Universidade de Aveiro Departamento de
2011 Electrónica, Telecomunicações e Informática,

Ricardo Jorge
de Sousa Rodrigues

Controlo distribuído de um braço robótico

Universidade de Aveiro
2011

Departamento de
Electrónica, Telecomunicações e Informática,

Ricardo Jorge
de Sousa Rodrigues

Controlo distribuído de um braço robótico

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica de Prof. Paulo Bacelar Reis Pedreiras e Prof. Filipe Miguel Teixeira Pereira da Silva, Professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Este trabalho é financiado por Fundos FEDER através do Programa Operacional Factores de Competitividade – COMPETE e por Fundos Nacionais através da FCT – Fundação para a Ciência e a Tecnologia no âmbito do projecto "HaRTES: Hard Real-Time Ethernet Switching"(PTDC/EEA-ACR/73307/2006);

o júri / the jury

presidente / president

Professor Doutor José Alberto Gouveia Fonseca
Professor Associado da Universidade de Aveiro

vogais / examiners committee

Professora Doutora Ana Luisa Lopes Antunes
Professora Adjunta do Departamento de Engenharia Eletrotécnica da Escola Superior de Tecnologia de Setúbal do Instituto Politécnico de Setúba (arguente principal)

Professor Doutor Paulo Bacelar Reis Pedreiras
Professor Auxiliar da Universidade de Aveiro (orientador)

Professor Doutor Filipe Miguel Teixeira Pereira da Silva
Professor Auxiliar da Universidade de Aveiro (coorientador)

agradecimentos / acknowledgements

Em primeiro lugar tenho que agradecer aos meus pais e ao meu irmão pelo apoio incondicional e por acreditarem sempre em mim. Eles são os principais responsáveis por eu ter chegado onde cheguei.

Agradeço aos meus orientadores, Paulo Pedreiras e Filipe Silva, que me ajudaram imenso durante esta dissertação e se mostraram sempre disponíveis para tal.

Agradeço ao pessoal da ESN, principalmente ao Vilaça e ao Chaves, que me deram ânimo para acabar esta dissertação.

Agradeço aos meus treinadores de ciclismo (família Carvalho) e aos meus colegas de equipa com quem aprendi imenso.

Agradeço ainda aos meus amigos de Erasmus, tanto aos da Polónia como aos de Aveiro, que me fizeram crescer a nível social.

A todas estas pessoas, o meu muito obrigado.

Palavras-Chave FTT-SE, Cinemática, Controlo, RTOS, Braço Robótico, RTLinux

Resumo

Muitos dos sistemas electrónicos de hoje em dia necessitam de executar tarefas com determinados requisitos de pontualidade, previsibilidade ou relações de precedência. O cumprimento destes leva muitas vezes à necessidade de utilização de sistemas operativos de tempo-real (RTOS), devidamente complementados por redes de tempo-real, no caso de sistemas distribuídos.

No âmbito desta dissertação desenvolveram-se estruturas básicas para um demonstrador do protocolo FTT-SE, que oferece comunicação tempo-real sobre Ethernet. Este demonstrador é baseado num braço robótico com cinco graus de liberdade, o que corresponde a cinco juntas ou eixos, estando cada uma destas ligada a um computador que a controla.

Foram desenvolvidos *device drivers* para o RTOS RTLinux para efectuar a comunicação com o *hardware* desenvolvido que permite obter informação sobre a posição da junta do braço robótico bem como controlar o movimento do motor. Foram desenvolvidos também algoritmos de controlo e cinemática com o objectivo de controlar o movimento de cada junta do braço de forma a que esta siga uma trajectória previamente definida.

Keywords FTT-SE, Kinematics, Control, RTOS, Robotic Arm, RTLinux

Abstract

Many of the electronic systems of today need to perform tasks with certain requirements of timeliness, predictability or precedence relations. The fulfilment of this requirements often leads to the need to use real-time operating systems (RTOS), fully complemented by real-time networks, in case of distributed systems.

In this dissertation it was developed basic structures for a FTT-SE protocol demonstrator, which provides real-time communication over Ethernet. This demonstrator is based on a robotic arm with five degrees of freedom, corresponding to five joints or axes, with each one of these connected to a computer that controls it.

We developed device drivers for the RTOS RTLinux to allow the communication with the hardware developed that provides the joint position of the robotic arm and control the movement of its engine. In addition, kinematics and control algorithms were developed in order to control the movement of each joint of the arm so that it follows a predefined trajectory.

Contents

Contents **i**

List of Figures **iii**

List of Tables **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Objectives . . . 2
1.3 Document Structure . . . 3

2 Fundamental concepts **5**
2.1 Real-time systems . . . 5
2.1.1 Types of real-time tasks concerning time constraints . . . 5
2.1.2 Types of real-time tasks concerning its periodicity . . . 6
2.1.3 Basic concepts . . . 6
2.1.4 Scheduling algorithms . . . 7
2.1.5 RTLinux . . . 8
2.2 Kinematics . . . 9
2.2.1 Forward kinematics . . . 10
2.2.2 Inverse kinematics . . . 11
2.3 Trajectory planning . . . 12
2.3.1 Polynomial trajectory . . . 13
2.3.2 Point-to-point motion . . . 13
2.3.3 Continuous motion . . . 13

3 System architecture **15**
3.1 Hardware . . . 15
3.2 Software . . . 16
3.2.1 Device Drivers . . . 17
3.2.2 Filters . . . 19
3.2.3 Internal scheduler . . . 19
3.2.4 Trajectory planning . . . 20
3.2.5 Control systems . . . 20
3.3 Experimental assessment . . . 21
3.3.1 Time measurements . . . 21
3.3.2 Filters . . . 27

3.3.3 Internal Scheduler 28

4 Control System **31**
4.1 Proportional controller 32
4.2 Proportional-Integral Controller 36
4.3 PID Controller 37
4.4 Cascade controller 39
4.5 Motor Linearity 41

5 Network sniffer **43**
5.1 Configuration 43
5.2 Implementation 44
5.3 Test 45

6 Conclusions and Future Work **47**

Bibliography **48**

Appendix A **50**
6.3 Circuit 50

Apendix B **51**
6.5 Acronyms 51

List of Figures

1.1 System architecture diagram 2

2.1 Utility of the results produce by a firm real-time task 6
2.2 Utility of the results produce by a soft real-time task 6
2.3 Priority assignment to tasks in RM 8
2.4 RTLinux structure 9
2.5 RTLinux task structure 9
2.6 Schematic of the robotic arm 10
2.7 Point-to-point motion 13
2.8 Simple continuous motion solution 14

3.1 Blocks diagram of the hardware architecture 15
3.2 Shared Memory Diagram 16
3.3 Device driver flow charts 18
3.4 Internal scheduler 19
3.5 Internal scheduler Flow chart 20
3.6 Control systems task 21
3.7 Time needed to read the position, not optimized 22
3.8 ADC reading process, Optimized 23
3.9 Time needed to read the position, optimized 24
3.10 Time needed to read the current 25
3.11 Time needed to update the PWM 26
3.12 signal from the position, without filter and engine stopped 27
3.13 signal from the position, with filter and engine stopped 28
3.14 Internal scheduler 29

4.1 General controller model 31
4.2 PID Contoller 33
4.3 Proportional Control response, Position x PWM 33
4.4 Proportional Control response, Real Position x Reference 34
4.5 Proportional Control response, with offset, Position x PWM 35
4.6 Proportional Control response, with offset, Real Position x Reference 35
4.7 PI Controller response, Position x PWM 36
4.8 PI Controller response, Real Position x Reference 37
4.9 PID Controller response, Position x PWM 38
4.10 PID Controller response, Real Position x Reference 39
4.11 P plus PI in cascade controller 39

4.12 Cascade Controller response, Position x PWM 40
4.13 Cascade Controller response, Real Position x Reference 41
4.14 Non-linearity of the engine, against the gravity 42
4.15 Non-linearity of the engine, in favour of the gravity 42

5.1 Configuration window 43
5.2 Application flowcharts 44
5.3 Screen capture of the network sniffer working 45
5.4 Screen capture of the configuration for the test 46
5.5 Screen capture of the first test 46
5.6 Screen capture of the second test 46

List of Tables

2.1 Kinematics parameters table 10

3.1 Shared memory 17
3.2 Table of the time needed to read the position, not optimized 22
3.3 Table of the time needed to read the position, optimized 23
3.4 Table of the time needed to read the current 24
3.5 Table of the time needed to update PWM 25
3.6 Worst case scenario for each task 26
3.7 Table with statistic values from the signal from the position, without filter . . 27
3.8 Table with statistic values from the signal from the position, with filter noise 28
3.9 Statistic values from position signal, with and without internal scheduler . . . 29

4.1 Table with statistic analyse from the proportional control 32
4.2 Table with statistic analyse from the proportional control, with offset 34
4.3 Table with statistic analyse from the proportional-Integral control, with offset 36
4.4 Effect cause by each parameter of the PID 38
4.5 Table with statistic analyse from the PID controller 38
4.6 Table with statistic analyse from the cascade controller 40

Chapter 1

Introduction

This dissertation is integrated on the FCT funded project named HaRTES [6] that was already in development. In the scope of this project, the hardware interface between the computer and the robotic arm was already developed. Also a preliminary implementation of the device-drivers was made available. The project is built upon recent work on the FTT (Flexible Time-Triggered) [5] communication paradigm to develop Ethernet switches with enhanced transmission control, traffic scheduling, service differentiation, transparent integration of non-real-time nodes and improved error confinement mechanisms, particularly with respect to temporal misbehaviours.

1.1 Motivation

Although there is a great diversity of network protocols, most of them fail in flexibility and don't provide admission control mechanisms and management of quality of service (QoS) that would allow to dynamically change the operation of the system. Currently these features are being seen as essential in some electronic systems that require an adaptive behaviour changing the mode of operation during run time.

The FTT-SE protocol was created to answer this need and therefore it was developed a switch based on it to test its effectiveness. This switch is capable of produce enough flexibility and management of the resources efficiently without compromise the timeliness required by real-time applications. To test the capabilities of the switch, it was developed a challenging demonstrator, which is a robotic arm, for which some modules have been developed in the scope of this dissertation.

In this dissertation we had to build basic structures for a demonstrator based on a robotic arm that would be used to test the FTT-SE protocol, so we developed a system that would control a joint of the robotic arm. This system, as shown in figure 1.1, is composed by different blocks connected to each other:

- **Computer** runs the software (device drivers and control systems) and physically uses a parallel port to allow the communications with other blocks.
  - **Control Systems** is responsible for the movement control of the arm joint, which means that giving a previous trajectory for the joint, the control systems would ensure the realisation of the trajectory. The control systems use the device drivers to communicate with the HW interface board (Hardware interface board) in order

  to read the actual position and the H-bridge to control the power given to the electrical motor.
  - **Device driver** makes possible the communication between the control systems and the rest of the system. It is composed by low level tasks that read the position and current and generates the PWM signal fed to the H-bridge.

- **HW interface Board** (hardware interface board) is a board designed to read data from the position sensor and the current consumed by the electrical motor. The current is measured from the H-bridge using a sensing resistor.

- **H-Bridge** is a board that allows the system to change the direction of the movement of the arm joint and also has a sensor that is used by the HW interface board to measure the current.

- In the **Arm Joint**, there is a position sensor and a electrical motor that provides the movement to it.
  - The **Sensor** is a simple potentiometer in which the voltage measured on it represents the position of the joint.
  - The **electrical motor** is controlled by the control systems using a PWM signal that represents the power given to it.

The work done in this dissertation will be used to make a distributed control of the robotic manipulator which has strict timing requirements in order to work properly. Therefore it provides a good visual way to observe the effectiveness of the switch. This dissertation required a wide knowledge, not only of electronics, control systems but also real-time systems and kinematics.

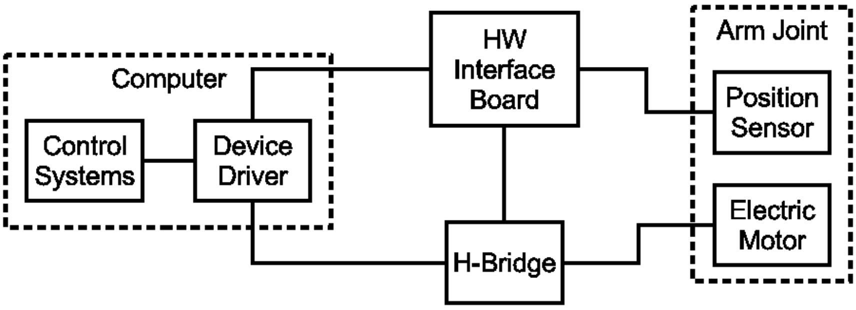


Figure 1.1: System architecture diagram

1.2 Objectives

As said before, the work done in this dissertation envisions the implementation of a system which would make possible to test a switch created on the scope of the project HaRTES. To achieve the main goal, we would have to pass on some steps:

- Become familiar with concepts of real-time (industrial communications, operating systems real-time)
- Complete the electronic interface of control of the robotic arm
- Develop the "device-drivers"
- Develop of control and motion planning software
- Develop of demo applications
- Evaluate the impact of the communications on the performance of the robotic arm

From all of these objectives the only one that was not achieved was the evaluation of the impact of the communication on the performance of the robotic arm. This would need to have the distributed control implemented which didn't happen.

1.3 Document Structure

This thesis is divided in five main chapters:

- **Fundamental concepts** - Here it is explained some basic concepts and terminology that are necessary to understand the work done
- **System architecture** - It has an introduction to the system developed (hardware and software) where it is explained how the system works. Some tests are presented, more precisely, time tests and filter results.
- **Control System** - This is where it is described the methods used to control the joint of the robotic arm. There is a statistic interpretation of the response of the joint to each controller used.
- **Network sniffer** - Here it is introduced a software that we developed in the beginning of this dissertation, which can be used to observe a set of streams in a network and graphically showing its transmission rate during a window of time.
- **Conclusions** - The last chapter summarizes the main results achieved during the dissertation. There is an evaluation of work done and also some remarks concerning the future work that can be done.

Chapter 2

Fundamental concepts

2.1 Real-time systems

"Real-time systems are computing systems that must react within precise time constraints to events in the environment" [17]. These temporal requirements usually arise from the dynamics of the process under control. These conditions have to be always fulfilled, even in the worst case.

It is typical to use real time systems in critical applications in terms of economic or security (e.g. nuclear power plants, rail traffic control and aviation, etc.).

It is normal to consider safety and reliability to be independent issues, nevertheless, in real-time systems, safety and reliability are coupled together. When there is failure in the system and there is no damages resulting, we can say that it has a fail-safe state. But if the system failure cause severe damages we say that we are facing a safety-critical system. To achieve high reliability the software development should have in attention fault tolerance, error avoidance, error detection and removal, built in self test (BIST) and redundancy.

2.1.1 Types of real-time tasks concerning time constraints

Real-time tasks have some kind of time constraints, and concerning this characteristic they can be classified into either hard, soft or firm real-time tasks depending on the consequences of a task missing its deadline.

A **Hard real-time** task is a task in which missing a deadline may cause serious problems in the system. When any of its hard real-time tasks doesn't successfully complete its required results within the time bound, the system is considered to have failed.

Firm real-time tasks are tasks in which its failure affects the system but not in a catastrophic way. In case the firm real-time task does not complete within its deadline, the system does not fail, it just discards the late results. An example of a firm real-time task is a video conference.

Soft real-time tasks, unlike the previous ones, don't cause immediate problems to the system, but after a while it can cause system failure. The control of the temperature in a room can be an example.

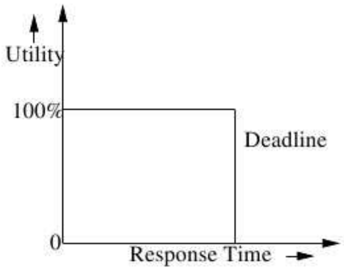


Figure 2.1: Utility of the results produce by a firm real-time task

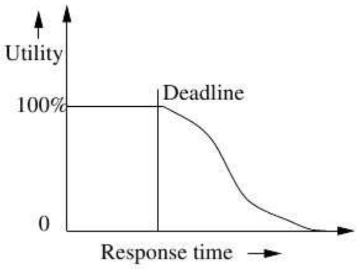


Figure 2.2: Utility of the results produce by a soft real-time task

2.1.2 Types of real-time tasks concerning its periodicity

A real-time tasks can be classify into three main categories in terms of its periodicity: periodic, sporadic and aperiodic.

A **periodic** task is one that recur with a fixed time interval, being the precise time instants at which it repeats usually demarcated by clock interrupts. Due to that, periodic tasks may be referred to as clock-driven tasks. The fixed time interval is called the *period* of the task.

A **sporadic** task is one that repeats at random instants. A sporadic task T_i can be mathematically represented by:

$$T_i = (e_i, g_i, d_i)$$

where e_i is the worst case execution time of an instance of the task, g_i denotes the minimum separation between two consecutive instances of the task, d_i is the relative deadline. We can say that once an instance of a sporadic task occurs, the next instance cannot occur before g_i time units have elapsed. That means, g_i restricts the rate at which the sporadic tasks can arise.

An **aperiodic** task can also occur at random instants. However, in this case, the minimum separation g_i between two consecutive instances can be 0. That means that two or more instance of an aperiodic task might happen at the same time.

2.1.3 Basic concepts

These are some basic concepts that will be used later:

Preemtive Scheduler is a scheduler in which when a higher priority arrives, suspends any lower priority task currently being executed and the higher priority task takes up for execution. The lower priority task can only resume when there are no higher priority tasks ready.

Utilization of a task is the average time for which it executes per unit time interval. For a periodic task T_i its utilization is $u_i = \dfrac{e_i}{p_i}$, where e_i is the execution time and p_i is the period of T_i. For a set of n periodic tasks $\{T_i\}$, the utilization due to all tasks is given by $U = \sum_{i=1}^{n} \dfrac{e_i}{p_i}$.

2.1.4 Scheduling algorithms

In real-time systems the tasks are performed in a particular order. The instances, ready to run, wait in a queue sorted according to certain scheduling criteria. The scheduler (function that will sort the queue of ready tasks) should use a deterministic criterion in order to allow calculating the maximum delay (worst case) that a task can suffer in the queue.

Each scheduler is characterized by its algorithm. There is a large number of scheduling algorithms for real-time systems so far, some of them are: Fixed priority pre-emptive scheduling [4], Fixed-Priority Scheduling with Deferred Preemption, Fixed-Priority Non-preemptive Scheduling, Round-robin scheduling [12], Critical section preemptive scheduling, Static time scheduling, Earliest Deadline First (EDF) [3], Rate Monotonic (RM) [11] and advanced scheduling using the stochastic and MTG.Due to its relevance, EDF and RM are now presented with some detail.

Earliest Deadline First

In **Earliest Deadline First (EDF)** scheduling, the order is based on the task deadline. In this algorithm the task having the shortest deadline is taken up for scheduling. The task set is schedulable, if and only if it satisfies the necessary condition that the total processor utilization caused by the task set is less then 1. This condition [19] can be expressed as:

$$\sum_{i=1}^{n} \frac{e_i}{p_i} = \sum_{i=1}^{n} u_i \leqslant 1 \tag{2.1}$$

where u_i is the average utilization due to the task t_i and n is the total number of tasks in the task set.

Rate Monotonic

In the Rate Monotonic (RM) scheduling algorithm the priority assigned to a task is based their request rates. It means that tasks with higher request rates will have higher priorities. Since periods are constant, in RM the priorities are fixed. A priority P_i is assigned to the task before its execution and does not change over the time. We can even say that RM is intrinsically preemptive: the currently executing task is preempted by a newly arrived task with shorter period.

The priority of a task T_i is directly proportional to its rate (or inversely proportional to its period) and can be represented as: $Priority = \dfrac{k}{p_i}$, where p_i is the period of the task T_i and

k is a constant. Using this expression we can plot the priority of tasks for different periods as shown in the figure 2.3(a) and 2.3(b).

A set of periodic real-time tasks are not guaranteed to be schedulable in RM unless they satisfy the **sufficient condition** [19] represented in the equation 2.2, where u_i is the average utilization due to the task t_i and n is the total number of tasks in the task set.

$$\sum_{i=1}^{n} \frac{e_i}{p_i} = \sum_{i=1}^{n} u_i < n(2^{1/n} - 1) \tag{2.2}$$

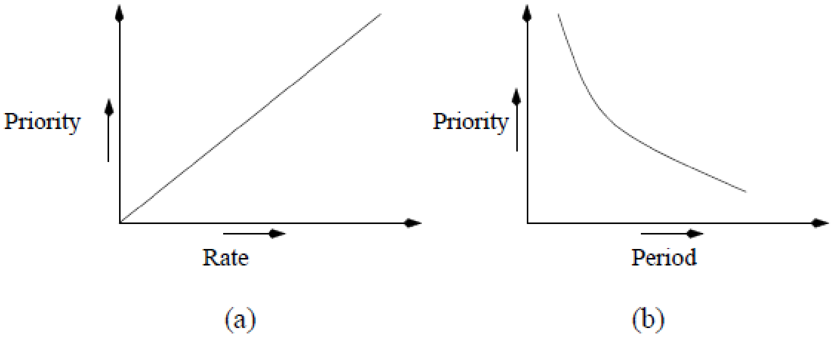


Figure 2.3: Priority assignment to tasks in RM

2.1.5 RTLinux

Schedulers are part of the operating system kernel. Some of these kernels are: ReTMiK [10], OReK, RTKPIC18 [14], SHaRK [16], RTAI [13] and RTLinux [15].

Real-Time Linux (RTLinux) is a version of Linux that provides hard real time capability. RTLinux provides the capability of running special realtime tasks and interrupt handlers on the same machine as standard Linux. These tasks and handlers execute when they need to execute no matter what Linux is doing. The operating system, as shown in the figure 2.4, can be divided in two parts: Kernel space and User space. Kernel space has the core of the system, here you have the low level tasks, such as device-drivers, operating system modules, etc. User space is where we have all the high level programs running. The user space can't communicate directly with the hardware: it must be done through the kernel space. These two spaces need a special API to be able to communicate with each other. In the RTLinux this API can be a FIFO or shared memory.

In RTLinux the tasks have a specific structure as shown in the figure 2.5. There is an initial configuration in the beginning of the Thread (programming wise, not during the run time) with parameters related to itself (priority, etc.) and in the end we have to include a sleep function that represents the period of the task. In RTLinux if we want periodic tasks we need to calculate the time to sleep in each tick, taking into account the execution time.

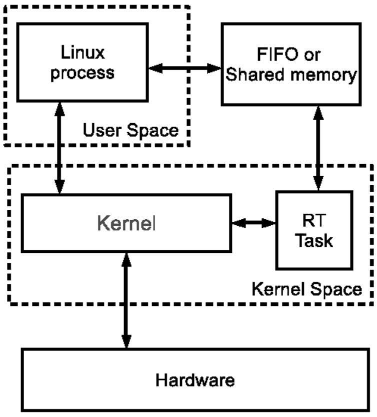


Figure 2.4: RTLinux structure

```
void *taskName(void *t) {
        // task configuration
        // variable initialization
        // resource allocation
        while(1){

                //task body

                rtl_clock_nanosleep(RTL_CLOCK_REALTIME, RTL_TIMER_ABSTIME, &time, NULL);
                //RTL_CLOCK_REALTIME and RTL_TIMER_ABSTIME are constants defined by the RTLinux
                //time is the time interval that the task will sleep
        }
}
```

Figure 2.5: RTLinux task structure

2.2 Kinematics

Kinematics is the science of *geometry in motion* [18]. It describes the motion as a function of the position, orientation, and time derivatives. It has an important role in this thesis since it describes the motion of the system and the relation between the various axis. In Kinematics there are two important equations: Forward kinematics and Inverse kinematics.

Forward kinematics is a computation of the position and orientation of robot's end effector (or end frame) as a function of its joint angles. It means that we can obtain the Cartesian coordinates through the joint angles. The reverse process, obtain the joint angles through the Cartesian coordinates, is called **inverse kinematics**.

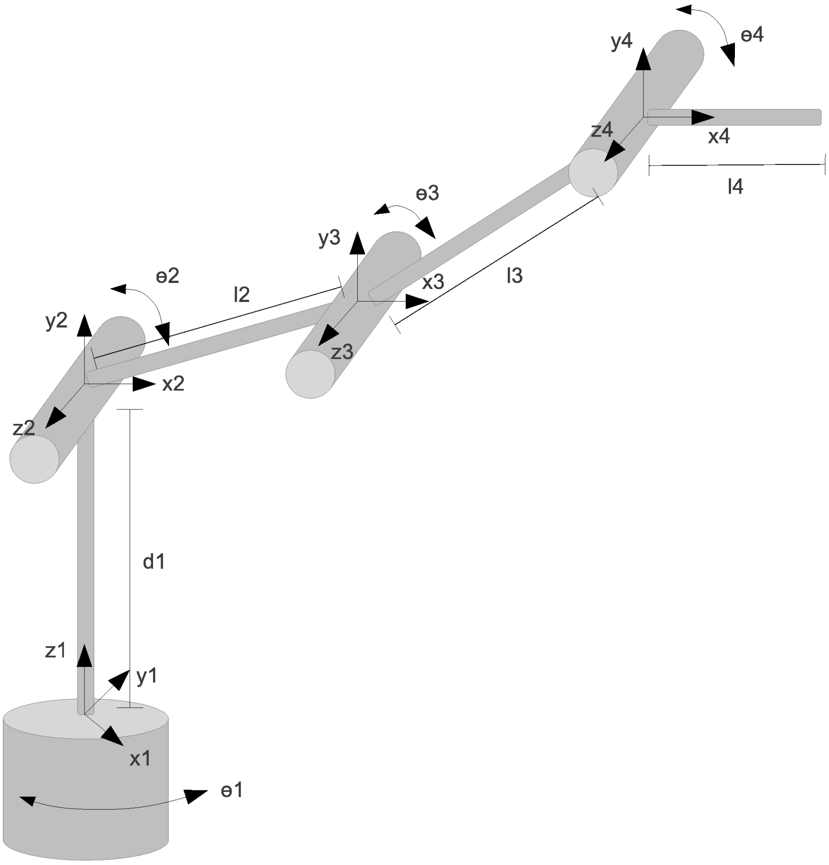


Figure 2.6: Schematic of the robotic arm

2.2.1 Forward kinematics

To find these equations we had to use the Denavit-Hartenberg (DH) convention. Firstly we defined the references for each joint, in our case we used the ones represented in figure 2.6. In the DH convention the Z-axis is in the direction of the joint axis, the X-axis is parallel to the common normal [2] and the Y-axis is the one left using the right-handed coordinate system. Then, based on these references, we can obtain the kinematics parameters represented in table 2.1, where θ_i is the joint angle, l_i is the link length, d_i is the link offset (axis 1 case) and α_i is the link twist.

In the DH convention, each one of the joints can be represented as an homogeneous trans-

i	θ_i	l_i	d_i	α_i
1	$\theta 1$	0	d_1	90^o
2	$\theta 2 + 90^o$	l_2	0	0
3	$\theta 3$	l_3	0	0
4	$\theta 4$	l_4	0	0

Table 2.1: Kinematics parameters table

formation (A_i) and it is defined as the equation 2.3 where R is the 3×3 submatrix describing rotation and T is the 3×1 submatrix describing translation. In that equation "c" is a short for cosine (cos) and "s" is the short for sine (sin) trigonometric function.

$$A_i = \begin{bmatrix} c\theta_i & -s\theta_i c\alpha_i & s\theta_i s\alpha_i & l_i c\theta_i \\ s\theta_i & c\theta_i c\alpha_i & -c\theta_i s\alpha_i & l_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} = \left[\begin{array}{ccc|c} & R & & T \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \tag{2.3}$$

If we replace the parameter in this equation (2.3) by the ones in the kinematics parameters table (2.1) we obtain the homogeneous transformation matrix for each joint, as shown in matrices 2.4.

$$A_1 = \begin{bmatrix} c1 & 0 & s1 & 0 \\ s1 & 0 & -c1 & 0 \\ 0 & 1 & 0 & d1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_2 = \begin{bmatrix} -s2 & -c2 & 0 & -l2 * s2 \\ c2 & -s2 & 0 & l2 * c2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} c3 & -s3 & 0 & l3 * c3 \\ s3 & c3 & 0 & l3 * s3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_4 = \begin{bmatrix} c4 & -s4 & 0 & l4 * c4 \\ s4 & c4 & 0 & l4 * s4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

These matrices give us the transformation in each joint, if we want the transformation matrices from the base to a specific joint we need to multiply every transformation matrices between them. For instance the transformation matrix from the base to the end frame M_{04} (that will be used to obtain the forward kinematic equations) we have to multiply the matrices A_1 A_2 A_3 and A_4, as it is represented in the equation 2.5, where $c234$ is the short for $cos(\theta_2 + \theta_3 + \theta_4)$ and the same for $s234$ that is the short for $sin(\theta_2 + \theta_3 + \theta_4)$.

$$M_{04} = A_1 \times A_2 \times A_3 \times A_4 = \begin{bmatrix} -s234 * c1 & -c234 * c1 & s1 & -c1 * (l3 * s23 + l2 * s2 + l4 * s234) \\ -s234 * s1 & -c234 * s1 & -c1 & -s1 * (l3 * s23 + l2 * s2 + l4 * s234) \\ c234 & -s234 & 0 & d1 + l3 * c23 + l2 * c2 + l4 * c234 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Finally, what we'll need from here is the Translation matrix T_{04} that will gives the equations that we need to calculate the Cartesian coordinates (x_e,y_e,z_e) through the joint angles and represented in the equation 2.6.

$$\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix} = T_{04} = \begin{bmatrix} -c1 * (l3 * s23 + l2 * s2 + l4 * s234) \\ -s1 * (l3 * s23 + l2 * s2 + l4 * s234) \\ d1 + l3 * c23 + l2 * c2 + l4 * c234 \end{bmatrix} \tag{2.6}$$

2.2.2 Inverse kinematics

After having found the forward kinematics equations it is easy to find the inverse kinematics ones. For that we need to mathematically solve a linear system with the three forward kinematics equations in function of the joint angles. However, as you probably realised, there are four variables and only three equations, so we have to replace one of the variables for something else and leaving it to be calculated separately. In our case we replaced the θ_4 by the equation 2.7, where β is the angle between the horizontal and the end frame.

$$\theta_4 = \beta - 90^o - \theta_2 - \theta_3 \tag{2.7}$$

After solving the linear system 2.8 and replacing $\theta_4$in it by the equation 2.7, we can obtain the inverse kinematics equations 2.9, 2.10, 2.11 and 2.12.

$$\begin{cases} x_e = -c1 * (l3 * s23 + l2 * s2 + l4 * s234) \\ y_e = -s1 * (l3 * s23 + l2 * s2 + l4 * s234) \\ z_e = d1 + l3 * c23 + l2 * c2 + l4 * c234 \end{cases} \tag{2.8}$$

$$\Theta_1 = tg^{-1}\left(\frac{y_e}{x_e}\right) \tag{2.9}$$

$$\begin{aligned} &\Theta_3 = \pm tg^{-1}\left(\frac{\sqrt{{k_2}^2 - {k_3}^2}}{k_3}\right) \\ &k_2 = 2 * l2 * l3 \\ &k_3 = \left(\frac{x_e}{c1} - l4 * c\alpha\right)^2 + (z_e - d1 - l4 * s\alpha)^2 - l3^2 - l2^2 \end{aligned} \tag{2.10}$$

$$\begin{aligned} &\Theta_2 + \Theta_3 = tg^{-1}\left(\frac{k_1}{k_2}\right) \pm tg^{-1}\left(\frac{\sqrt{k_1^2 + k_2^2 - k_3^2}}{k_3}\right) \\ &k_1 = -2 * l3 * \left(\frac{x_e}{c1} - l4 * c\alpha\right) \\ &k_2 = 2 * l3 * (z_e - d1 - la * s\alpha) \\ &k_3 = \left(\frac{x_e}{c1} - l4 * c\alpha\right)^2 + (z_e - d1 - la * s\alpha)^2 + l3^2 - l2^2 \end{aligned} \tag{2.11}$$

$$\Theta_4 = \beta - 90^o - \Theta_2 - \Theta_3 \tag{2.12}$$

2.3 Trajectory planning

Trajectory planning is a part of control, in which we plan a trajectory to be followed by the manipulator in a planned time profile. The trajectory can be planned in joint or Cartesian space. Joint trajectory planning directly specifies the time evolution of the joint variables. Cartesian trajectory specifies the position and orientation of the end frame during a period of time. It may include avoiding obstacles.

2.3.1 Polynomial trajectory

To obtain smooth paths, the trajectory planner must be a continuous function, with a continuous first derivative and hopefully also a continuous second derivative. There are several ways to calculate the path, one of them and probably the simplest is the polynomial path. In our case we will use the fifth degree polynomial, also known as *Quintic path*, that specifies the position, velocity and acceleration. The Quintic path is defined in equations 2.13 and 2.14, the first one giving the position and the last one the velocity.

$$a_3 = \frac{-10 * (\theta_0 - \theta_f)}{t_f^3}, a_4 = \frac{15 * (\theta_0 - \theta_f)}{t_f^4}, a_5 = \frac{-6 * (\theta_0 - \theta_f)}{t_f^5}$$

$$p(t) = a_5 t^5 + a_4 t^4 + a_3 t^3 + \theta_0 \tag{2.13}$$

$$\dot{p}(t) = 5a_5 t^4 + 4a_4 t^3 + 3a_3 t^2 \tag{2.14}$$

2.3.2 Point-to-point motion

There are a few ways to describe the motion of the path. The most important ones are the *Point-to-point motion* and the *Continuous motion.* The first one is the simplest and it plans to move the end frame from a point to another within a specific given time, without any concern about the intermediate points. An implementation of this is represented in figure 2.7, where r_f is the final destination in Cartesian coordinates, the q_f is the final destination in joint angles and $q(t)$, $\dot{q}(t)$ and $\ddot{q}(t)$ are the position, velocity and acceleration for each moment of the movement. This technique uses the inverse kinematics to convert from final destination in Cartesian coordinates (that we Humans are more used to work with) to joint angles, then knowing the actual position in joint angles, the final destination and the time in which we want to the movement be complete, the path planing can plan the whole movement giving the position, velocity and acceleration for each moment (in our case we used the quintic path). This operation can be computed offline, which means that can be computed before the movement starts. Finally with the result from the path planing, the operation rests in the control systems that will take care of the obedience to that planning.

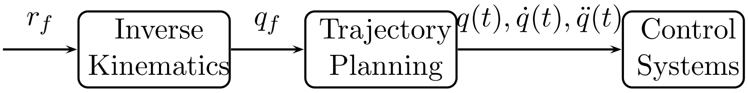
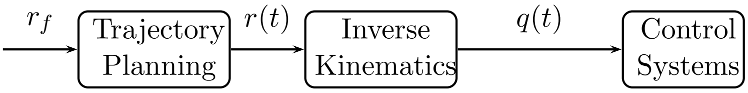


Figure 2.7: Point-to-point motion

2.3.3 Continuous motion

In the continuous motion, the end frame is moved through a geometric specified path within a predefined space of time. This one is really useful when the movement made by the manipulator cares about the way to go from a point to another and enables more complex tasks e.g. car painting, to make a geometric movement (line, curve, etc.).

There are several ways to implement this type of motion. One of them is illustrated in figure 2.8, where r_f means the final position in Cartesian coordinates, $r(t)$ is the position along the time and $q(t)$ is the position in joint angles along the time. In this case the path planing is done on the Cartesian space and then for each moment the position is converted to joint angles using the inverse kinematics. This two operations can be done separately and offline. Finally as in the point-to-point case, the control systems takes care of the rest.



r_f
Trajectory Planning
$r(t)$
Inverse Kinematics
$q(t)$
Control Systems

Figure 2.8: Simple continuous motion solution

Chapter 3

System architecture

This project has been worked on before I started, therefore it was partially defined, mainly the hardware part. The aim was to find a simple and reliable solution, so basically it is composed by a manipulator, a power supply, a H-bridge and a circuit board that takes care of the data acquisition, which we called *HW interface board*. In the computer there is a software running as a real-time task in a real-time operative system (RTLinux). The whole system is explained in detail in the following sections.

3.1 Hardware

The demonstrator design has been initiated before we started this dissertation, so we just did some minor changes and developed the printed circuit board (PCB) version. As said before, in this system we have a manipulator with five joints. However we just use four of them, since the end frame joint only allow us to rotate itself which is not useful for the purpose of this demonstrator. In each joint there is a set of modules integrated, as depicted in figure 3.1. The DAQ is used to read digital signals into the computer, allowing it to physically communicate with the rest of the modules.

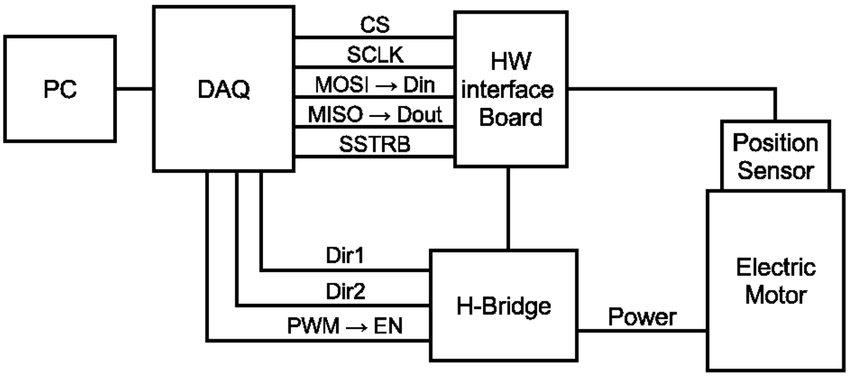


Figure 3.1: Blocks diagram of the hardware architecture

In the HW interface board, as you can see in section 6.3 of the appendix, we have an ADC (max186), one voltage regulator of 5V (7805) to power the ADC and some connectors. The ADC reads the joint position in Volts through a sensor that exists on it and also reads the instantaneous current consumed by the motor using a sensing resistor placed in the H-Bridge. The current is measured for security matters allowing to avoid damage caused by blockage on the movement. The ADC uses the SPI standard to communicate with the DAQ and therefore with the computer, so there are four logic signals that are usually used in this cases: Serial Clock (SCLK), Master Output Slave Input (MOSI), Master Input Slave Output (MISO) and slave select (SS). In our case, the ADC equivalent signal would be (respectively): $SCLK$, D_{in}, D_{out}, $\overline{CS}$ [1]. There is also a strobe signal (SSTRB) that is used to verify the state of the ADC.

The H-bridge is an element used to control the direction of the joint movement and the power given to its motor. One input is fed with a PWM signal to control the power delivered to the motor. There are two more inputs aiming to select the direction of the movement.

3.2 Software

The software is divided in two major parts: device drivers and the control systems and kinematics. These two parts were placed in two different spaces of the operative system: Kernel space and User space. Kernel space is where the core of the system is, in our case the device drivers. User space is where all the high level application components are and in our case we'll have the control systems and kinematics. These two spaces need to communicate with each other. RTLinux supports FIFO queue and shared memory as a way of communication. We decide to use shared memory, as shown in figure 3.2, because the variables needed to be shared between various modules which would make the FIFO inefficient since multiple FIFOs would be needed. in addition, the variables were considered state variables, it represents a state instead of an event trigger, which make them valid at every instants. We reserved two addresses in the shred memory: rAddr and wAddr. The rAddr is reserved for communication from the kernel into the user space and the wAddr the opposite. They were configured as described in the table 3.1, where $pos[]$ is the actual position divided in two bytes, vel[] is the velocity, Error is a error byte identifying what error happened, dir is the direction select variable, stop/move is a byte that orders the movement to be stop or to start and NU is not used.

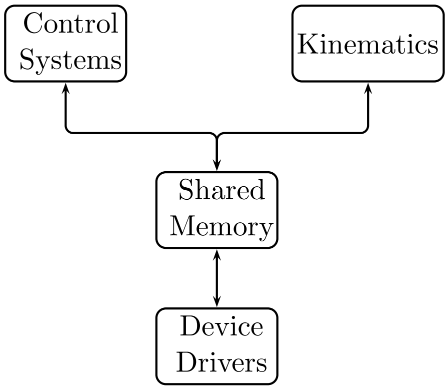


Figure 3.2: Shared Memory Diagram

Byte number	0	1	2	3	4
rAddr	pos[0]	pos[1]	vel[0]	vel[1]	Error
wAddr	PWM	dir	Stop/move	NU	NU

Table 3.1: Shared memory

3.2.1 Device Drivers

The first step taken in the software side was the development of the device drivers, that would have the capacity to read the joint position, the instant current consumed by the engine of the same joint and to apply a PWM signal in the H-Bridge allowing the software to change the power given to the electrical motor and therefore the joint velocity. The code was running in kernel space using real-time services and divided in tasks, one for each of the following cases. As explained further on, this software organization was changed during the course of the work due to signal acquisition constraints.

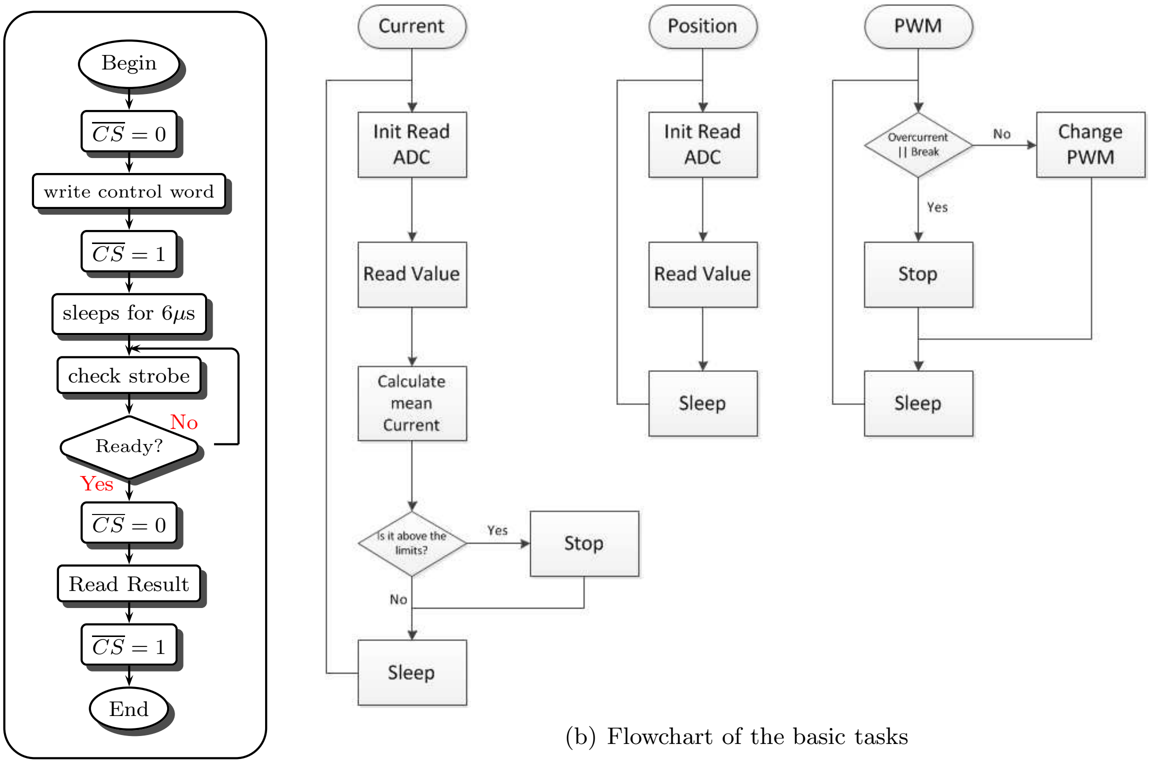
Some of the tasks in the device drivers will use the ADC to read values. To read a value from the ADC we need to follow the sequence represented in figure 3.3(a). It initiates by putting the $\overline{CS}$ at 0, then we have to write a control word which selects, among others parameters, the channel that the ADC will use to convert. In the position case we use the channel 1 ($sel = 100$) [1] and in the current case we use the channel 0 ($sel = 000$). Then the $\overline{CS}$ is set back to 1 and sleeps for $6\mu s$ [1] that is the average time taken by the ADC to convert. Afterwords, the ADC's strobe signal is checked to see if the conversion is really finished and if so the $\overline{CS}$ is set 0 and the value is read by the computer. Finally, the $\overline{CS}$ is set back to 1, signalling the end of use of the ADC.

Since the ADC will be a resource shared by more than one tasks we had to implement something to avoid the access to it when some task is already using it. The first solution we found was to use semaphores that can lock the access to a resource. This solution does its job, however, as it will be mentioned further away, there is a overhead that comes with it and then we will try a new solution. Figure 3.3(b) presents a flowchart illustrating the steps necessary to **read** the joint **position**. It starts by signalling the ADC to start reading the position and waits checking one of the ADC outputs if the operation is complete. When this happens the value is read and the routine sleeps, waiting for the next iteration.

In the reading of the current (see figure 3.3(b)), the routine is a little bit more complex. The beginning is similar to the position, it signals the ADC to start reading the current, waits for it to finish and reads the current. Then it calculates the mean current that is used along with the instantaneous current to prevent overload. If it is above the limits allowed the movement stops. If not it just sleeps and waits for the next iteration.

To change the PWM, represented by the flowchart in figure 3.3(b), the routine have to check if there is any indication to stop due to overcurrent or simple stop order. If there is no indication it changes the PWM otherwise it stops the movement and sleeps, waiting for the next iteration.

Having these tasks working properly it is possible to measure the time needed to read the position, current and to change the PWM and then define some important parameters such as the frequency of reading the position and the current.


Begin
$\overline{CS} = 0$
write control word
$\overline{CS} = 1$
sleeps for $6\mu s$
check strobe
Ready?
No
Yes
$\overline{CS} = 0$
Read Result
$\overline{CS} = 1$
End
Current
Init Read ADC
Read Value
Calculate mean Current
Is it above the limits?
Yes
Stop
No
Sleep
Position
Init Read ADC
Read Value
Sleep
PWM
Overcurrent || Break
No
Change PWM
Yes
Stop
Sleep

(a) ADC reading process

(b) Flowchart of the basic tasks

Figure 3.3: Device driver flow charts

3.2.2 Filters

One of the first observations that we made during the experiments was the high interference present in the position signal read by the ADC, so we had to implement a filter system to try to eliminate that interference. The filter used was a low-pass finite impulse response (FIR) filter. The output of the FIR filter is a weighted sum of the current and a finite number of previous values of the input. These weights were calculated using functions implemented on Matlab. Further, it will be explained the usage of this filter in more detail and presented some test results using it.

3.2.3 Internal scheduler

One of the problems we encountered was an interference between the rising edge of the PWM signal and the reading of the position. When the PWM was turned on the abrupt current position caused a strong interference on the position sensor signal. This interference was spotted using a oscilloscope where some peaks could be seen whenever the PWM was on the rising edge.

The obvious solution was to try to read the position in a different time of the PWM changes, so an internal scheduler seemed to be a perfect solution, synchronizing and scheduling the three tasks previously mentioned (read position, read current and change the PWM). This way we could define when each of those tasks would be active. We defined a schedule for those tasks which in practice is a junction of those three tasks in only one, as shown in figure 3.4 and 3.5, where *RCur* is the reading of the current, *RPos* is the reading of the position and *cnt* is a counter that is compared with the *kRCur* and *kRPos* indicating when is the next iteration of those tasks. With the scheduler, the current reading and the position reading would happen when the PWM is stable for some time. In addition, the current would be read in first place leaving the position to be read on the furthest place from the rising edge of the PWM signal. The execution rate of the *RCur* and *RPos* tasks is controlled by the internal scheduler, being selected according the time taken by those tasks to be complete and also on the CPU utilization that we want. To this subject will be analysed further after we have introduced the time tests.

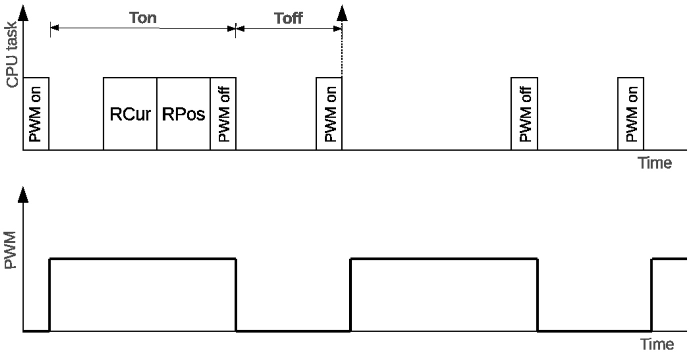


Figure 3.4: Internal scheduler

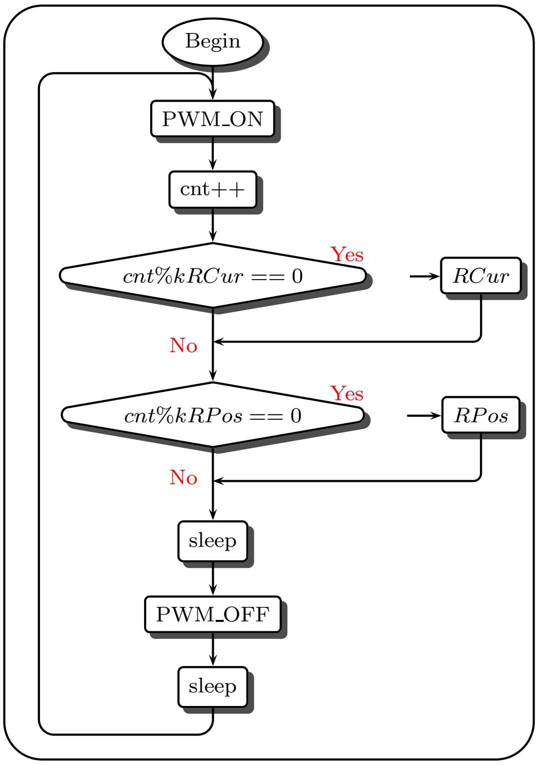


Figure 3.5: Internal scheduler Flow chart

3.2.4 Trajectory planning

The trajectory planning is pretty simple at this point. This process is done before the movement starts and consists in computing an array of position values along the movement that will be used as reference for the control systems. This array will have the solution of the equation 2.13 with a given destination coordinates and the time of execution of the movement. This equation has time as variable which can not exist programming wise, so we had to convert it into a discrete equation.

3.2.5 Control systems

In the control systems we have the highest level task on this project so far. Here we will use all the resources granted by the device drivers to get data and actuate on the robotic arm. With the help of figure 3.6 we see that the task starts by calling the function that would compute the path planning. This operation is just done once on the task. Afterwords and entering in the task cycle, we check if the joint is still within its position limits, and stops everything with an error message in the negative case. Then we calculate the error and the new PWM. The error is the difference between the actual position and reference, the PWM is calculated according to the control systems technique in use. This is presented in detail in the control systems chapter. Finally, the end of the trajectory array and the stop signal are checked. The end of that array tell us that we reached the end of the movement according to the reference. In case there is no problem the task sleeps until the next iteration of the cycle, otherwise the task ends.

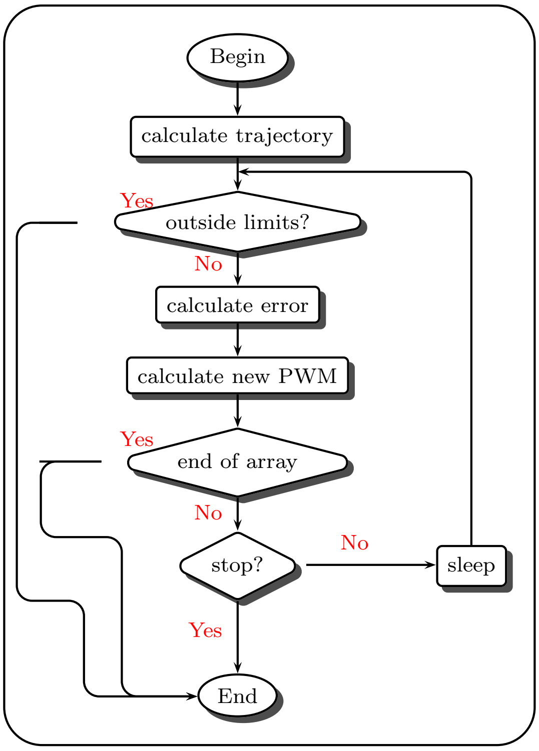

Begin
calculate trajectory
Yes
outside limits?
No
calculate error
calculate new PWM
Yes
end of array
No
stop?
No
sleep
Yes
End

Figure 3.6: Control systems task

3.3 Experimental assessment

This was the first step of the work and one of the most important ones since the quality of the data read would influence the rest of the project, mainly the control part. Here we had some problems regarding the noise present in the signal and some improvements had to be done in order to have a better reading.

3.3.1 Time measurements

One of the first tests I made was the execution time test because firstly I would need to define the frequencies of execution of the tasks previously mentioned (read position, read current and change PWM) taking into account the CPU utilization. The tests made were:

- Time needed to read the position;
- Time needed to read the current;
- Time needed to update PWM

Time needed to read the position

In order to measure the time needed to read the position, it was recorded, by software, the time in ns on the beginning and end of the reading process and then subtracted. The result obtained was the following:

Average Time (ns)	Max Time (ns)	Min Time (ns)	std (ns)
64716	67012	62774	689.38

Table 3.2: Table of the time needed to read the position, not optimized

Figure 3.7 illustrates the execution time of consecutive instances of the reading position task.

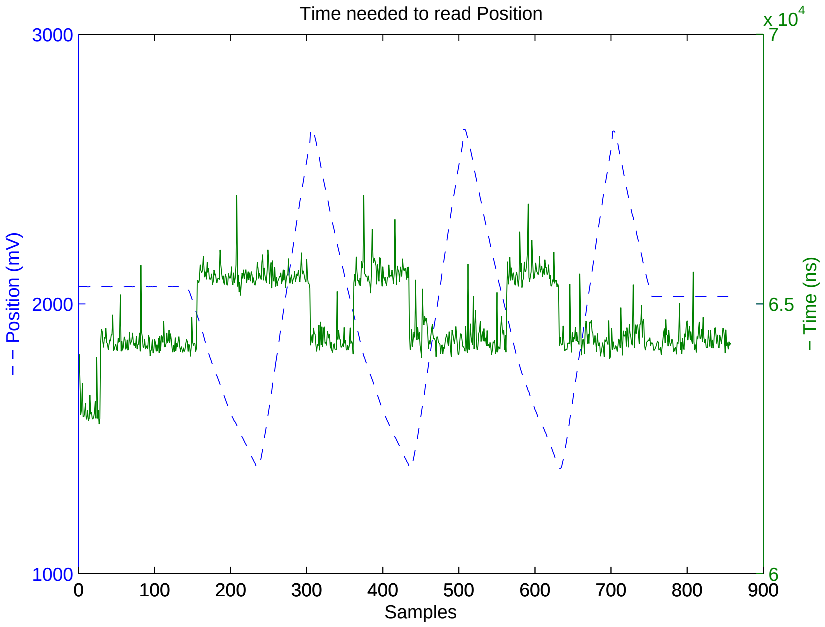


Figure 3.7: Time needed to read the position, not optimized

The worst case was 67012 ns, which is relatively high. Most part of the time is spent on the ADC conversion, as expected, however there was some overhead caused by the sleeps waiting for the ADC to finish its conversion and semaphores that prevent simultaneous access to the ADC, so we made the following optimizations:

- remove the semaphores;
- replace the ADC conversion waiting time by a pooling cycle checking the ADC strobe status.

In order to remove the semaphores, the threads had to be defined with the same priorities and I had to use a FIFO scheduler. This way the threads cannot be interrupted by others with the same or less priority. This was actually very important, because the access to the ADC must not be interrupted or have any interference with the risk of read a wrong value from it. Also it would allow me to make an internal scheduler as you will see further on.
The sleep on the ADC process was replaced by a while cycle checking its strobe status as it is shown in figure 3.8. This was just done because the waiting time was short, around 6μs [1].

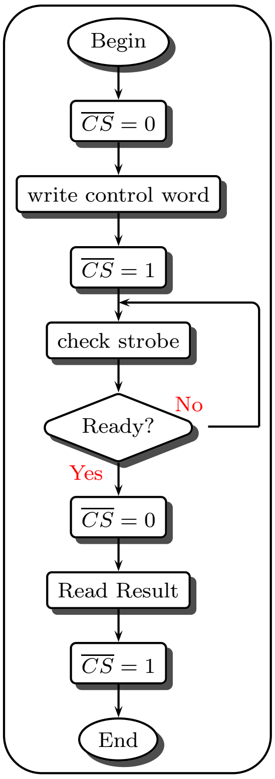

Begin
$\overline{CS} = 0$
write control word
$\overline{CS} = 1$
check strobe
Ready?
No
Yes
$\overline{CS} = 0$
Read Result
$\overline{CS} = 1$
End

Figure 3.8: ADC reading process, Optimized

Average Time (ns)	Max Time (ns)	Min Time (ns)	std
61641	64060	59444	748.16

Table 3.3: Table of the time needed to read the position, optimized

---

The results obtained after the optimization was the one presented in table 3.3

The worst case was 64060 ns which is around minus 3 μs than the previous. So we have an reduction on the time of execution of almost 5% . See also figure 3.9.

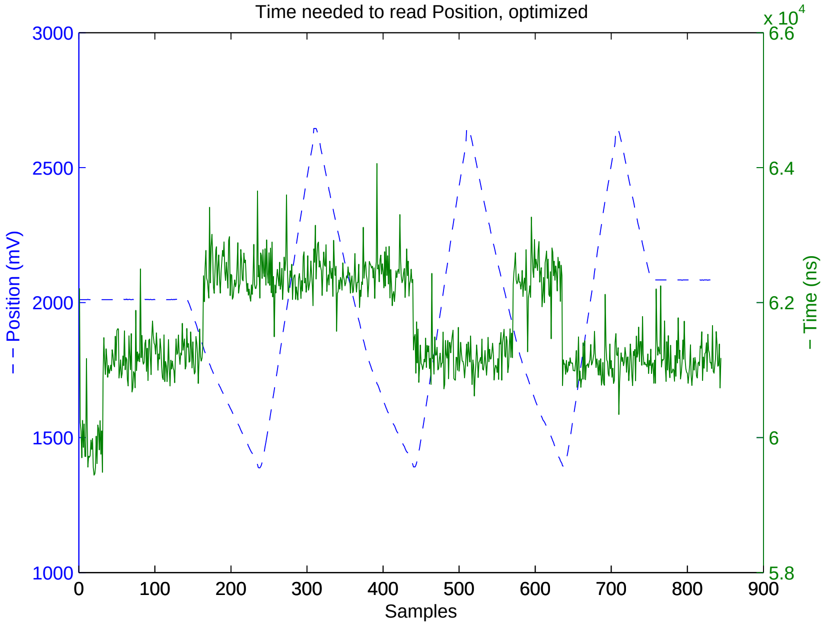


Figure 3.9: Time needed to read the position, optimized

Time needed to read the current

The process to measure the time needed to read the current was similar to the position case. It was measured already using the same optimizations made on the position and the result obtained was the following:

Average Time (ns)	Max Time (ns)	Min Time (ns)	std (ns)
60020	62476	58786	559.96

Table 3.4: Table of the time needed to read the current

The worst case of 62476 ns, which is similar to the position case. See figure 3.10 for a graphical approach.

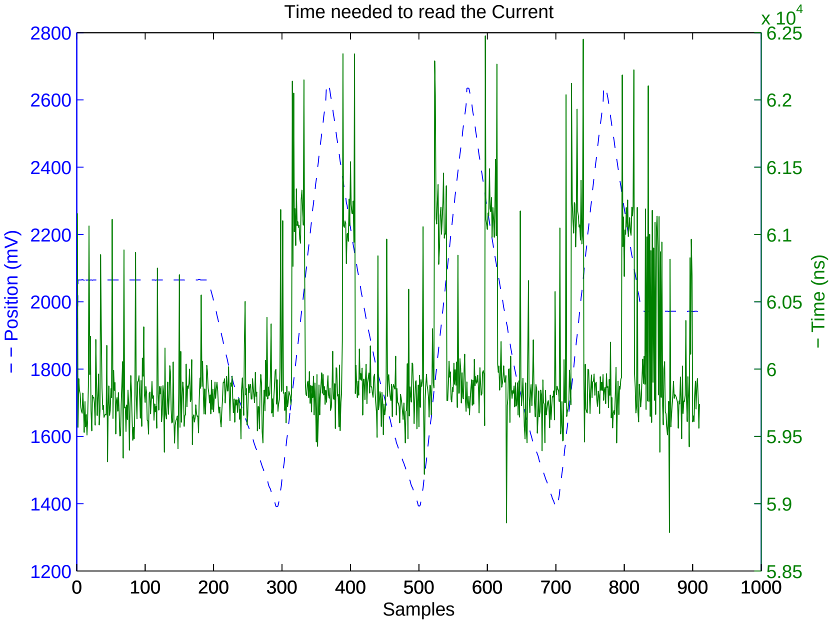


Figure 3.10: Time needed to read the current

Time needed to update the PWM

The process was, once again, similar to the previous ones. The result was the following one:

Mean Time (ns)	Max Time (ns)	Min Time (ns)	std
885	1018	824	22.76

Table 3.5: Table of the time needed to update PWM

The worst case 1018 ns, which is really low, as expected, since it only involves toggling a digital pin. See also the figure 3.11.

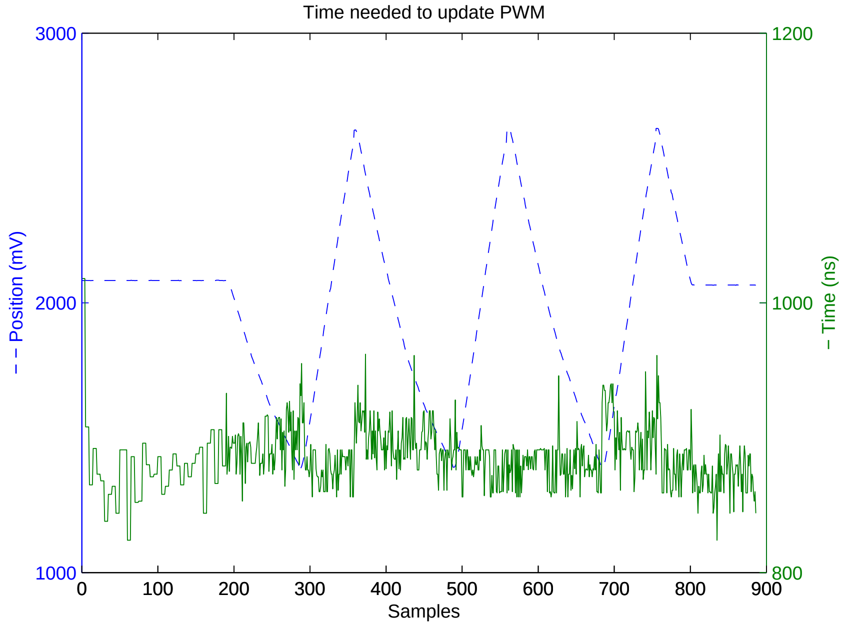


Figure 3.11: Time needed to update the PWM

Time tests conclusions

To summarize, the worst case scenario for each task was the one present in the table 3.6.

Read Position (ns)	Read Current (ns)	Change PWM (ns)
64060	62476	1018

Table 3.6: Worst case scenario for each task

With these values I could calculate the maximum frequency allowed to each task. If we consider to allocate a maximum of 50% of the utilization to these task we can conclude the following:

$Totaltime = 64060 + 62476 + 1018 = 127554ns$
$MaxUtilization = 50\% \Rightarrow T_{min} = 127554 + 127554 = 255108ns$

$$f_{max} = \frac{1}{T_{min}} = 3.92kHz \tag{3.1}$$

The frequency of PWM commonly used in this situations is about a few kHz, so we decided to use a frequency 2kHz which is lower than the f_{max}. The frequency for the reading of the position and current was also chosen as 2kHz, which is close to the values commonly used in cases like this.

3.3.2 Filters

The signal initially obtained from the position sensor was severely corrupted by interference, as can be seen in figures 3.12 and table 3.7. So it came necessary to implement a filter.

Mean (mV)	Max(mV)	Min (mV)	ΔV (mV)	std (mv)
1486.1	1533	1435	98	18.26

Table 3.7: Table with statistic values from the signal from the position, without filter

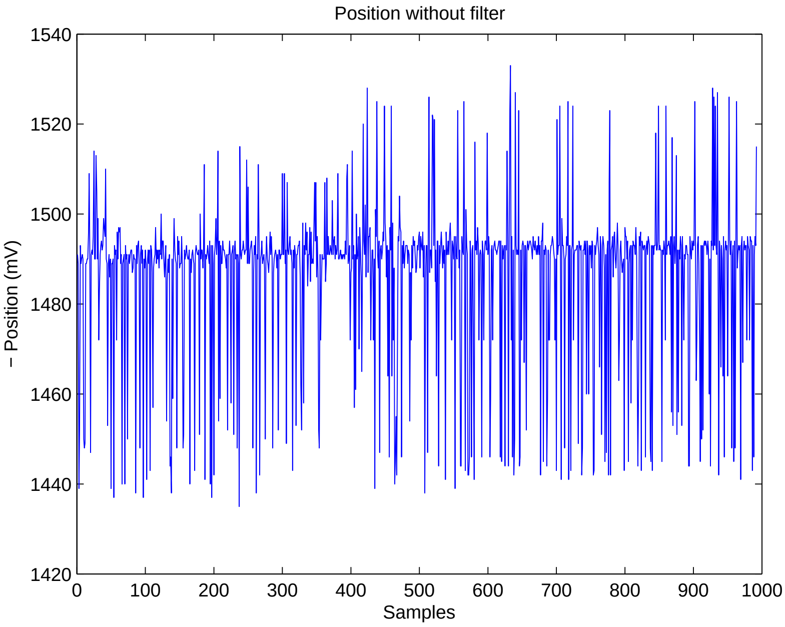


Figure 3.12: signal from the position, without filter and engine stopped

We tried two different filters implemented by software: a median filter and a low-pass FIR. The filters were implemented on the position reading task, right after the position has been read. The median filter worked very well when the engine was stopped but with the engine running had a bad behaviour so this one was not adequate to this project. The FIR filter had a better response depending on its order. The choice of the order was made testing the various possibilities using already a controller because the filter order would influence the control once the FIR order changes the delay on the samples causing the system to try to control using a signal delayed and therefore a delay on its own compensations. After some tests, the order that fitted better was 10 and the result is shown in table 3.8 and figure 3.13. The filter had a good response, improving a signal with a 18.26mV of STD and 98mV of ΔV to a signal with a 0.49mV and 1mV respectively.

Mean (mV)	Max(mV)	Min (mV)	ΔV (mV)	std (mV)
2138.4	2139	2138	1	0.49

Table 3.8: Table with statistic values from the signal from the position, with filter noise

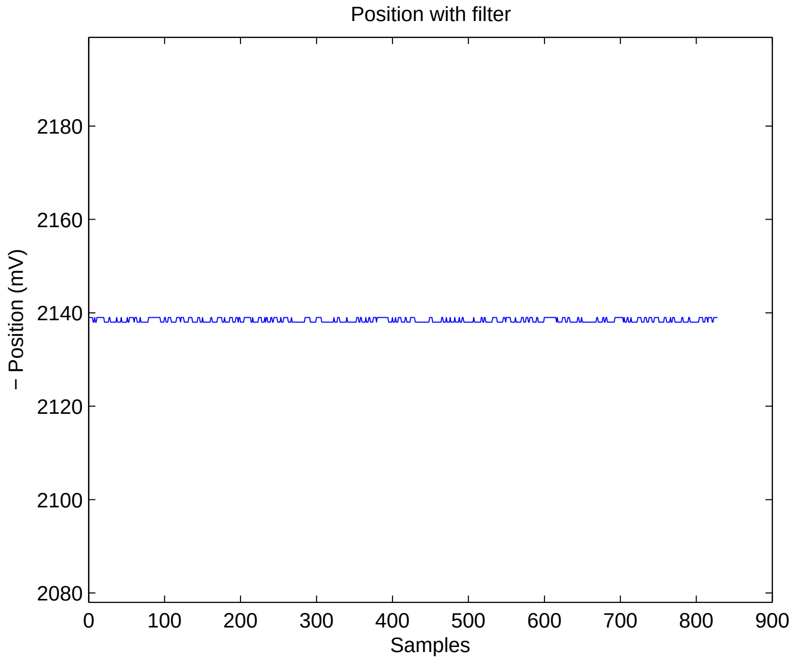


Figure 3.13: signal from the position, with filter and engine stopped

---

3.3.3 Internal Scheduler

After testing the internal Scheduler we noticed that for a low value of PWM there was still a slight interference because the time between the changing of the PWM to on and the reading of the position was still short, so we had to add a waiting time, named named stab time (stabilization time) as shown in figure 3.14, where we have one tick of the internal scheduler when every tasks are executed. The time needed to be waited before we read the position was measured on the oscilloscope with the value of $10\mu s$.

At this point, you might think that it would still not work properly because this forced waiting time between the changes of PWM to on and to off (between the rising edge and falling edge) would limit the minimum PWM. That is true, this way the minimum PWM would be 13.7% (see equation 3.2). However, when the PWM is at 13.7% or lower the engine doesn't have enough power to work, or in other words, it is in a dead band so there is no harm in using this.

$$MinPWMtimeOn = ForcedTime = stab + time_{readPosition} + time_{readCurrent} = 137\mu s$$

$$T_{PWM} = 1ms \Rightarrow PWM_{min} = \frac{PWM_{on}}{T_{PWM}} * 100 = \frac{137}{1000} * 100 = 13.7 \quad (3.2)$$

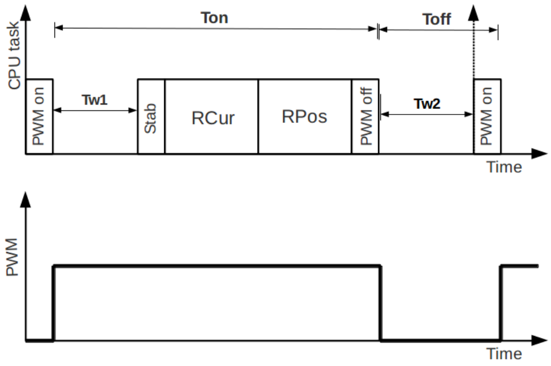


Figure 3.14: Internal scheduler

As said before, the frequencies of the tasks involved in this scheduler were all the same and with the value of 2kHz. Having all these tasks the same frequency, we didn't need to do the scheduling, so only the synchronization was done. However, this internal scheduler is prepared to do the schedule in case the frequencies are different.

To test this internal scheduler improvement, the joint had to be moving because only this way we would see the interference of the PWM rising edge. So we measured the noise in two similar situations with and without the internal scheduler. The results are showed in the table 3.9. There was a visible improvement in the performance of the system, which was reflected on the MSE that decreased from 1.33(mV) to 1.15(mV).

Situation	maxError (mV)	MSE (mV)	Error std (mV)
Without internal scheduler	3.08	1.33	1.09
With internal scheduler	2.61	1.15	1.05
Improvement (%)	15.26	13.53	3.67

Table 3.9: Statistic values from position signal, with and without internal scheduler

Chapter 4

Control System

In the last decades, robot manipulators received much attention with particular emphasis on the control system. Point-to-point control enables simple tasks such as materials transfer and spot welding, while continuous-path tracking enables more complex tasks such as arc welding and spray painting. Although a multi-link manipulator is a highly non-linear and coupled system, the presence of reduction gears of high ratios tends to linearise the system dynamics and thus to decouple the joints. Therefore, a common approach to robot control that is used in many industrial robots is single-axis PID control with each joint controlled independently as a single-input/single-output (SISO) linear system. Coupling effects among joints due to varying configurations during motion are treated as disturbance inputs. Typically, the control design is based on the knowledge of the physical system, its dynamical model and the control specifications defined, for example, in terms of stability and trajectory tracking. In line with this, the joint-space control problem is articulated in two sub-problems. First, the manipulator inverse kinematics is solved to transform motion requirements from the operational space into the joint space. Then a joint space control scheme is designed that allows tracking of the reference inputs.

This chapter discusses the implementation of a computer controller servo loop for a single axis of the robotic manipulator. As said before, the manipulator used on this project has five rotational axes (four plus the claw rotation), but only the axis 4 (see figure 2.6) was used in the analysis that will appear along the chapter. The lack of an accurate model of the system under control was determinant to pursue an empirical approach aimed at studying simple controllers and to evaluate their performance characteristics. The digital servo loop contains the motor, its load, the power amplifier and the electronics and software necessary to implement some type of closed-loop control. The block diagram of a single-axis servo loop is shown in Figure 4.1.

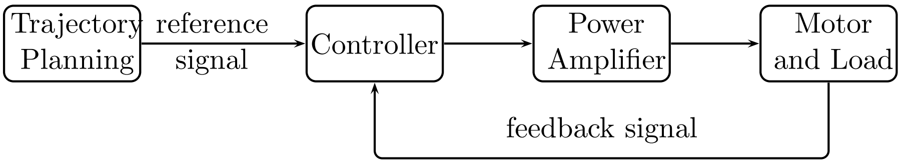


Figure 4.1: General controller model

The profile generator computes discrete position versus time set points over some interval and sends them to the digital servo loop. This discrete data signal also carries implicit information about the desired velocity and acceleration states of the motor's shaft. The goal of the digital controller is assure the position of the robot axis, driven by the motor's shaft, follows the prescribed function of time (trajectory used as reference).

Several experiments were carried out using different reference profiles and execution times. However, in order to provide a fair comparison among them, it is assumed hereinafter that the execution time is 6 seconds, the initial joint angle is -48° and the desired final joint angle is -3°. After some initial tests, the sampling frequency of the controller was adjusted to 100 Hz and the PWM signal frequency to 2 kHz. The joint position is measured from a rotary potentiometer which produces a voltage that is proportional to the shaft angle. Whenever velocity feedback is needed, the velocity information is obtained by software by taking the current position and subtracting it from the one obtained at the previous time (the time between samples is fixed).

As stated before (Chapter 2), the reference used as input for the controller is obtained by the position trajectory planning based on a 5th order polynomial function (see equations 2.13 and 2.14).

The experiments started with the simplest control model - with proportional control only – followed by the effect of adding the integral term, resulting in a PI controller. Finally, two different controller models referred in the literature [20] as the most used in robotic mechanisms are described. The use of a linear fixed-gain PID (three-term controller) is justified, primarily, by their simplicity and performance characteristics, where the "I" term ensures robust steady-state tracking of step commands, while the "P" and "D" terms provide stability and desirable transient behaviour. The cascaded P-PI controller uses an inner PI loop that controls velocity and there is an additional outer loop consisting of the position error multiplied by a gain (P term). Here, the tuning of the parameters of the PID controller was made by trial-and-error to obtain a desired performance response. More specifically, the performance evaluation is based on measures of the difference between the reference trajectory and the actual angular positions available from the potentiometer, namely the final error, the maximum error (maxError), root-mean-square deviation (RMSD) and the standard deviation of error (STD error).

4.1 Proportional controller

Although this controller is the simplest one and rarely used it could be a good point to start understanding the behaviour of the manipulator and also it is an initiation to the PID controller that is going to be used after. The model used was the one in figure 4.2 but just with proportional term in the controller. As you would expect, the proportional controller doesn't solve the control problem. After tuning the controller (pretty easy since you only have on parameter) the result was bad as expected and shown in table 4.1 and in figure 4.3 and 4.4. The value of the proportional term obtained on the tuning was $k_p = 0.24$.

Final error (DEG)	maxError (DEG)	RMSD (DEG)	Error std (DEG)
5.03	15.65	7.00	3.76

Table 4.1: Table with statistic analyse from the proportional control

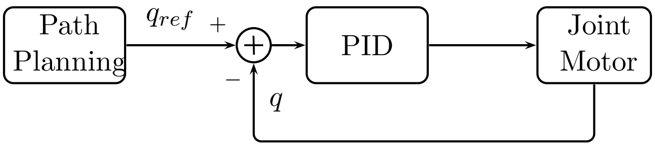


Figure 4.2: PID Contoller

You might have noticed in figure 4.4 that there is a huge delay to the controller to take effect on the initial movement. This is due to the non-linear behaviour of the engine or more precisely its deadzone. The size of this zone depends on the type of engine, actual position and the direction of the movement, this effect is demonstrated in the section 4.5. In the case shown in figure 4.4 you can see that the joint only starts to move when the PWM hits the 50%, in this case we were facing one of the worst situations, where the joint had to move against the gravity and starting on a horizontal position. Facing this deadzone problem, we tried one simple solution that consisted in giving an offset to the PWM, obliging it to react sooner. The results in presented in table 4.2 and in figure 4.5 and 4.6. The proportional term in this case was $k_p = 0.44$. This simple solution decreased the RMSD to almost half of it, improving in general the whole movement, and yet, slightly reduced the final error. The only problem resulting from this offset is an increase of minimum PWM since the PWM lower than the offset is not reachable any longer for the whole movement.

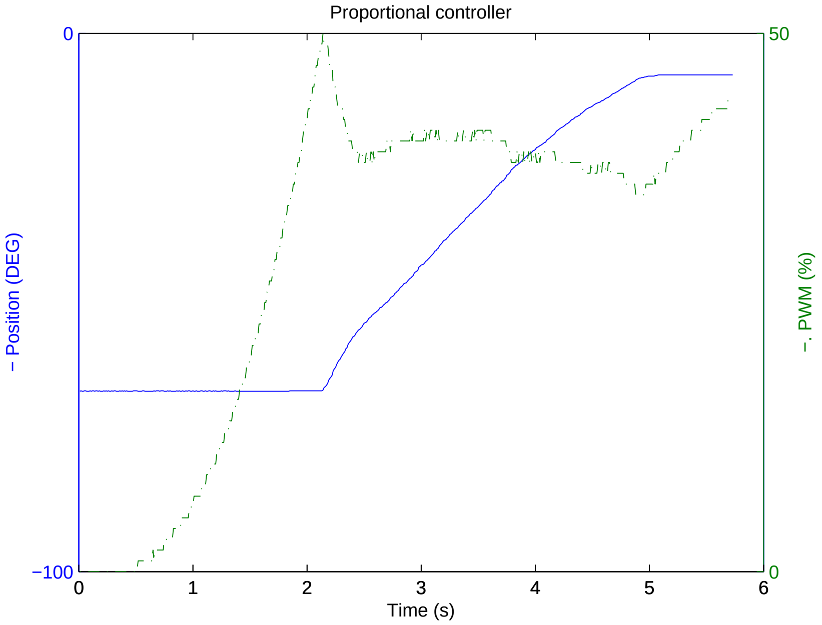


Figure 4.3: Proportional Control response, Position x PWM

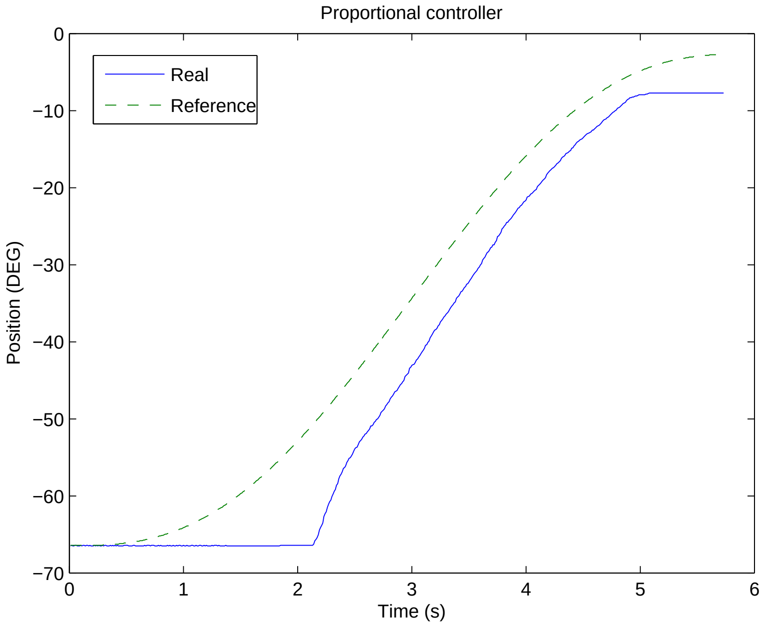


Figure 4.4: Proportional Control response, Real Position x Reference

Final error (DEG)	maxError (DEG)	RMSD (DEG)	Error std (DEG)
4.35	7.78	3.63	1.55

Table 4.2: Table with statistic analyse from the proportional control, with offset

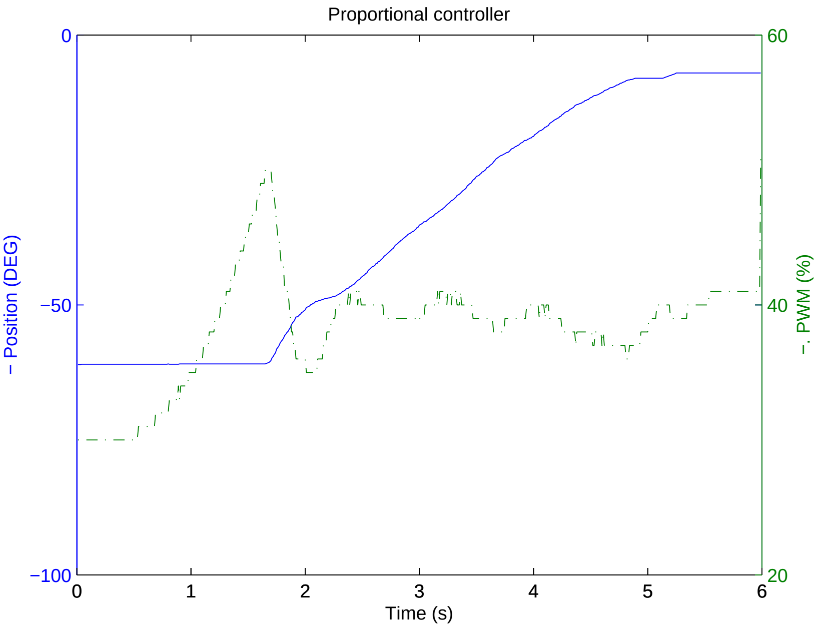


Figure 4.5: Proportional Control response, with offset, Position x PWM

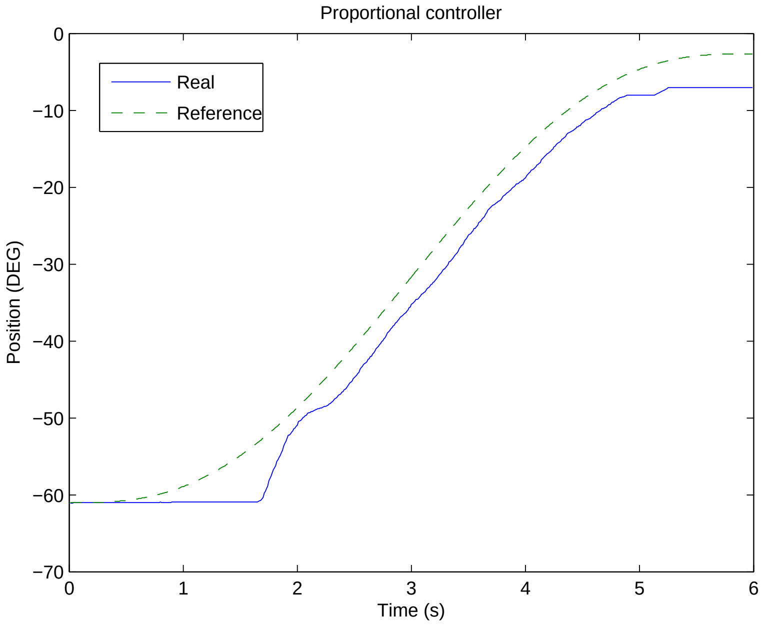


Figure 4.6: Proportional Control response, with offset, Real Position x Reference

4.2 Proportional-Integral Controller

The PI controller, unlike the proportional one, can make a good work by itself, it's not the best controller ever developed but in case we can't use derivative term it is a good choice and in this work it is showed. The result signal of the controller, the PWM, was calculated as in the equation 4.1, where $k_p = 0.44$, $k_i = 1.5 \times 10^{-3}$, the error is the result of the subtraction between real position and its reference, and errorSum is the summation of all the error until the moment. In this controller is also used and offset to deal with the engine deadzone, however in this case the offset was added as an initial value of the errorSum, thus it wouldn't affect the whole movement.

$$PWM = k_p \times error + k_i \times errorSum \tag{4.1}$$

According to table 4.3 and figures 4.7 and 4.8 the final error is pretty low (0.23^o). However there are some oscillations mainly at the beginning and the end of the movement, also the error throughout the whole movement could be better even though there was a huge improvement when compared to the proportional controller.

Final error (DEG)	maxError (DEG)	RMSD (DEG)	Error std (DEG)
0.23	2.52	0.75	0.54

Table 4.3: Table with statistic analyse from the proportional-Integral control, with offset

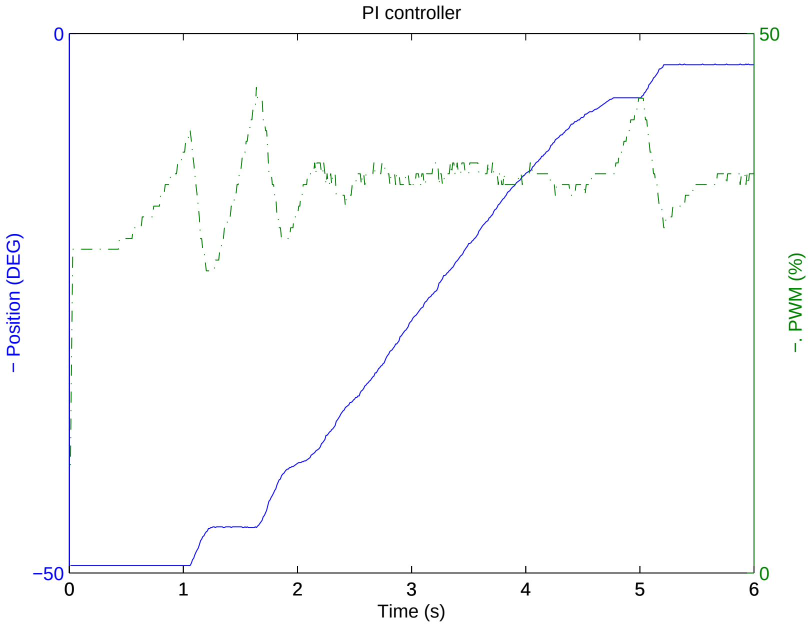


Figure 4.7: PI Controller response, Position x PWM

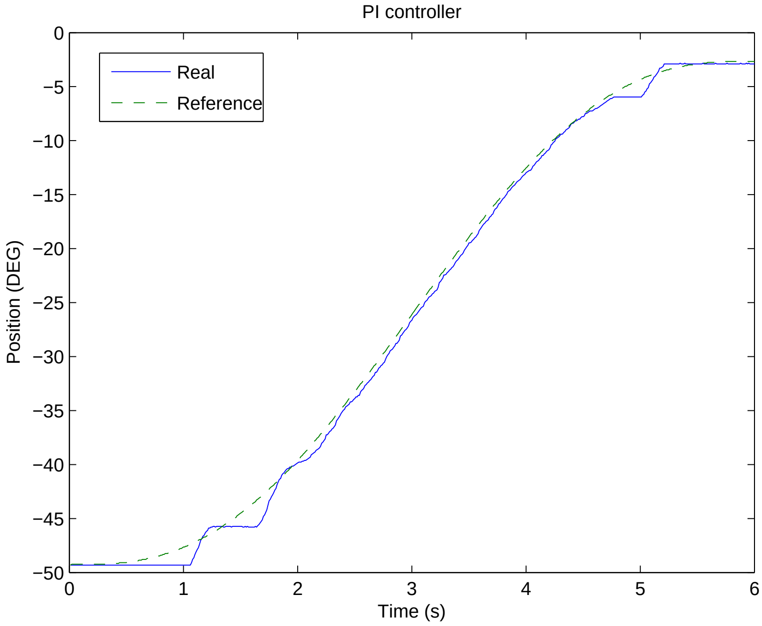


Figure 4.8: PI Controller response, Real Position x Reference

4.3 PID Controller

The PID controller is the natural evolution of the previous ones, where we have a proportional, an integral and a derivative component. Each one of these components have a different effect on the system as it is represented in table 4.4. The tuning of the parameters was based on the PI controller and followed the classic rules of how to tune the PID:

- Use KP to reduce the rise time.
- Use KI to eliminate the error.
- Use KD to reduce the overshoot and settling time.

The expression o the controller response was the one in the equation 4.2, where $k_p = 0.44$, $k_i = 4.5{\times}10^{-3}$, $k_d = 1.2$, error is the result of the subtraction between real position and its reference, errorSum is the summation of all the error until the moment and lastError is the error in the last iteration of the control task.

$$PWM = k_p \times error + k_i \times errorSum + k_d \times (error - lastError) \tag{4.2}$$

It was expected to have a good response with this controller, however the improvements in comparison with the PI controller were behind our expectations as you can see in table 4.5 and figures 4.9 and 4.10. Overal, the RMSD decreased as expected but there was still some oscillations, specially in the beginning of the movement ([0 2.5](s)) that we thought would be cancelled. These oscillations may have some connection with the high non-linearity. In the middle of the movement ([2.5 4.5](s)) we have a perfect response with a RMSD of 0.16. In the end of the movement there is some oscillation but not that serious, with a RMSD equal to 0.47. To have a better response than the PID controller one, we would need to use an other control method, maybe an adaptive control.

Response	Rise Time	Overshoot	Settling Time	SS Error
K_P	Decrease	Increase	Not affected	Decrease
K_I	Decrease	Increase	Increase	Eliminate
K_D	Not affected	Decrease	Decrease	Not affected

Table 4.4: Effect cause by each parameter of the PID

Time	Final error (DEG)	maxError (DEG)	RMSD (DEG)	Error std (DEG)
[0 6](s)	0.23	2.37	0.59	0.55
[0 2.5](s)	-	2.37	0.83	0.73
[2.5 4.5](s)	-	0.38	0.16	0.10
[4.5 6](s)	0.23	1.37	0.47	0.23

Table 4.5: Table with statistic analyse from the PID controller

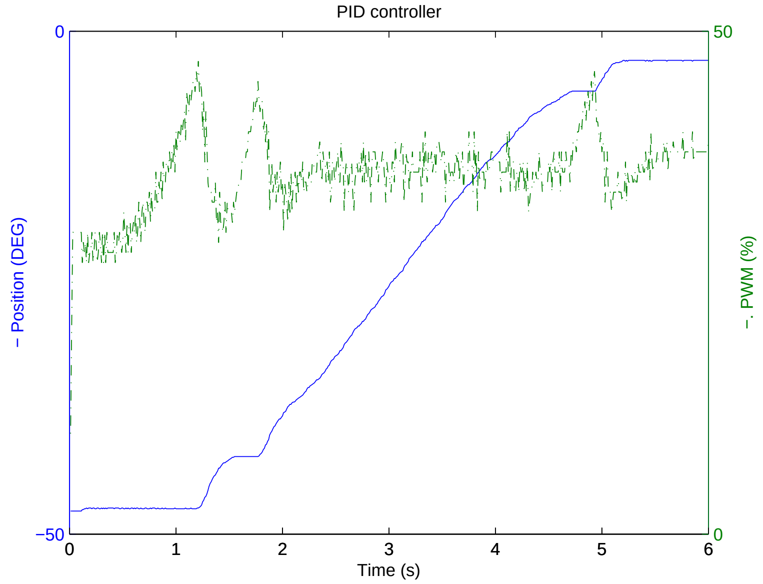


Figure 4.9: PID Controller response, Position x PWM

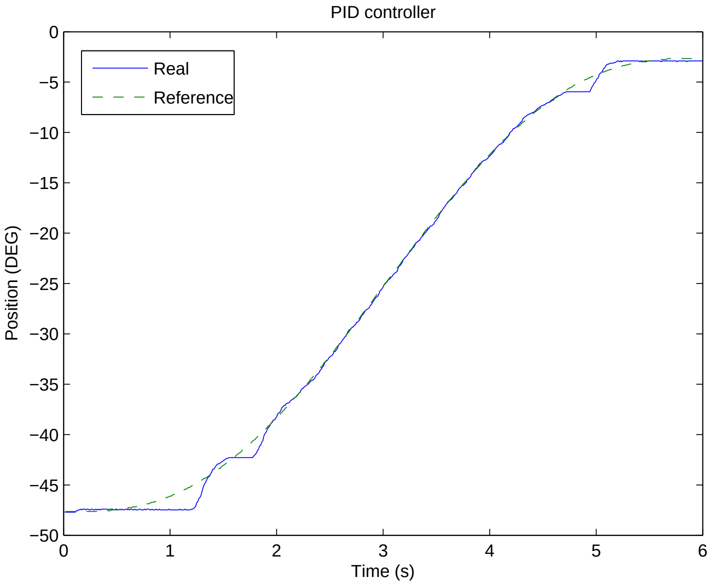


Figure 4.10: PID Controller response, Real Position x Reference

---

4.4 Cascade controller

This controller is slightly more complex than the previous ones, it controls not only the position but also the velocity (see figure 4.11). There is a proportional controller that controls the position. The output of this proportional controller is used as an input for a second controller that controls the velocity and generates the PWM signal.

The expression of the controller response was the one in the equation 4.3, where $k1_p = 0.5$, $k2_p = 3.5$, $k2_i = 4.5{\times}10^{-3}$, the error is the result of the subtraction between the real position and its reference, error2Sum is the summation of all the error2 until the moment and error2 is the error based on the velocity reference, output from the first controller and the real velocity of the joint.

$$PWM = k2_p \times error2 + k2_i \times error2Sum, erro2 = \dot{q}_{ref} + k1_p \times error - \dot{q} \qquad (4.3)$$

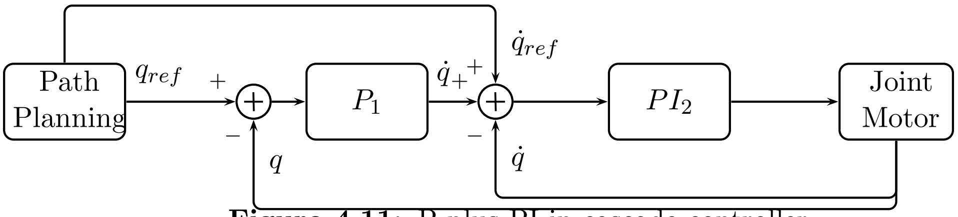


Figure 4.11: P plus PI in cascade controller

The result of this controller was worse than we expected in terms of error elimination but it had a good response in the beginning which tell us that has a better behaviour with the non-linearity than the PID controller. As presented in table 4.6 and figures 4.12 and 4.13, the error is bigger than in the PID however the big oscillations present on the the last one don't appear in this case. The reason why the error is that hight may be related to the interference existent in the velocity, since it is calculated from the position instead of being measured.

After comparing this controller to the PID, we decided to use the PID because the error is lower than here, being the error one of the most important parameters for this work. Nevertheless, we think that with a better velocity signal or some changes in the controller model this controller can be better than the PID.

Time	Final error (DEG)	maxError (DEG)	RMSD (DEG)	Error std (DEG)
[0 6](s)	1.15	1.98	1.15	0.39
[0 2.5](s)	-	1.98	1.17	0.56
[2.5 4.5](s)	-	1.53	1.19	0.59
[4.5 6](s)	1.15	1.53	1.05	0.46

Table 4.6: Table with statistic analyse from the cascade controller

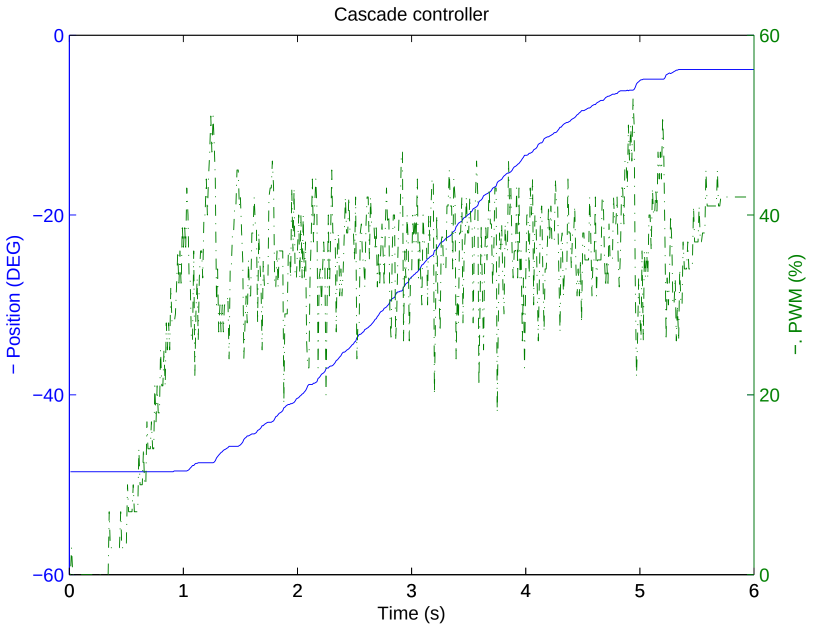


Figure 4.12: Cascade Controller response, Position x PWM

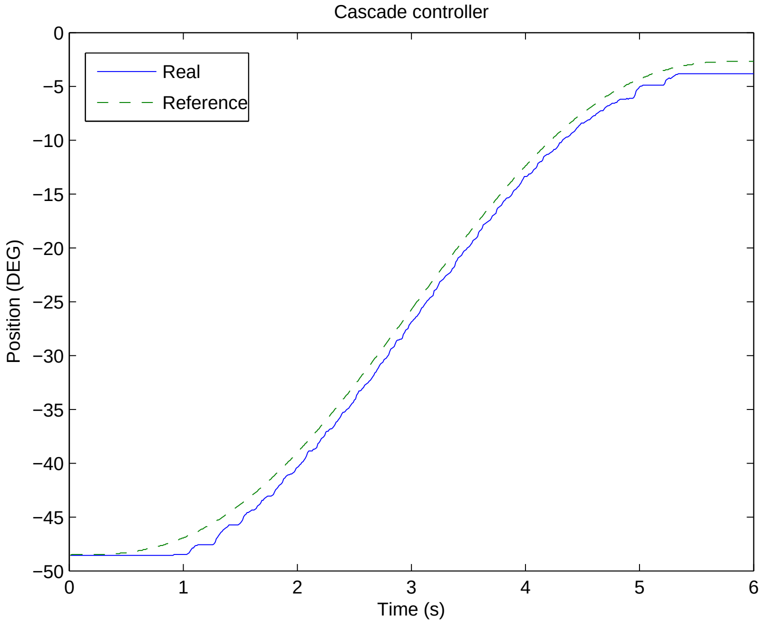


Figure 4.13: Cascade Controller response, Real Position x Reference

---

4.5 Motor Linearity

After noticing the deadzone and the dependence on the position and direction of the joint we tried to "measure" the non-linearity of the engine hopping that it could be used improve the control. To obtain the values of the figures 4.14 and 4.15 it was calculated the velocity, always on the same position, for different values of PWM. The movement was initiated with 50% of PWM, just to to have a stable initiation of the movement, and then changed to the respective PWM. Then, with this values, we made an interpolation in matlab to retrieve an approximation for the remaining values of PWM.

In figure 4.14, representing the movement against the gravity or climbing, we can see that, apart from the deadzone, there is not much non-linearity. In the opposite direction, in favour of the gravity we can see that it has a hight non-linearity especially for low values of PWM. Most important than that, comparing the two figures we can see that there is a big difference between the two movement directions and therefore the influence of the weigh of the arm on this motor.

After analysing this results we conclude that in order to have a better performance by the control we should use a different technique, probably adaptive control.

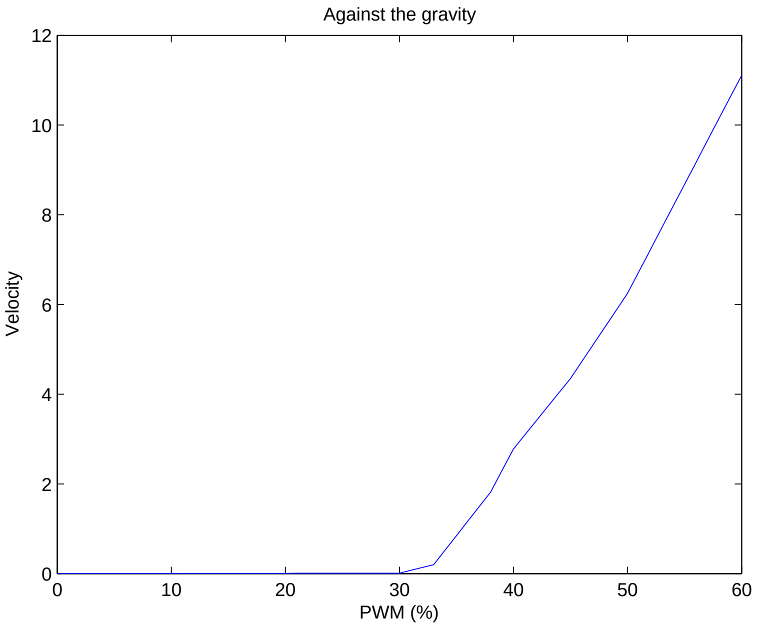


Figure 4.14: Non-linearity of the engine, against the gravity

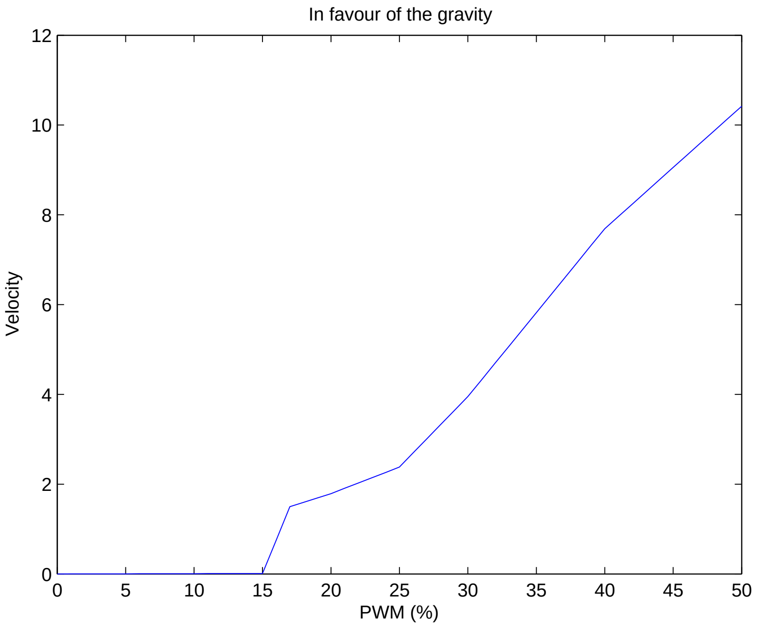


Figure 4.15: Non-linearity of the engine, in favour of the gravity

Chapter 5

Network sniffer

In the beginning of this dissertation we developed an application that would be helpful not only in this dissertation but also in related projects. This application's objective was to capture specific packets in the network, belonging to a specific stream, and show a chart, in real-time, of the bandwidth used by that stream of packets. For instance, this would allow us to see the transmission rate of a video streaming and its variations, in real-time. The sniffer needed to filter the packets according to its protocol. The protocols in which we were interested were: TCP, UDP and FTT (FTT-SE). This application should be able to show multiple streams on the chart, corresponding each of those to a specific filter applied. This software was developed in *java* and used the library *libpcap* [8] that was responsible for the capturing of all the packets reaching the network interface.

5.1 Configuration

This application initiates with a configuration window that allow us to configure the filters that we pretend as well as the view preferences, as shown in figure 5.1.

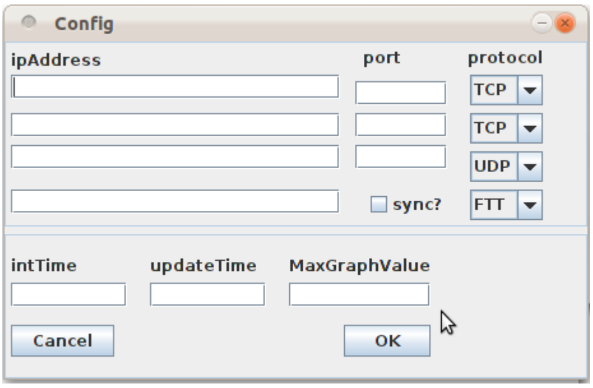


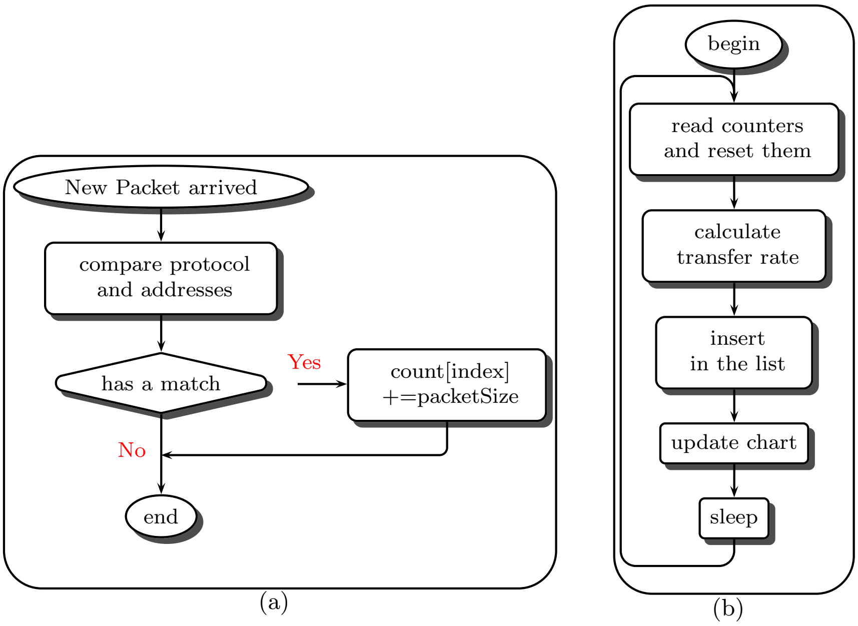
Figure 5.1: Configuration window

There we can enter the IP address of the source in the *ipAddress* field. In the *port* field we should write the destination port of the packet. In the protocol select box we can choose which protocol we want to capture, having the possibility to choose between the TCP,

UDP and FTT. If we have selected the FTT protocol, we should enter different parameters corresponding to its protocol (messageID in the *ipAddress* field and select or not the sync check box). In the bottom of the window we have some parameters related to the chart visualization. This parameters were not implemented yet. The *intTime* is the time between between the calculations of the transmission rate of the stream, and is used by default as one second. *updateTime* is the interval time between the chart update, and is also being used as one second. The *MaxGraphValue* is the maximum value that the chart should show.
All of the configuration done here is automatically saved to a file and restored in the next session.

5.2 Implementation

As said before, this application was developed in java and used the library *libpcap* through a java wrapper named *jnetpcap*. To draw the chart we used the java library *jchart2d* [7]. The implementation of this software can be divided in two parts: the capturing and the visualization. The capturing of the packets (figure 5.2(a)) is initiated by the arrival of a new packet. An event is launched by the library upon the arrival of a new packet, and a handler in the software takes care of the rest. After the arrival, the packet is compared with the stream information that we gave on the initial configuration. If there is a match, the packet size is added to the counter of the respective stream.



(a) (b)

Figure 5.2: Application flowcharts

While the packets are being captured, there is a chart showing the transmission rate for each stream in real-time. As shown in figure 5.2(b) the cycle is initiated by reading the counters of the capture and reseting them right away. Then, with those values, it is calculated the transmission rate of each stream and all the data is inserted in a circular linked list. Finally we update the chart and sleep until the next iteration. The chart always shows the lasts values for each stream within a window of time, as shown in figure 5.3, where we can see screen capture of the application while capturing packets. Each of the streams are represented by a chart line which describes its transmission rate at each instant of time. In that figure each unity of the time axis represents one second, which is the default value for it.

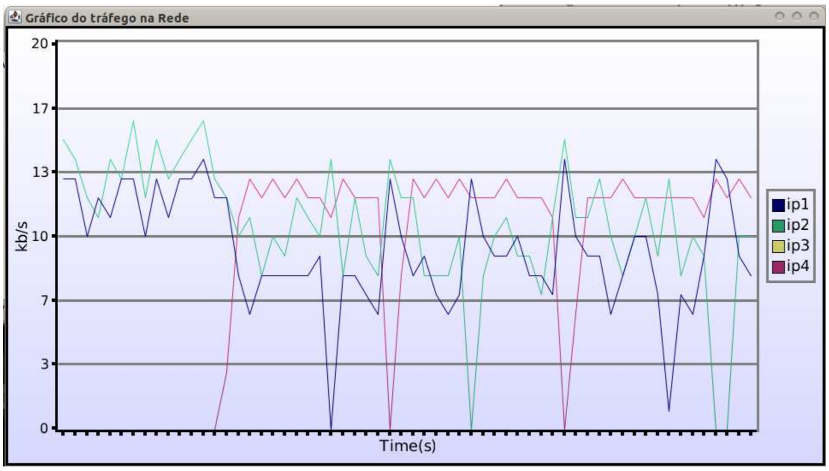
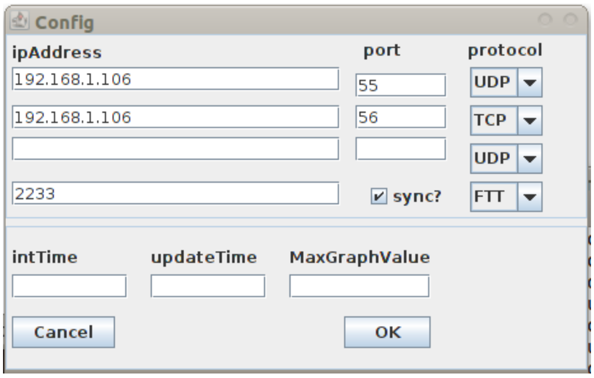


Figure 5.3: Screen capture of the network sniffer working

5.3 Test

To confirm the effectiveness of this application, we made a simple test with three streams which were being transmitted from one computer using the program *packETH* [9]. The three streams used the three different protocols (TCP, UDP, FTT). The stream with the UDP protocol was configured with a packet size of $512kB$ and with a delay between packets of $2.5ms$, which results in a transmission rate of $204.8kB/s$ or $1638.4kb/s$. In the TCP the packet size was $1024kB$ and the delay $6ms$, which makes a transmission rate of $170.67kB/s$ or $1365.33kb/s$. Finally, the FTT had a packet size of $1024kB$ and a delay of $4.5ms$ resulting in a rate of $227.56kB/s$ or $1820.44kb/s$. The configuration on the network sniffer was the one shown in figure 5.4, where the UDP protocol is on the stream one, the TCP on the stream two and the FTT on the stream four. Having the configuration done, we made two tests. The first one, represented in figure 5.5, was done with the streams being transmitted in different periods of time. The second one, as shown in figure 5.6, had periods in which the streams were being transmitted at the same time. With this test we can see that the software is working properly, showing the correct values.

Although there are a lot of improvements to be done, this program can already perform its main objective without any problems. For the moment, the program only supports the three protocols previously mentioned and only four streams can be displayed. However, thanks to the extensibility concern in which the software was developed, it can be easily improved allowing it to display a variable number of streams and filter more protocols.



ipAddress	port	protocol
192.168.1.106	55	UDP
192.168.1.106	56	TCP
		UDP
2233	☑ sync?	FTT

intTime	updateTime	MaxGraphValue

Figure 5.4: Screen capture of the configuration for the test

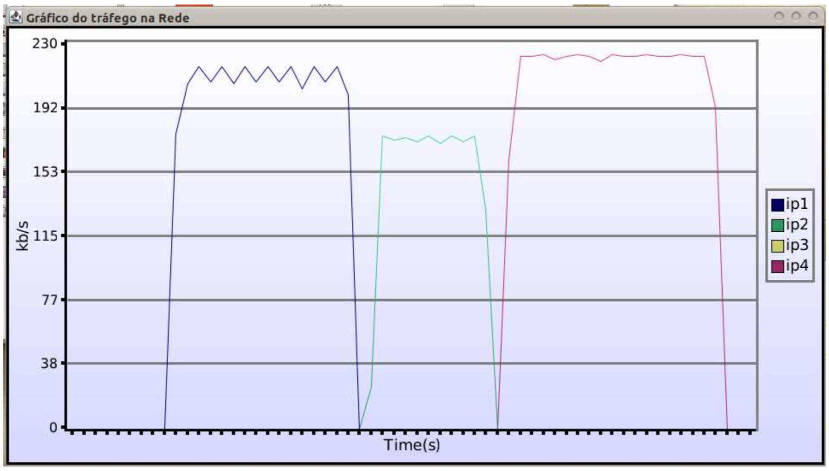


Figure 5.5: Screen capture of the first test

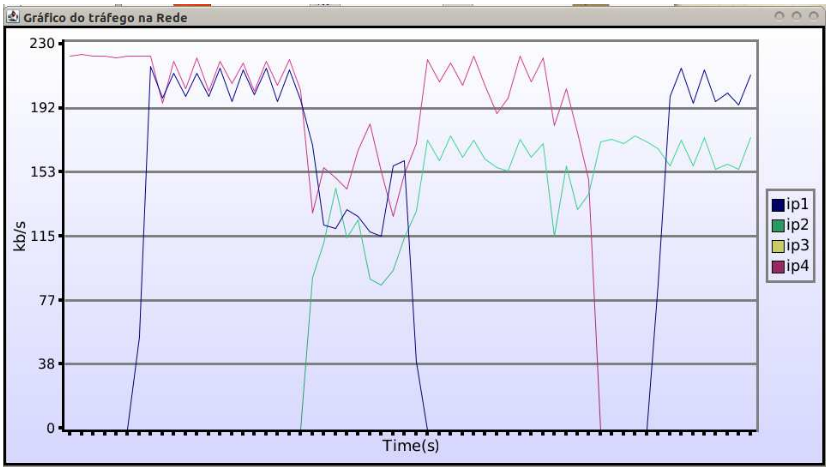


Figure 5.6: Screen capture of the second test

Chapter 6

Conclusions and Future Work

This dissertation can be divided in two main parts: Signal acquisition and control. In the signal acquisition we had some problems related to the interference present on the signal. To face this problem we implemented filters and an internal scheduler aiming to synchronize the device drivers tasks in order to remove some interference in the signal caused by the PWM change to on. The filters had a good response, decreasing the std and the noise amplitude from 18.26(mV) and 109(mV) to 0.49(mV) and 1(mV) respectively. In the control we found two different solutions, the PID and the cascade controller, with two different behaviours. The PID controller presented a low final error of $0.23^o Deg$ and a low error in the middle of the movement ([2.5 4.5](s)), having a RMSD of $0.16^o Deg$. However the beginning ([0 2.5](s)) of the movement was a little problematic, presenting some oscillations and having a RMSD of $0.83^o Deg$. The cascade controller presented a good answer facing the non-linearity of the motor and didn't show any big oscillations in the beginning of the movement but the error in general was worse than in the PID having a RMSD equal to $1.15^o Deg$.

The main objective was to develop basic structures that would allow us to test the FTT-SE protocol and it was achieved. This structures are ready to implement the distributed control, nevertheless the control can be improved. The distributed control and therefore the test of the protocol was not possible to be done due to time issues. There are some changes and improvements that can be done:

- Finish the remaining objectives of this dissertation (distributed control and test of the FTT-SE protocol).
- Test different control techniques, such as, an adaptive control.
- Evaluate different filters to improve the quality of the position measurements.
- Investigate a new approach to measure/calculate the velocity in case it will be needed for the control.

Bibliography

[1] Max186 datasheet.

[2] Common normal. `http://en.wikipedia.org/wiki/Common_normal_(robotics)`, 2011.

[3] Edf scheduling. `http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling`, 2011.

[4] Fixed priority pre-emptive scheduling. `http://en.wikipedia.org/wiki/Fixed_priority_pre-emptiv` 2011.

[5] Ftt protocol. `http://paginas.fe.up.pt/~ftt/index.html`, 2011.

[6] Harts project. `http://www.ieeta.pt/lse/hartes`, 2011.

[7] jchart2d. `http://jchart2d.sourceforge.net`, 2011.

[8] libpcap. `http://www.tcpdump.org/`, 2011.

[9] packeth. `http://packeth.sourceforge.net`, 2011.

[10] Retmik. `http://sweet.ua.pt/~lda/retmik/retmik.html`, 2011.

[11] Rm scheduling. `http://en.wikipedia.org/wiki/Rate-monotonic_scheduling`, 2011.

[12] Round-robin scheduling. `http://en.wikipedia.org/wiki/Round-robin_scheduling`, 2011.

[13] Rtai. `http://en.wikipedia.org/wiki/RTAI`, 2011.

[14] Rtkpic18. `http://sweet.ua.pt/~lda/str/revdet42-rtk.pdf`, 2011.

[15] Rtlinux. `http://en.wikipedia.org/wiki/RTLinux`, 2011.

[16] Shark. `http://shark.sssup.it/kernel.shtml`, 2011.

[17] Giorgio C. Buttazzo. *Hard RealTime Computing Systems: Predictable Scheduling Algorithms and Applications, third edition.* 2011.

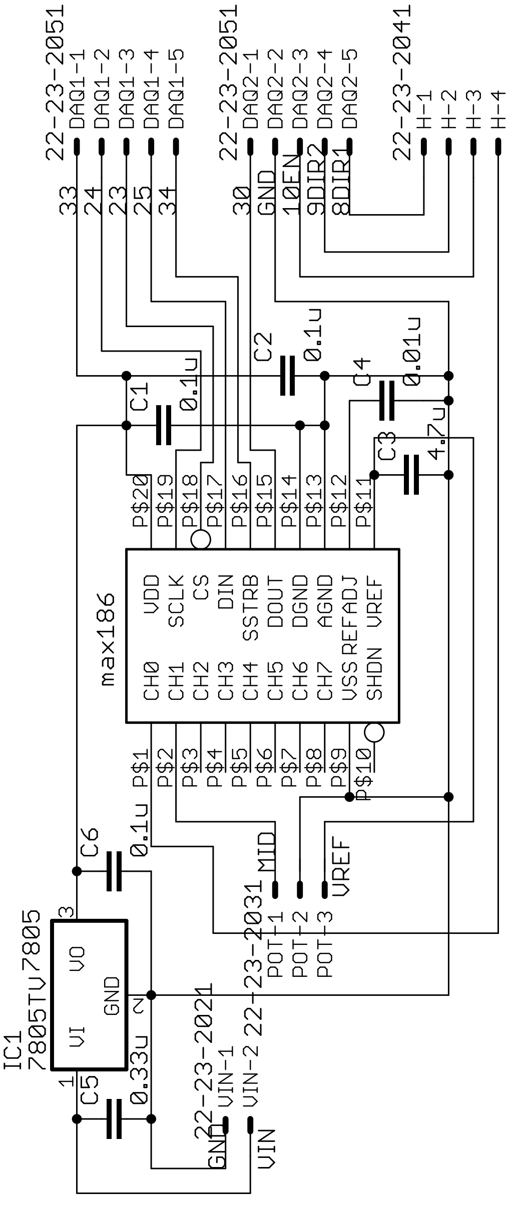
[18] Reza N. Jazar. *Theory of Applied Robotics.* 2010.

[19] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[20] V. Santibáñez R. Kelly and A. Loría. *Control of Robot Manipulators in Joint Space.* 2005.

Appendix A

6.3 Circuit


22-23-2051
DAQ1-1
DAQ1-2
DAQ1-3
DAQ1-4
DAQ1-5
33
24
23
25
34
22-23-2051
DAQ2-1
DAQ2-2
DAQ2-3
DAQ2-4
DAQ2-5
30
GND
10EN
9DIR2
8DIR1
22-23-2041
H-1
H-2
H-3
H-4
C2
0.1u
C4
0.01u
C1
0.1u
C3
4.7u
P$20
P$19
P$18
P$17
P$16
P$15
P$14
P$13
P$12
P$11
max186
VDD
SCLK
CS
DIN
SSTRB
DOUT
DGND
AGND
REFADJ
VREF
CH0
CH1
CH2
CH3
CH4
CH5
CH6
CH7
VSS
SHDN
P$1
P$2
P$3
P$4
P$5
P$6
P$7
P$8
P$9
P$10
C6
0.1u
MID
VREF
22-23-2031
POT-1
POT-2
POT-3
IC1
7805TV
7805
VI
VO
GND
1
2
3
C5
0.33u
22-23-2021
VIN-1
VIN-2
GND
VIN

Apendix B

6.5 Acronyms

ADC Analog-to-digital converter
DAQ Data acquisition board
FIR Finite impulse response
MSE Mean squared error
PWM Pulse-width modulation
RSMD Root-mean-square deviation
SPI Serial Peripheral Interface Bus
STD Standard deviation