.
activate_CS	Activate CS given an existing label.
activate_CS	Attempt to activate a non-existing CS.
getActiveCS	Get the name and definition of the active CS, when one has been defined.
getCSData	Read all the CS data from a file.
Module: utilities	
isRelative	Parse numbers written as relative (Total 17 test cases).
isPercent	Parse number written as percent (Total 8 test cases).
Module: MotionControlUnit	
MotionControlUnit	Constructor.

7.1.3 Manual testing

While it is easy and convenient to write automatic tests for individual modules, this is not always the case for higher levels of testing, such as system and acceptance testing. This is because in those levels, many things happen in a test case, or because there are interactions with other hardware components. For instance, when executing a robot task, there are multiple events that happen, such as searching for a given task, calculating inverse kinematics, moving the actuators, etc. It would be very time consuming and difficult to write a program that keeps track of execution and ensures all these operations are performed.

Instead, the preferred approach is to perform system and integration tests manually, and verify that the system behavior corresponds with the expected one. This was done by testing the features of the system manually, and ensuring that the behavior of the system matches the behavior specified in the different use cases and sequence diagrams.

However, the use case diagrams only specify best-case scenarios and show a flow of execution where all elements work correctly. Additional conditions were tested, such as running the software without powering the drivers, or when a wrong port is specified. This information was added to the user manual (Appendix A) as troubleshooting information.

7.2 Documentation of the source code

Technical systems and engineering projects are usually upgraded to add new functionality to an existing system, repurposed to fit a similar need, or have their components reused for a different application. For this reason, it is very important in the projects development lifecycle to write documentation. When the system is upgraded, changed, or a component is reused, reading the documentation will provide an understanding of how the system works, and what actions need to be done in the source code.

Traditionally, software documentation is written in the form of comments in the source code files of a project.

For this thesis project, the program Doxygen was used to document the code [26]. Doxygen reads the comments in the source files, and generates an online documentation browser (in HTML) and/or an off-line reference manual (in LaTeX).

7.2.1 Different types of documentation

With Doxygen, it is possible to document source files, classes, structs, class members (methods and fields), etc. In addition to this, Doxygen automatically generates inheritance diagrams, dependency graphs, and collaboration diagrams, in order to visualize the relationships between various elements.

It is also possible to create documentation in formats other than C/C++ comments. For example, a mainpage can be created in a separate file, using markdown syntax.

7.2.2 Documentation Example

The following example shows how to write comments in order for Doxygen to generate the documentation from them.

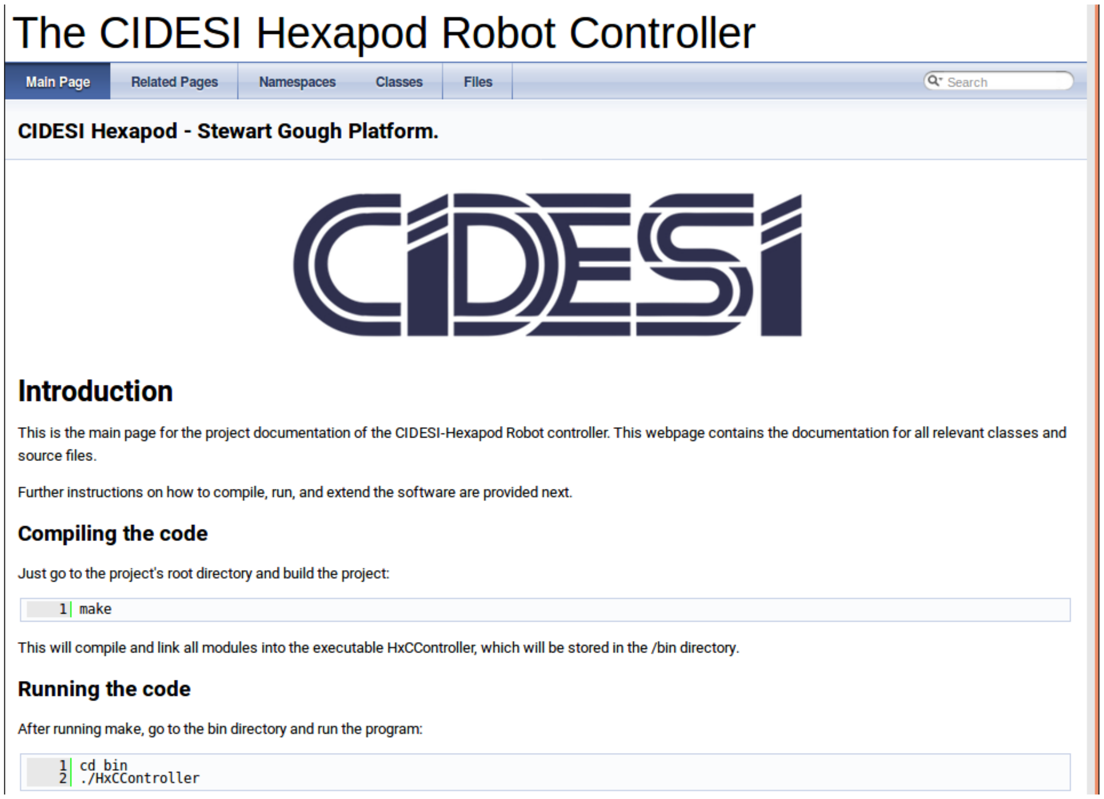
```
/**
 * \file doxy_example.cpp
 * \brief Example of Doxygen style comments.
 *
 * This file shows how to write comments in a source code file in order
   for Doxygen to correctly generate documentation from them.
 * \author Jose Luis German Felix
 */

#include <stdio>

/**
 * \class ExampleClass
 * \brief A class for demonstrating Doxygen comments.
 */
class ExampleClass {
    public:
        /**
         * Add two numbers.
         * \param a the first number to add.
         * \param b the second number to add.
         * \return the sum a + b.
         */
        int add(int a, int b) {
            this->n++;
            return a + b;
        }
    private:
        /**
         * Counts the number of times the function add has been called.
```

```
             */
            int n {};
};
```

Listing 7.2: Example code for writing Doxygen style comments

7.2.3 Source code documentation

The main page for the source code documentation is shown in Fig. 7.2. It is useful for documenting general information, such as an introduction to the code, instructions on how to compile, run and modify the code.



CIDESI Hexapod - Stewart Gough Platform.

Introduction

This is the main page for the project documentation of the CIDESI-Hexapod Robot controller. This webpage contains the documentation for all relevant classes and source files.

Further instructions on how to compile, run, and extend the software are provided next.

Compiling the code

Just go to the project's root directory and build the project:

```
make
```

This will compile and link all modules into the executable HxCController, which will be stored in the /bin directory.

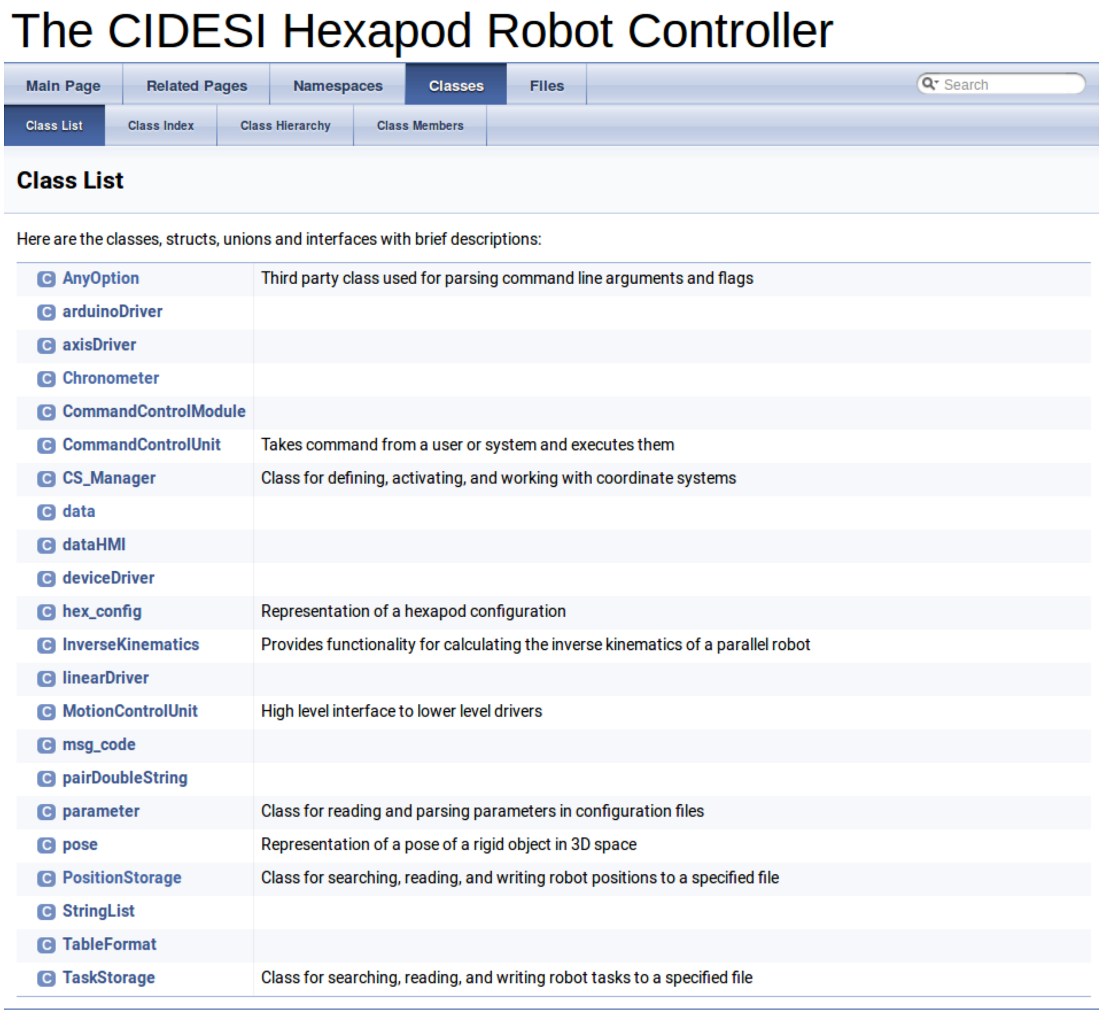
Running the code

After running make, go to the bin directory and run the program:

```
cd bin
./HxCController
```

Figure 7.2: Main page for the source code documentation.

There is a menu bar with five buttons: *Main Page*, *Related Pages*, *Namespaces*, *Classes*, and *Files*. The *Main Page*, as its name suggests, directs the user to the start page of the documentation. The *Related* Pages button displays a list of files that are not source code files, but contain documentation. The *Namespaces* button displays all namespaces that are used in the project. The most important is the *Classes* section, shown in Figure 7.3. It contains a list and brief description of all classes in the project. It is possible to list the classes alphabetically or as a Hierarchy diagram, which shows the inheritance relation between the different classes. Additionally, the browser allows to list all class members in the project.


The CIDESI Hexapod Robot Controller

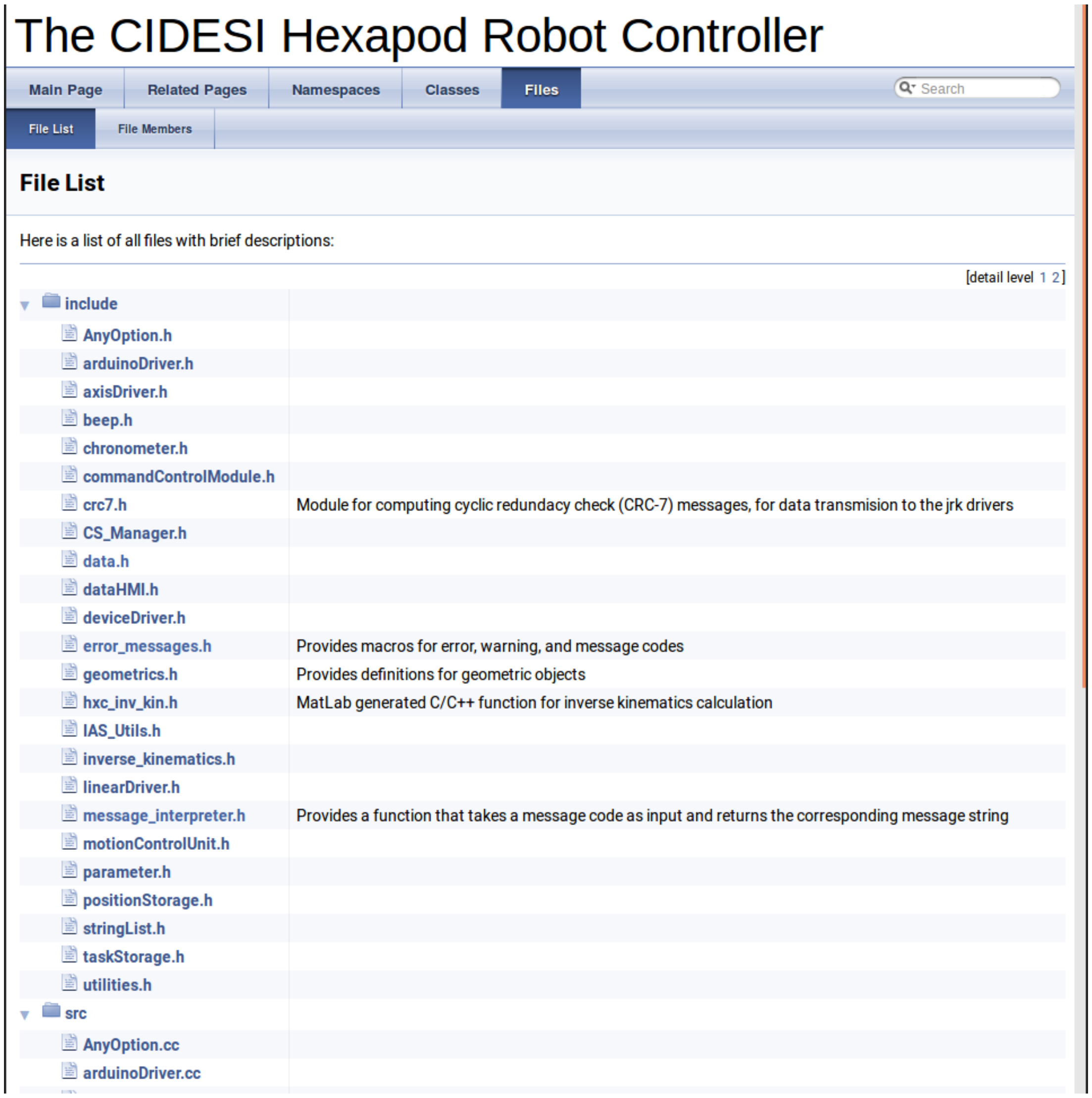
Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

AnyOption	Third party class used for parsing command line arguments and flags
arduinoDriver	
axisDriver	
Chronometer	
CommandControlModule	
CommandControlUnit	Takes command from a user or system and executes them
CS_Manager	Class for defining, activating, and working with coordinate systems
data	
dataHMI	
deviceDriver	
hex_config	Representation of a hexapod configuration
InverseKinematics	Provides functionality for calculating the inverse kinematics of a parallel robot
linearDriver	
MotionControlUnit	High level interface to lower level drivers
msg_code	
pairDoubleString	
parameter	Class for reading and parsing parameters in configuration files
pose	Representation of a pose of a rigid object in 3D space
PositionStorage	Class for searching, reading, and writing robot positions to a specified file
StringList	
TableFormat	
TaskStorage	Class for searching, reading, and writing robot tasks to a specified file

Figure 7.3: Classes section for the project documentation.

Finally, the *Files* section, shown in Figure 7.4, displays all the files that were parsed by doxygen, and the directory structure. It shows the documentation of files that don't implement classes, but implement useful functions for other classes to use.



File List

Here is a list of all files with brief descriptions:

[detail level 1 2]

▼ include	
AnyOption.h	
arduinoDriver.h	
axisDriver.h	
beep.h	
chronometer.h	
commandControlModule.h	
crc7.h	Module for computing cyclic redundancy check (CRC-7) messages, for data transmision to the jrk drivers
CS_Manager.h	
data.h	
dataHMI.h	
deviceDriver.h	
error_messages.h	Provides macros for error, warning, and message codes
geometrics.h	Provides definitions for geometric objects
hxc_inv_kin.h	MatLab generated C/C++ function for inverse kinematics calculation
IAS_Utils.h	
inverse_kinematics.h	
linearDriver.h	
message_interpreter.h	Provides a function that takes a message code as input and returns the corresponding message string
motionControlUnit.h	
parameter.h	
positionStorage.h	
stringList.h	
taskStorage.h	
utilities.h	
▼ src	
AnyOption.cc	
arduinoDriver.cc	

Figure 7.4: Files section for the project documentation.

This chapter described the two important activities that in the development of the system. Many forms of testing were performed in order to assure that the software has as few errors as possible, and that the overall system fulfills the requirements.

Documentation involved the creation of documents that can be used as reference for the user of the system, and for future developers of the system. The different software tests that were developed are also an effective form of documentation, because the developers can see what features the modules have, how they are used initialized, and how the modules react against errors. The development of tests also ensures that new features can be added to the code base, without breaking the existing ones.

Chapter 8

Results

In this chapter, the results achieved are presented, as well as some problems and limitations that were encountered during the system development.

8.1 Software highlighted features

Regarding the development of the controller software, the following results were achieved:

- The controller software manages the movement of the robot by communicating with the jrk21v3 drivers via the serial port.
- The positions for the robot can be specified with coordinates, from a previously stored robot position, or by reading a file that contains robot positions. Additionally, the actuators lengths can be specified.
- The user can program robot tasks. These are based on previously executed commands and they are stored in a task database.
- The user is able to define a user home position and move the robot to the user or the universal home position.
- The user is able to define, edit and delete coordinate systems. However, the coordinate system transformation was not implemented, and only the base coordinate system is used.
- The user is able to define, edit, and delete robot positions.
- Different functionalities were added for setting parameters, and for calibrating the actuators.
- A user manual was created (see Appendix A), as well as an online source code documentation, for future developers of the software.
- The source code is thoroughly documented in the form of a web browser, which contains useful information about the different modules, as well as instructions on how to compile,

run, modify and extend the functionality of the program.

8.2 Path following of the robot

A function was developed to read robot positions from a file, and send them to the drivers at a given time interval between each move command. The robot positions for a given trajectory are calculated in MatLab and saved as a csv file, which is then read by the controller.

This function allows to test the possibilities of the robot to follow different paths. The different robot positions can be precomputed in MatLab, and by selecting a suitable time interval, the robot can follow the desired path. By commanding a precomputed trajectory, and measuring the robot's actual positions, it is possible to use this functionality for parameter identification of the actuators.

8.3 Robot workspace

The robot workspace was explored by moving the robot to extreme positions, where the mechanical elements (joints) are close to colliding with each other. The mobility tests show that the robot workspace is not only limited by the actuators' lengths, but also by the mechanical limits of the joints. For this reason, it is needed to develop code that prevents the robot from positions that might damage its mechanical components, even if the actuators are capable to reach this position. This work is still in progress.

Chapter 9

Conclusion

In this chapter, the thesis conclusion is presented, and then future work advice is given.

9.1 Project Conclusion

The aim of this thesis project was the design and implementation of a robot controller. This included hardware, software, and communication interfaces.

However, the main contribution in this project was in the development of the controller software.

This was achieved by using a Systems Engineering approach and applying the V Model. The systems requirements and concept of operations were the input to this thesis work. The analysis of the requirements, and consideration of available resources led to the development of a preliminary software design.

This preliminary design was then refined into a detailed software architecture. The software, as well as the system, was developed, tested, and a system verification was performed.

As a summary, most of the requirements were fulfilled given the time and resources available. There were some problems related to the mechanical design, as well as some limitations with the current drivers. Nevertheless, the controller accomplished its objective to provide basic functionalities for demonstration and experimental purposes.

9.2 Future Work

The following are possible future tasks related to the development of the CIDESI Hexapod project. Some of these tasks are needed in order to solve some of the problems and limitations that were discussed in chapter 8, while other are related to providing additional functionality to the system.

- Addition of Real Time support in Linux by way of using a real time patch for the Linux kernel, e.g. RTAI. The review of literature showed that a lot of research in robotics is being focused in the use of real-time operating systems. Moreover, specific applications (medical, safety), have hard-real time requirements of the system. The addition of real-time support will allow the application of the system in real-time applications.
- Integration of the software with ROS. Some applications, such as collaborative robotics, require the use of simulated environments, in order to test that the robots behave as expected. ROS may be required for some of these applications, especially if the Hexapod is expected to work together with another robot.
- Integration of the robot with the CIDESI Linear actuator. Improved actuators and drivers are considered for the following phases of the system.
- Implementation of a Teach Pendant, by way of an Android application. This task is considered for future phases of the system.
- Integration of the robot with different technologies. With a working prototype of the robot, it is possible to research different applications and integration with different technologies, such as Industry 4.0, and Internet of Things, by using an external or automated system to command the robot to perform different tasks.
- Implementation of a GUI to facilitate the system usage. A GUI may be implemented in order to facilitate the use of the robot to an operator and make the software more intuitive to use. Some previous works in this area have been developed by members of the automated systems department, in CIDESI.

Bibliography

[1] *Physik Instrumente - Hexapod blog*. URL: `http://www.pi-usa.us/blog/50-hexapod-6-dof-alignment-systems-for-the-sub-reflectors-of-alma-the-largest-telescope-project-in-the-world/` (visited on 05/07/2018).

[2] Gengis K. Toledo Ramírez. "Desarrollo de sistemas con la metodología de Sistemas de Ingeniería". *Internal document not published, CIDESI-CONACYT, Mexico* (2018-04-17).

[3] J. P. Merlet. *Parallel Robots*. Springer.

[4] Lung-Wen Tsai. *Robot Analysis- The Mechanics of Serial And Parallel Manipulators*. John Wiley & Sons, 1999.

[5] D. Stewart. "A platform with six degrees of freedom." *Proceedings of the Institution of Mechanical Engineers.* 180 (1965), pp. 371–386.

[6] Se-Han Lee et al. "Position control of a Stewart platform using inverse dynamics control with approximate dynamics". *Mechatronics* 13 (2003), pp. 605–619.

[7] Alessandro Macchelli and Claudio Melchiorri. "A Real Time Control System for Industrial Robots and Control Applications Based on Real-Time Linux". *IFAC Proceedings Volumes* 35 (2002), pp. 55–60.

[8] Adrian Gambier. "Real-time control systems: a tutorial". *5th Asian Control Conference* (2004).

[9] Deng Liming and Zhao Xianchao. "A real-time walking robot control system based on Linux RTAI". *Proceedings of the 2013 International Conference on Advanced Mechatronic Systems* (2013), pp. 530–534.

[10] A. Barbalace et al. "Performance Comparison of VxWorks, Linux, RTAI and Xenomai in a Hard Real-Time Application". *IEEE Transactions on Nuclear Science* 55.1 (2008), pp. 435–439.

[11] Hongxing Wei et al. "RT-ROS: A real-time ROS architecture on multi-core processors". *Future Generation Computer Systems* (2016), pp. 171–178.

[12] Tomas Docekal and Zdneck Slanina. "Control System Based on FreeRTOS for Data Acquisition and Distribution on Swarm Robotics Platform". *Carpathian Control Conference (ICCC), 2017th 17th International* (2017), pp. 434–439.

[13] Miguel-Angel Martínez Prado. "An FPGA-Based Open Architecture Industrial Robot Controller". *IEEE Access* (2018-01-24).

[14] *Executive Summary World Robotics 2017 Industrial Robots*. 2017. URL: `https://ifr.org/downloads/press/Executive_Summary_WR_2017_Industrial_Robots.pdf` (visited on 05/07/2018).

[15] *Physik Instrumente - Hexapod blog*. URL: `http://www.hexapods.net/` (visited on 05/07/2018).

[16] *F-200iB product information sheet*. URL: `https://www.fanucamerica.com/docs/default-source/robotics-product-information-sheets/f-200ib-series_9.pdf` (visited on 05/07/2018).

[17] Gengis K. Toledo Ramírez. "Hexápodo-CIDESI", fase 1. Conceptos de Operación y Requerimientos". *Internal document not published, CIDESI-CONACYT, Mexico* (2018-05-24).

[18] Ernesto Salazar-Joya. *Desarrollo de DEMO para robot hexápodo FANUC*. 2014.

[19] Víctor David Acosta-Abraham. *Truck simulator mechanism development, preliminary stage*. Internal document not published, CIDESI-CONACYT, Mexico, 2016.

[20] Gerson Andrés Díaz López. *Desarrollo de prototipo Alfa de Plataform Stewart-Gough (Hexápodo)*. Internal document not published, CIDESI-CONACYT, Mexico, 2017.

[21] Rubén Ordaz Madrigal. *Diseño de plataforma Stewart-Gough "Hexápodo-CIDESI", fase 1. Pare Mecánica*. Internal document not published, CIDESI-CONACYT, Mexico, 2017.

[22] José Ramón Barajas Fletes. *Diseño de detalle de prototipo alfa Hexápodo CIDESI - Fase I*. Internal document not published, CIDESI-CONACYT, Mexico, 2018.

[23] *Jrk User's Guide*. URL: `https://www.pololu.com/docs/pdf/0J38/jrk_motor_controller.pdf` (visited on 07/15/2018).

[24] Mourad Debbabi et al. *Verification and Validation in Systems Engineering*. Springer, 2010.

[25] Luciano Baresi. "An Introduction to Software Testing". *Electronic Notes in Theoretical Computer Science* (2006).

[26] *Doxygen website*. URL: `http://www.stack.nl/~dimitri/doxygen/index.html` (visited on 06/11/2018).

[27] Andres Sala-Almonacil. *Development of a Real-Time Controller for a Robot Arm*. 2011.

Appendix A

User Manual

CIDESI Hexapod system
User manual

José Luis Germán Félix

August, 2018

Version 0.6

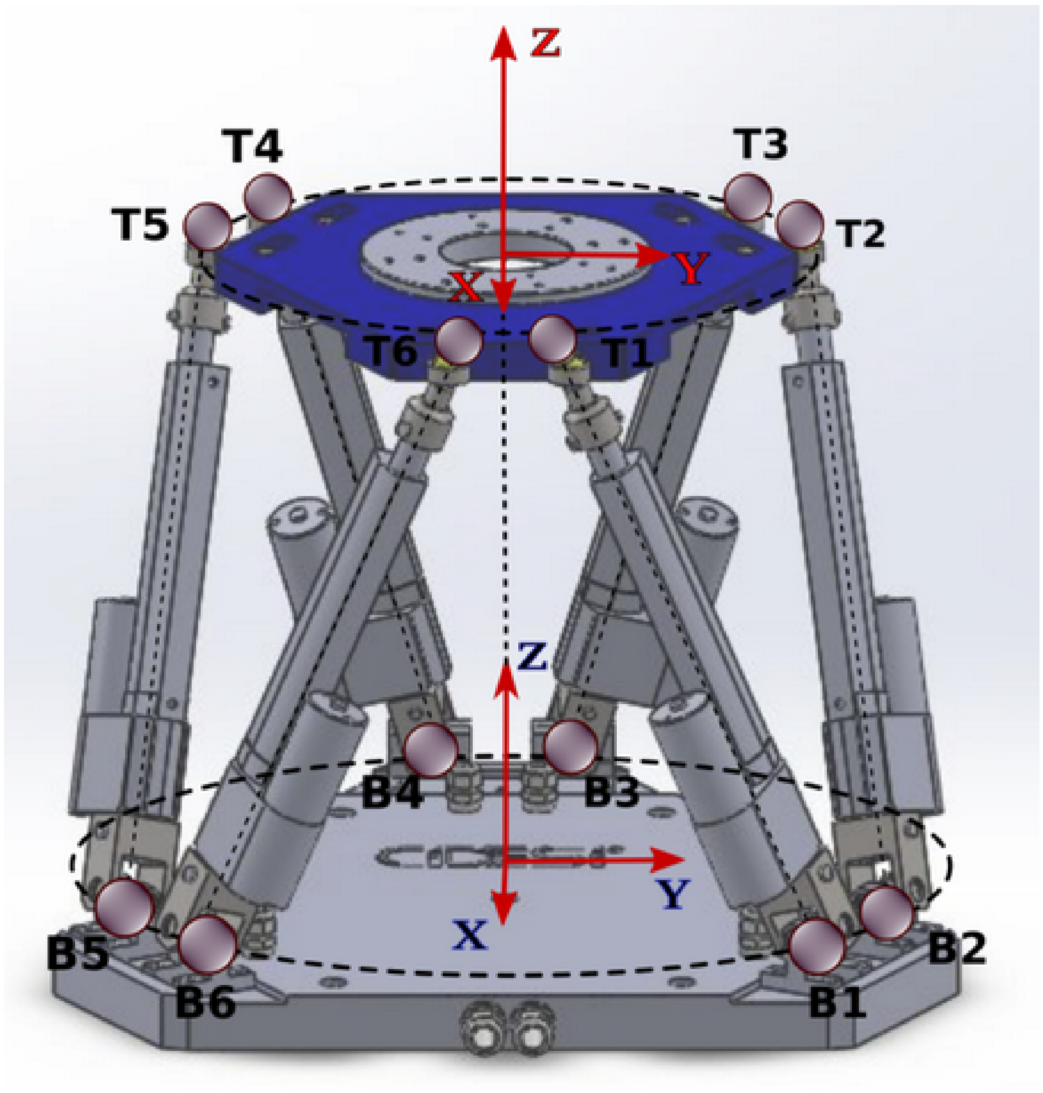


Table 1: Document version control

Version	Date	Notes/changes
0.1	2018.08.18	First version
0.6	2018.17.18	Added section: SW installation and minor changes in structure

1 Introduction

The CIDESI Hexapod is a robotic system based on the Stewart Gough platform design.

This user manual contains all information needed to operate the robotic system. This information includes hardware connections, power-up procedure, robot operation via the controller software, power-off procedure, and troubleshooting.

2 Hexapod system components

The Hexapod Robotic System includes the following components:

- Mechanical structure
- Robot controller
- Power supply
- Controller software (PC not included)

3 Hardware setup and connection

Before powering any device, please make sure that all devices and sub-systems are correctly connected.

Unless the controller cabinet is rewired or the robot is disassembled and reassembled, the usual connections that must be checked are the USB cable from the master driver, the cable and connections between the mechanical structure and controller, and the power supply connection.

4 Power ON procedure

The systems should be started in the following sequence in order to avoid errors in the drivers.

1. Turn ON the PC.

2. Turn ON the power supply for the actuators.
3. If using an additional power supply for the 5V line, turn in ON.

At this point, the jrk drivers should be in the "Awaiting command" mode, and the orange light of the drivers should be blinking.

5 Command Line operation

The program provides a Command Line Interface for interacting with the robot operator. The user types a command, and the software parses and executes the command. As the command is executed, information related to the execution and state of the system is displayed. When the command has been executed, the user can type another command.

6 Program start

The controller software was developed for the Linux operating system. Therefore, it is assumed that the robot operator is familiar with Linux and has basic knowledge with using the terminal.

In order to start the software, follow the next instructions:

- Make sure all specified parameters in the *configuration* file are correct.
- Navigate to the directory of the software controller, usually HxCController/.
- Navigate to the sub-directory HxCController/bin contains the executable program, as well as other files which will be described later.
- Execute the program by running: ./HxCController.

When the software starts, a number of steps are performed:

- Software modules are initialized with the parameters.
- The serial port is open.
- The drivers are initialized.

Detailed information about the start sequence is displayed on the screen when the program starts. When the start sequence finishes, the user is prompted for input. If the program fails to finish the start sequence or an error is displayed, see section 10.

7 Commands description

This section describes the commands that are offered to the user.

7.1 Structure of a command

A command is a string of text, which contains tokens separated by spaces. A token can be the name of a command, a number, an identifier, an option, etc.

The command structure is as follows:

<command_name> <argument_1> <argument_2> ... <argument_n>

The first token in every command must always be the name of the command. All remaining tokens are arguments of a command. Some commands take no arguments, while other commands, may require up to six different arguments.

The commands have a short and a long name, and either can be used as the name of a command. In the following section, the long name of the command is shown, and the short name is shown in parenthesis for each command.

7.2 Information and help commands

7.2.1 help (h)

- Arguments: None
- Description: This command displays a list of all available commands, as well as the maximum arguments it can take, and a short description of the command.

7.2.2 info (i)

- Arguments: None
- Description: This command displays information on the current state of the system, such as the program's parameters, the robot's current position, and other data which will be explained in the following sections.

7.2.3 version (v)

- Arguments: None
- Description: This command displays the current version of the software.

7.3 Manual Mode Commands

7.3.1 move-actuator (mva)

- Arguments: n d

- Description: Moves the actuator **n** a distance **d** millimeters. The distance can be specified as an absolute distance, or as a percentage. The following are valid examples for a Hexapod robot with actuators with a minimal length of 10 mm and maximal length of 140 mm:
  - mva 1 50
  - move-actuator 6 35%
  - mva 4 140

7.3.2 move (mv)

- Arguments: -x -y -z -r -p -w
- Description: Moves the robot to the position specified by the coordinates. The coordinates **x**, **y**, and **z** are specified in millimeters, while the coordinates **r** (roll), **p** (pitch), and **w** (yaw) are specified in degrees.

  This command accepts the six coordinates in the order **x**, **y**, **z**, **r**, **p**, **w**, all of them separated by spaces.

  The following are valid examples of the move command, when six coordinates are specified:
  - mv 0 0 450 0 0 0
  - mv 75 -75.5 450 0 0 7.2

  It is also possible to specify only some coordinates. In this mode, coordinates must be preceded by a dash (-) and the coordinate identifier. All unspecified coordinates are set to 0.

  The coordinates can be specified as absolute or relative coordinates. Absolute coordinates are simply written as their value, such as -x450 (450 mm in x direction). Relative coordinates are written with two signs preceding the number, which indicate if the distance is added to or subtracted from the robot's current position. E.g. -z--22.5 (subtract 22.5 mm from the robot's current z coordinate).

  The following are valid examples for a Hexapod robot with actuators with a minimal length of 10 mm and maximal length of 140 mm, specifying only some coordinates:
  - mv -z450
  - mv -z450 -w15
  - move -z450 -x100 -w15
  - move -z++30.5 -x--20.5 -w-10.0
- Additional information:

- If a position cannot be reached by the actuators, an error will be displayed and the command will be ignored.
- If the command is correctly executed (a valid position is provided and the robot reaches the position), the command will be stored in the *last command* variable.
- The program will check the position of the robot, and the user will be able to introduce another command only after the robot has reached the target position.

7.3.3 move-csv (mcsv)

- Arguments: filename dt
- Description: Move the robot to the positions specified in **filename**, with a time **dt** (milliseconds) between each move command.
- Additional information:
  - The file must consist of six comma-separated-values, corresponding to the six coordinates (x, y, z, r, p, w).
  - The software does not check whether the previous position was reached, before issuing the next command.
  - After the command is completed, it will be stored in the *last command* variable.

7.3.4 move-lengths (ml)

- Arguments: l_1 l_2 l_3 l_4 l_5 l_6
- Description: Move the robot actuators to the specified lengths $l_1, ..., l_6$, in that order. The following is a valid example of this command, which will move the first three actuators to a length of 50 mm and the last three actuators to a length of 80 mm:
  - ml 50 50 50 80 80 80
- Additional information:
  - It is necessary to provide all six actuator lengths.
  - If one of the lengths is out of the specified limits, an error message will be displayed and the command will be ignored.
  - If the command is correctly executed (valid lengths are provided and the robot reaches the position), the command will be stored in the *last command* variable.

7.4 Robot positions commands

7.4.1 read-actuator (ra)

- Arguments: None
- Description: Display the position of all actuators.

7.4.2 save-position (sp)

- Arguments: name
- Description: Saves the robot's current position to the robot's position database under the given **name**. The robot's current position can be displayed with the info (i) command.

7.4.3 restore-pos (rp)

- Arguments: name
- Description: Move the robot to the position specified by **name**. The argument **name** must be a previously stored label in the robot positions database. In order to see all stored robot positions, run the command list-saved-pos (lp).
- Additional information:
  - After the robot reaches its target position, the command will be stored in the *last command* variable.
  - If a robot position with the given name is not found in the database, an error message will be displayed and the command will be ignored.

7.4.4 delete-pos (dp)

- Arguments: name
- Description: Delete the robot position specified by **name** from the robot positions database. The argument **name** must be a previously stored label in the robot positions database. In order to see all stored robot positions, run the command list-saved-pos (lp).
- Additional information:
  - If a robot position with the given name is not found in the database, an error message will be displayed and the command will be ignored.

7.4.5 list-saved-pos (lp)

- Arguments: None
- Description: Display all stored robot positions.

7.5 Home positions commands

7.5.1 move-uhome (mv-uhome)

- Arguments: None
- Description: Move the robot to the user-defined home position.
- Additional information:
  - When the program starts, this position is equal to the universal home position.
  - This position is lost when the program exits.

7.5.2 move-home (mv-home)

- Arguments: None
- Description: Move the robot to the universal home position.
- Additional information:
  - This position cannot be changed by the user.

7.5.3 save-position-home (sp-home)

- Arguments: None
- Description: Save the robot's current position as the user-home position.

7.6 Automatic Mode commands

7.6.1 task-define (tdef)

- Arguments: name
- Description: Defines a new task with the given **name** in the task database.

7.6.2 task-add (tadd)

- Arguments: name
- Description: Adds the command stored in *last command* to an existing task with the provided **name**.
- Additional information:
  - If the task does not exist, an error message will be displayed and the command will be ignored.
  - If the *last command* variable is empty, an error message will be displayed and the command will be ignored.

7.6.3 task-list (tls)

- Arguments: None
- Description: Lists all defined tasks and their associated commands.

7.6.4 task-execute (tex)

- Arguments: name
- Description: Execute the task with the given **name**.
- Additional information:
  - Each command associated with the task is executed only after the previous command has finished executing.
  - Information of the execution of each command is displayed as the command is being executed.

7.7 Coordinate Systems Commands

7.7.1 cs-define (csdef)

- Arguments: name
- Description: Saves the robot's current position as a coordinate system with the given **name** in the coordinate systems database.

7.7.2 cs-set (csset)

- Arguments: name
- Description: Activates a previously defined coordinate system with the given **name** as the current coordinate system. In order to see the current coordinate system name and definition, run the info (i) command.

7.7.3 cs-list (csls)

- Arguments: None
- Description: Lists all defined coordinate systems in the coordinate systems database.

7.7.4 cs-delete (csdel)

- Arguments: name
- Description: Delete the coordinate system with the given **name** from the database.

7.8 Additional commands

7.8.1 sleep (slp)

- Arguments: n
- Description: This command makes the software sleep (wait) for **n** milliseconds, and then returns.
- Additional information:
  - **n** must be a positive integer.
  - After this command is executed, it will be stored in the *last command* variable.

7.8.2 driver-info (di)

- Arguments: n
- Description: Displays information of the current state of the driver **n**.
- Additional information:
  - If one of the jrks shows an error (red light), running this command will clear the error flags.

  - If the error condition (for example, motor power disconnected) remains after running this command, the jrk will still show an error.

7.8.3 exit (q)

- Arguments: None
- Description: Performs power-off procedure and exits the software.

8 Power OFF procedure

The preferred way to power off the entire system is as follows:

1. If the HxCController software is executing a command, wait until the command is finished and then exit the software by running the exit (q) command.
2. The exit command will move the robot to the home position. **Please wait for the robot to reach its home position.**
3. Turn OFF the power supply for the drivers.
4. If used, turn OFF the power supply for the 5V line.
5. Turn OFF PC.

9 Configuration parameters

When the program starts, it loads different parameters from a file called *configuration*, which is found in the /bin directory. This file contains a set of parameters, followed by their value. The values should be properly determined and registered in the file before running the program.

These values are described next:

- **MOBILE_PLATFORM_ RADIUS**: The radius, in millimeters, of the mobile platform. This parameter is needed in order to calculate the actuator lengths, and their value should be measured as indicated in Fig. 3.
- **BASE_PLATFORM_RADIUS**: The radius, in millimeters, of the base platform. This parameter, should be measured as indicated in Fig. 2.
- **MOBILE_PLATFORM_THETA_P**: The angle, in degrees, between a pair of actuators in the mobile platform. This parameter, should be measured as indicated in Fig. 3.
- **BASE_PLATFORM_THETA_B**: The angle, in degrees, between a pair of actuators in the base platform. This parameter, should be measured as indicated in Fig. 2.

- **ACTUATOR_CONTRACTED_LENGTH**: The distance between the joints of the base frame and the mobile frame of the actuator in its contracted position. This parameter should be measured as indicated in Fig. 4.
- **ACTUATOR_MIN_LENGTH**: The minimal allowed length of the actuators. Any attempt to move an actuator below this length will display an error message, and the command will be ignored.
- **ACTUATOR_MAX_LENGTH**: The maximal allowed length of the actuators. Any attempt to move an actuator beyond this length will display an error message, and the command will be ignored.
- **SERIAL_PORT_NUMBER**: The number that was assigned by the operating system to the serial port. This value is usually 0 if no other devices have been assigned first. For problems related to the serial port, and instructions on setting this value, see section 10.
- **CALIBRATION_ACT_N**: This parameter is a calibration distance which is added to each actuator when moving them. This is done to compensate for possible variations in the dimensions of the joints, or actuators. There is one calibration parameter for each actuator, from 1 to 6. These values should be obtained experimentally. If on doubt, leave them with their preconfigured values, or set them to 0.

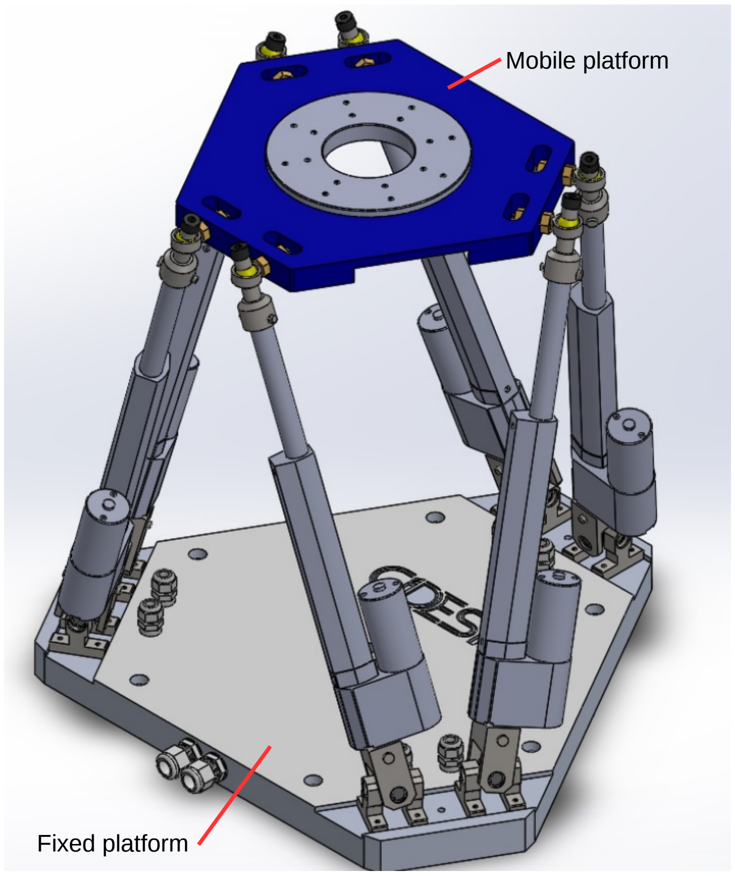


Figure 1: Hexapod platforms identification

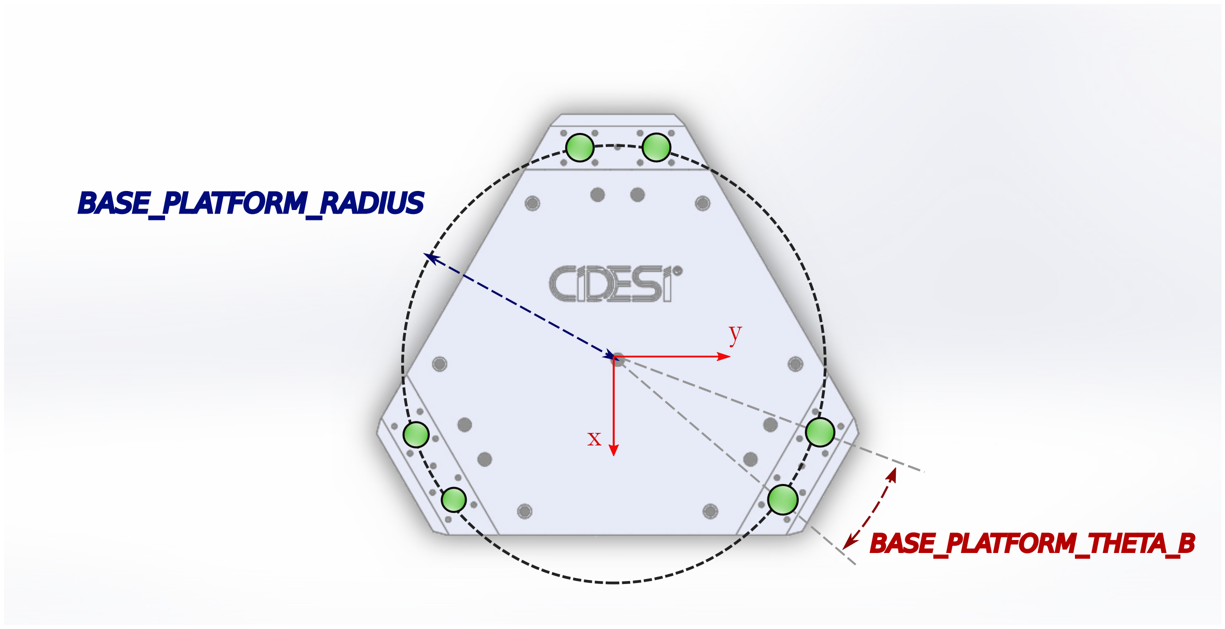


Figure 2: Base platform parameter identification.

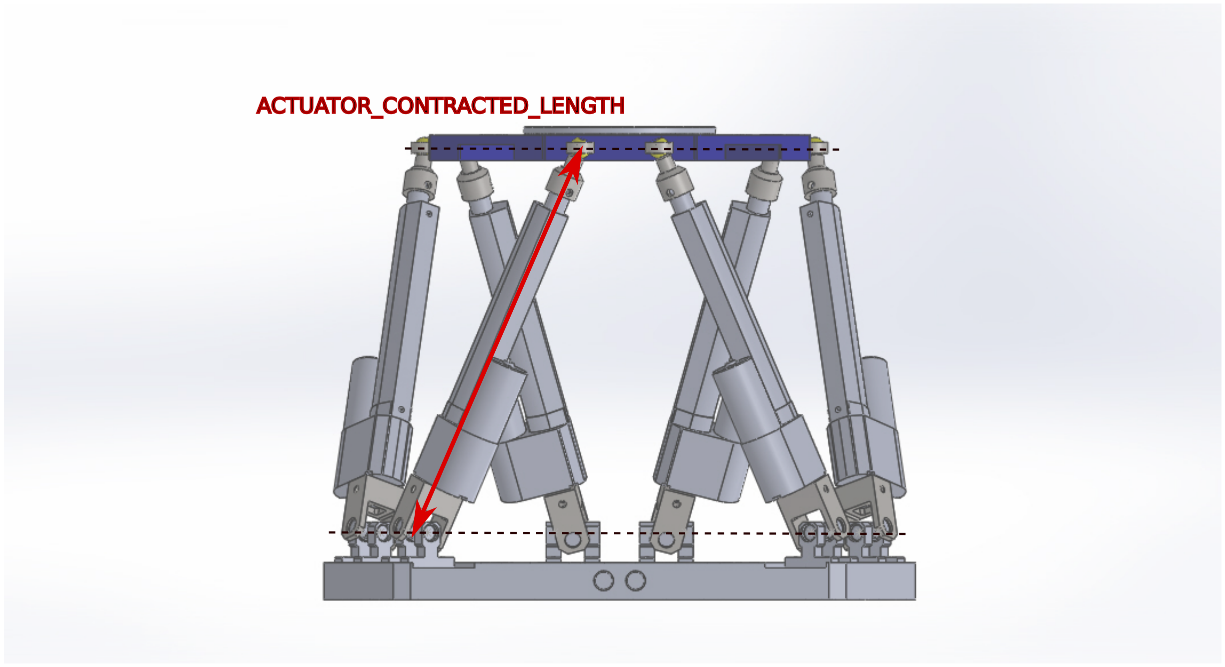


Figure 4: Actuator length parameter identification.

10 Troubleshooting

Many error conditions may occur during the system operation, and others may even prevent the system from correctly starting up. Some of them may be caused by incorrect hardware

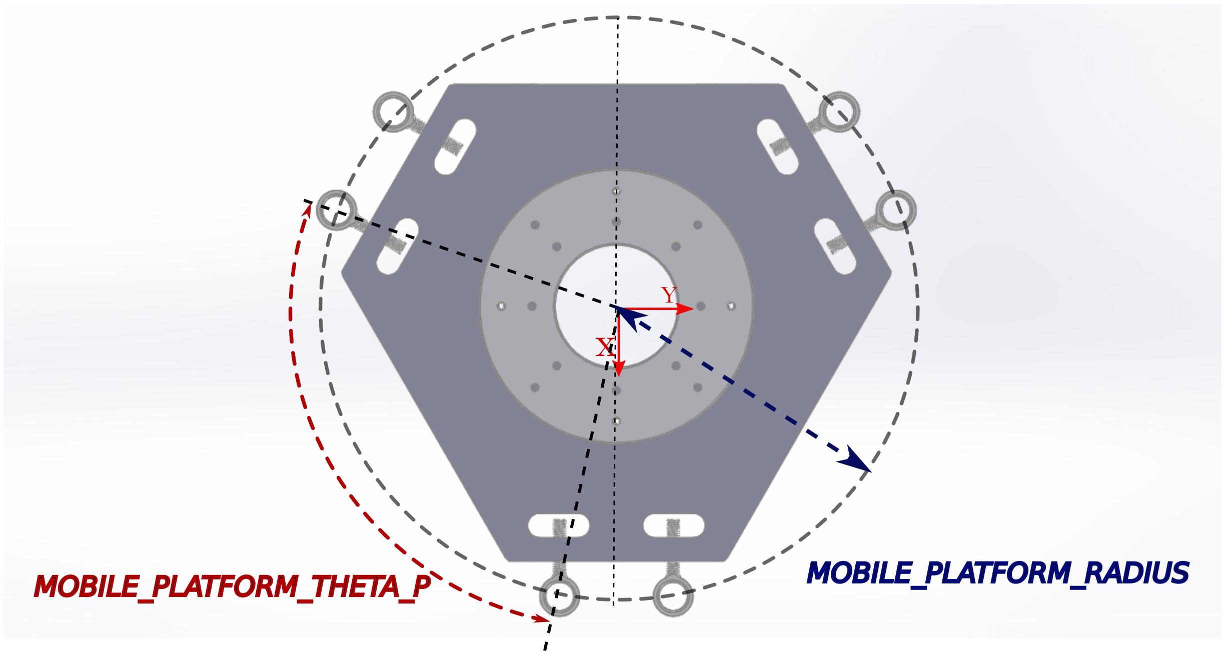


Figure 3: Mobile Platform parameter identification.

connections, others may be due to communication errors, and others can be caused by a problem with the robot structure, defect on the actuators, or wrong configuration parameters.

This section classifies errors in *hardware*, *software*, *communication*, and *robot* errors, and provides possible causes and solutions for these errors.

10.1 Hardware errors

10.1.1 Only one driver is on

- Symptom: All driver have their LED turned off, and the only driver that is on is the one connected to the PC.
- Cause : The power supply to the drivers is off.
- Solution: Make sure that the drivers are supplied with a 12V connection on their power supply terminals.

10.2 Driver errors

10.2.1 Blinking red light on driver(s)

- Symptoms: A red light is blinking on one of the drivers
- Cause: The jrk has detected an error condition and will not drive the motor.

- Solution: Identify the number (n) of that driver and run the command in the HxCController software:
  >>> di n
  This will display the errors associated with that driver, and clear its error flags. Correct the error causes that are displayed and run the command again.

10.3 Communication errors

10.3.1 Serial port could not be opened.

- Symptoms: When running the program, an error shows on the screen: "Error opening serial port".
- Cause: A wrong port number was specified in the configuration file. When plugging in the USB cable, the jrk drivers appear as two serial ports in Linux, usually under the names ttyACM0 and ttyACM1. However, if another serial device under the same name is connected, different names will be assigned to the jrk ports.
- Solution: Unplug the USB cable for the jrk drivers and run the command:
  $ ls /dev/ttyACM*
  This will return a list with the devices registered under the name ttyACM. Then, plug the USB cable and run the command again:
  $ ls /dev/ttyACM*
  Now, two additional devices should be listed. The device with the lower number is the jrk driver serial port. Now open the file *configuration* in the software's /bin directory and change the corresponding value for the parameter SERIAL_PORT_NUMBER.

10.3.2 The software fails to receive response from the jrk drivers

- Symptoms: When running the program, it is possible to send commands to the drivers, but not to read responses from the drivers.
- Cause: A problem with the RX line. This can happen because the 5V supply is disconnected, or because there is a wrong connection in the hardware.
- Solution: Ensure that the 5V power supply is ON and ensure all hardware connections are correctly wired.

10.4 Robot errors

10.4.1 Robot goes to wrong position

- Symptoms: When moving the robot, it moves to the wrong position.

- Cause: The most likely cause for this is that the geometry parameters in the file *configuration* have wrong values.
- Solution: Measure the robot's geometric parameters as indicated in Figures 2-4, and save those values in the *configuration* file.

10.5 Additional error reporting and help

If an error does not appear here, please report it to
luisgermanfelix@gmail.com
or
gengiskanhg.geo@yahoo.com

Appendix B

Inverse kinematics implementation

```
%% function inverse kinematics for the CIDESI Hexapod phase 1.

% the inputs are the desired pose, and the outputs are the required
% actuator lengths.

% Note: When converting this .m code to C code, the outputs l1,...,l6
  will
% be converted to inputs which are passed by reference.

% parameters:
% rp moving platform radius in millimeters
% rb base platform radius in millimeters
% tP angle between moving base actuators in degrees
% tB angle between base frame actuators in degrees
% x, y, z, roll, pitch, yaw: coordinates of desired pose in millimeters

% outputs:
% l1,...,l6: actuator lengths in millimeters
function [l1, l2, l3, l4, l5, l6] = hxc_inv_kin(rp, rb, tP, tB, x, y, z,
    roll, pitch, yaw)
% convert input degrees to radians
tP=tP*pi/180;
tB=tB*pi/180;
roll=roll*pi/180;
pitch=pitch*pi/180;
yaw=yaw*pi/180;

% Moving platform
% Angle of position vectors of Ti in moving frame
Lambda1=(1*pi/3-tP/2);
Lambda3=(3*pi/3-tP/2);
Lambda5=(5*pi/3-tP/2);
Lambda2=(Lambda1+tP);
Lambda4=(Lambda3+tP);
Lambda6=(Lambda5+tP);

% Base(fixed) platform
% Angle of position vectors of Bi in fixed frame
Ipsilon1=(1*pi/3-tB/2);
Ipsilon3=(3*pi/3-tB/2);
Ipsilon5=(5*pi/3-tB/2);
Ipsilon2=(Ipsilon1+tB);
Ipsilon4=(Ipsilon3+tB);
Ipsilon6=(Ipsilon5+tB);

% Position vectors for points Ti in moving frame coords
GT1=[rp*cos(Lambda1);rp*sin(Lambda1);0];
GT3=[rp*cos(Lambda3);rp*sin(Lambda3);0];
GT5=[rp*cos(Lambda5);rp*sin(Lambda5);0];
GT2=[rp*cos(Lambda2);rp*sin(Lambda2);0];
GT4=[rp*cos(Lambda4);rp*sin(Lambda4);0];
GT6=[rp*cos(Lambda6);rp*sin(Lambda6);0];

% Position vectors for points Bi in fixed frame coords
B1=[rb*cos(Ipsilon1);rb*sin(Ipsilon1);0];
B3=[rb*cos(Ipsilon3);rb*sin(Ipsilon3);0];
B5=[rb*cos(Ipsilon5);rb*sin(Ipsilon5);0];
B2=[rb*cos(Ipsilon2);rb*sin(Ipsilon2);0];
B4=[rb*cos(Ipsilon4);rb*sin(Ipsilon4);0];
B6=[rb*cos(Ipsilon6);rb*sin(Ipsilon6);0];

% Rotation matrix elements
a = roll;  % alpha
b = pitch; % beta
g = yaw;   % gamma
r11=cos(b)*cos(g);
r12=cos(g)*sin(a)*sin(b)-cos(a)*sin(g);
r13=sin(a)*sin(g)+cos(a)*cos(g)*sin(b);
r21=cos(b)*sin(g);
r22=cos(a)*cos(g)+sin(a)*sin(b)*sin(g);
r23=cos(a)*sin(b)*sin(g)-cos(g)*sin(a);
r31=-sin(b);
r32=cos(b)*sin(a);
r33=cos(a)*cos(b);

% Rotation matrix
Rxyz=[r11 r12 r13;r21 r22 r23;r31 r32 r33];

% translation vector P0
P0 = [x; y; z]';

% positions of points T
T1=P0+(Rxyz*GT1)';
T2=P0+(Rxyz*GT2)';
T3=P0+(Rxyz*GT3)';
T4=P0+(Rxyz*GT4)';
T5=P0+(Rxyz*GT5)';
T6=P0+(Rxyz*GT6)';

% actuator vectors L
L1 = T1 - B1';
```

```
L2 = T2 - B2';
L3 = T3 - B3';
L4 = T4 - B4';
L5 = T5 - B5';
L6 = T6 - B6';

% lengths (norms) of vectors
l1 = sqrt(L1(1)*L1(1)+L1(2)*L1(2)+L1(3)*L1(3));
l2 = sqrt(L2(1)*L2(1)+L2(2)*L2(2)+L2(3)*L2(3));
l3 = sqrt(L3(1)*L3(1)+L3(2)*L3(2)+L3(3)*L3(3));
l4 = sqrt(L4(1)*L4(1)+L4(2)*L4(2)+L4(3)*L4(3));
l5 = sqrt(L5(1)*L5(1)+L5(2)*L5(2)+L5(3)*L5(3));
l6 = sqrt(L6(1)*L6(1)+L6(2)*L6(2)+L6(3)*L6(3));
end
```
Listing B.1: MatLab function for the calculation of the inverse kinematics.