ISSN 0280-5316
ISRN LUTFD2/TFRT--5722--SE

Hard Real-Time Control of an Inverted Pendulum using RTLinux/Free

Mathias Svensson

Department of Automatic Control
Lund Institute of Technology
February 2004

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
February 2004

Document Number
ISRN LUTFD2/TFRT--5722--SE

Author(s)
Mathias Svensson

Supervisor
Manuel Velasco Garcia, Barcelona, Spanien
Anders Rantzer LTH Lund

Sponsoring organization

Title and subtitle
Hard Real-Time Control of an Inverted Pendulum using RTLinux/Free (Hård realtidsreglering av en inverterad pendel i RTLinux/Free).

Abstract
The content of this master thesis regards the implementation of a control system in RTLinux/Free, and the use of this for control of a straight track inverted pendulum.
A controller developed on the hypothesis that the system is sampled with the sampling time t_s will perform differently than expected if sampled with the sampling time t _= t_s. If implemented in a regular operating system, the computer will fail to meet demands of hard real-time sampling constraints, i.e. that the desired sampling time must be the one used by the system at all times and in all situations. A computer controlled system in this environment will thus perform di.erently than expected most of the time. Often ”di.erently” is the same as various degrees of worse.
The underlying objectives for this project was the desire to be able to control and do research on a hard real-time system connected to an unstable non-linear plant — the inverted pendulum. This master thesis covers the work of creating a hard real-time platform using RTLinux/Free on which to add control features. It deals with communication betweenhard real-time and soft real-time, kernel module programming, modelling, linear state feedback, linearization, energy functions, friction compensation, etc.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
88

Security classification

Recipient's notes

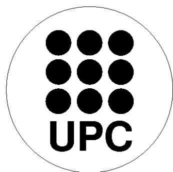
The report may be ordered from the Department of Automatic Control or borrowed through:
University Library 2, Box 3, SE-221 00 Lund, Sweden Fax +46 46 222 44 22 E-mail ub2@ub2.se

Master Thesis

Hard Real-Time Control of an Inverted Pendulum using RTLinux/Free

Mathias Svensson

5th February 2004



Universitat Politècnica de Catalunya
Departament d'Enginyeria de Sistemes
Automàtica i Informàtica Industrial
Director: Manel Velasco Garcia

Lund Institute of Technology
Department of Automatic Control
Supervisor: Prof. Anders Rantzer

Contents

1 Introduction **1**
1.1 Structure . . . 2
1.2 Definitions of Terms Used . . . 2

2 What is Real-time? **3**
2.1 Definition of Real-time . . . 5
2.1.1 Soft vs. Hard Real-time . . . 6
2.2 Real-time and Automatic Control . . . 7
2.3 Real-time Characteristics for Regular Operating Systems . . . 8
2.3.1 Main Goal of Operating Systems . . . 9
2.3.2 Latency Test for Standard Linux . . . 10
2.3.3 How to Achieve Hard Real-time in Linux . . . 10

3 RTLinux **12**
3.1 The Real Time Kernel . . . 12
3.2 Latency Test for RTLinux . . . 14

4 The Process **16**
4.1 Pendulum Description . . . 16
4.2 Hardware Communication . . . 16
4.3 RT-DAC4/PCI Drivers . . . 17

5 The Program **19**
5.1 Program Overview . . . 19
5.2 RT-DAC4/PCI Drivers . . . 19
5.3 The Kernel Modules . . . 20
5.3.1 Calculation of Derivatives . . . 21
5.3.2 Faster Floating Point to Integer Conversions . . . 21
5.3.3 Input-Output Latency and Computation Time . . . 22
5.3.4 Dummy Module . . . 24
5.4 The Java Program . . . 24
5.5 Putting Everything Together . . . 25

6 Model of Pendulum **27**
6.1 Mathematical Model of Pendulum . . . 27
6.2 Equilibrium Points . . . 29
6.3 Linearization in Up-position . . . 30
6.4 Linearization in Down-position . . . 30
6.5 State Space Description . . . 31

6.6 Finding the Parameters 32
6.6.1 Center of Gravity of Pendulum 32
6.6.2 Moment of Inertia for Pendulum 32
6.6.3 Pendulum Friction 33
6.6.4 Cart Friction 33
6.6.5 Numerical Matrixes 34

7 Controlling the Pendulum 35
7.1 Control Objectives 35
7.2 Discretization 35
7.3 State Machine Description 36
7.4 Calibration and Assumptions 36
7.5 State Feedback 38
7.5.1 Stabilizing Control Up 39
7.5.2 Stabilizing Control Down 41
7.6 Model vs Real Pendulum 42
7.7 Swing-Up and Swing-Down 42
7.8 Cart Control 46
7.8.1 Software Spring 47
7.9 Friction Compensation 47

8 Deterministic Disturbance 51
8.1 Variable Sampling Time 51
8.2 Variable Disturbance 53
8.3 Possible Future Tests 54

9 Result and Summary 55

A User Manual 56

B Miscellanous 57
B.1 RT-DAC4/PCI Driver API 57
B.2 Latency Test Diagrams 58
B.3 Screen Dump of the Java Program 60

Bibliography 61

Acknowledgements

I wish to thank the following persons for their help with this master thesis project. It is always a privilege to work with intelligent people.

Dr. Ricard Villà and PhD Student Manel Velasco for your warm welcome and your constant support, tips, and interest. It was very nice to work with you!

Dr. Josep M. Fuertes for all clever questions, funny tests of my skills in Catalan (which I never passed), and your cheerful mood.

Prof. Anders Rantzer and PhD Student Johan Åkesson for your support via mail from Sweden.

PhD Student Miquel Monroig for good collaboration.

Magnus Bäck for your expertise within LaTeX and with every computer related question I seem to be able to ask.

Marcus Bellander for the thorough proofreading of this report.

I also would like to express my gratitude towards the following persons whose contribution was not directly related to the project, but nevertheless very important:

Thank you Pilar "Pili" Aldana Alfaro for your great company and for all nice cups of cortado and magdalenas. I will miss you in Sweden.

Carlos and Dulce for the nice trip to Castelldelfels and Garaf and for the nice appartment we rented.

All the nice visitors from Sweden.

Most of all

Thank you Katja for being so supportive and for taking so good care of me!

Chapter 1

Introduction

In the process of converting a continuous description of a controlled system into the discrete-time equivalent, the system is observed at equidistant samples and the dynamic relations of the states and the input signals are derived. Different time distances between two consecutive samples (sample time) result in different discrete time systems. A controller developed on the hypothesis that the system is sampled with sample time t_s will perform differently than expected if the system is sampled with a sample time $t \neq t_s$. If implemented in a regular operating system, the computer will fail to meet demands of hard real-time sampling constraints for a number of reasons. The perhaps most important reason for this is that the scheduler in a regular operating system does not have real-time as its main objective, but instead throughput. A computer controlled system in this enviroment will thus perform differently — often worse — than expected.

The objectives for this master thesis were to implement a hard real-time control system, and to use this for control of a straight track inverted pendulum. The priority was higher for getting the control program to work than actually getting the control to work perfect using it. If the program worked satisfying, desired control could always be achieved later. The inverted pendulum is a process with many interesting features, of which the most interesting for this project are:

- Stable in down-position and unstable in up-position
- Non-linear
- Contains friction

The pendulum in question was delivered with drivers and software for use in a Windows MATLAB system, but for many reasons it would be interesting to control it in a true real-time system.

It was desired to use Linux as a base for the controller, since Linux in many ways is interesting. It is free software, well established, and offers a nice way of customizing every individual system. The operating system used was RTLinux/Free, an operating system released under GPL v2[1] by the company FSM Inc[2]. RTLinux offers a nice way of combining the well tuned soft real-time features of standard Linux with true hard real-time constraints on selected parts of the system.

[1]GNU General Public Licence version 2, (www.gnu.org)
[2]Finite State Machine, (www.rtlinux.org)

1.1 Structure

The structure for the thesis is as follows:

Chapter1 Introduction.

Chapter 2 explains the meaning of real-time and why it is interesting to study. This chapter also introduces basic terms related to real-time and it gives example of that standard operating systems are unsuitable for hard real-time.

Chapter 3 introduces RTLinux and explains the fundamental difference compared with standard Linux.

Chapter 4 describes the process to control — the inverted pendulum.

Chapter 5 explains how the program is put together. Figures illustrate the separation of hard real-time and soft real-time, and the general composition of the different parts.

Chapter 6 derives a model for the inverted pendulum and gives suggestions of how to find various parameters.

Chapter 7 describes the procedure for swinging the pendulum up and controlling the pendulum using state feedback.

Chapter 8 is dedicated to test the control when it is disturbed in different controlled ways.

Chapter 9 Discussion of the result and possible future work.

1.2 Definitions of Terms Used

A Process may contain a number of different tasks. Each process typically runs in its own protected memory space.

A Task takes care of something that has to be done. A task may contain/be zero or more threads, allthough it is most common that each task is implemented as one single thread.

A Thread executes periodically or by interrupts, and often shares the same memory space with different threads belonging to the same task/process. A thread is also known as *light weight process.*

Chapter 2

What is Real-time?

It is important to describe the meaning of real-time. Without this knowledge it is difficult to understand this report and its purpose.

Typical for real-time systems (and for computers in general) is that they must respond to events. An event could be the user pressing a key on the keyboard, a camera being connected to the USB, or a connected transmitter indicating that something has happened. These are all examples of *non-periodic* events, but events may also be *periodic*, i.e. the event occurs at times t, $2t$, $3t$, etc. Examples of periodic events are the timer event of the internal clock and the time for execution of a periodic task. The timer events of the internal clock are very important for the computer. These events are actually the core driving factor for the computer's extreme versatility.

The time from when an event occurs to when the computer becomes aware of it and responds is of major interest in real-time. Depending on the application, the demand of knowing how long time it will take until the computer responds to an event will vary.

An easy way to implement the part of the system that deals with events would be to use a loop that runs at a high frequency. Every revolution of the loop would go through a list of things, checking if anything has happened that requires action. An algorithm of this kind is said to be based on *polling* and may for instance look like this:

```
while(true) {
  ...
    if( transmitter signal above threshold )
      then ( take action )
    else if( time for thread x to execute )
      then ( execute thread x )
    ...
    else if( any key on keyboard down )
      then ( ... )
    ...
}
```

An implementation like this will quickly get large, complex and difficult to overview. This strategy has many negative features, and a few of the ones that relate to real-time are the following:

- The time from when an event occurs to the time when action is taken depends among other things on the current position of the loop execution, and on what the actions for the other events may be.
- It is impossible to set priorities to different tasks. For instance, it may be very much more important that the system responds to the transmitter signal than send a file to the printer. In this case it would be nice if there were a guarantee that the action for the transmitter signal would be taken before anything else, i.e. it will be given highest priority. A guarantee like that cannot be given using polling.
- Adding events will make the response times longer.

In the early age of computers polling was the way to solve event problems, but today most computers use *interrupts* instead. The polling loop is then replaced by an implementation that waits for interrupts that are associated with events, and while waiting the CPU is free to execute other tasks. This is a much more economic way of solving the problem, and using this strategy it is also possible to set priorities on different tasks, which is particularly important in real-time. The fundamental difference between polling and the use of interrupts, is that in polling it is the computer that has to check if something has happened, but using interrupts it is instead the environment that informs the computer.

Whenever an event occurs, the current execution is stopped and the *interrupt service routine* (ISR) related to that particular event is launched. If the event happens to be related to the highest priority task among the group of tasks that want to execute, it gets to execute immediately.

With the invention of interrupts it became possible to separate large and complex processes that deal with many things into smaller parts — one for every individual task. This is called *multitasking* and has proven to be a powerful paradigm for structuring real-time, *interrupt driven* systems. The basic idea is to break down a large problem into smaller ones, and then let every problem be solved by a number of individual interrupt driven tasks. Consider for instance the surveillance system for a large manufacturing industry. Instead of using a program based on a polling loop similar to the one shown above, the program is instead separated into smaller parts, i.e. *threads* with the following pseudo functionality:

```
...
init();
...
while( running ) {
  wait_for_interrupt()

    // Do work

}
```

On a system with only one CPU the different tasks must share this single CPU. In fact, as soon as the situation occurs that there are more tasks then CPUs, these CPUs must be shared. Whenever things must be shared there must be a supreme thing that decides how this sharing is performed. Without this there would be chaos. This supreme thing, *the scheduler*, tries to satisfy all demands and make the computer

run in a nice way. There are many scheduling algorithms that a scheduler can use, of which algorithms based on the democratic algorithm *Round Robin* probably are the most common ones.

2.1 Definition of Real-time

Many people assume that real-time means really fast. This is not true. The constant development of computers and CPUs does not by itself push us closer to the limit of real-time, and a slower computer is not by nature worse on real-time applications than a faster one. In the same manner it is not always important that the response is extremely fast, although fast is in general preferable before slow.

The definition of real-time is instead *deterministic* and *fast enough*. Fast enough is, as many things, subjective without further explanation, and therefore it is vital to explain what it means. Later on the different extents of deterministic times will be discussed.

Example 1 Consider a person filling water into a bath tub. Filling a normal bath tub takes a while and in the most common case the level only have to be "high enough" to make the bather happy. The strategy for controlling the level of water is to open the tap, keep it open until the desired level is reached, and then close the tap. Most people open the tap, leave the room and then check the level on a regular basis until the tub is full. This regular basis is approximately once every minute, and at maximum once every 30 seconds. If we would use a computer for this purpose it would therefore be sufficient to take samples (checking the current water level) once every 30 seconds. To check the water level 10 times a second would not make the bather happier, and would only be a waste of time since this time probably can be used more efficiently. On the other hand, checking the water level once every 5 minutes would highly increase the risk of finding the tub overfilled. For this control, the first computer ever constructed would be equally good as the latest X GHz on the market.

Example 2 Now change the bathroom to a nuclear power plant and change the bath tub to the reactor holding the radioactive rods. Instead of filling the tub with water, the control objective is instead to control the activity of the nuclear reaction underlying the electric power delivered by the plant. This activity must be within a range that is much more specified than "high enough", as for the water in the previous example, and the computer that measures the activity must perhaps measure 1000 times a second to be able to correct[1] any deviation. The high sampling rate is due to the unstable process — a chain reaction like this increases its activity exponentially without control. To control this would require a computer which is much faster than the previous one. It is likely to believe that the requirements on the computer would be very different as well.

However, not even example 2 qualifies as an example requiring an extreme sampling time. Consider instead the process of controlling the laser of a DVD which requires a sampling rate close to 100 kHz. This demands a very fast computer.

[1] Activity is often controlled by moving the rods in and out of a "shell" that absorbs the radioactive radiation and hence slow down the chain reaction.

These examples demonstrate that the definition of *fast enough* depends on the system to be controlled. It is important to find[2] a sample time which is suitable for every individual case.

Besides being fast enough, the response also have to be deterministic. To which extent the response have to be truly deterministic depends on the situation. Next section deals with the fact that some applications demand a truly deterministic response time, whereas it is enough with a more vague determinism in other cases.

In non-real-time systems the correctness, λ, of a result depends only on its value, i.e. $\lambda = f(value)$, but in a real-time system it also depends on the time at which it was produced, i.e. $\lambda = f(value, time)$. To rely on old information, in the belief that it is new, could be dangerous.

The two examples also, in a natural way, bring us to further definitions – the division of real-time into *hard* real-time and *soft* real-time.

2.1.1 Soft vs. Hard Real-time

The division of real-time in to *hard* and *soft* real-time is also important to understand. It is important to know to what kind of real-time a system belongs. The circumstances to this separation is the question of what the consequence will be if the computer fails to perform an operation (e.g. take a sample) on time. If the sample would be extremely delayed in the first example, the bather may face the fact of water on the floor, money spent on more water than we needed and perhaps the small frustration of even more water on the floor when he gets in the tub, since it is full to the top. If the sample would be moderatly delayed, it would not make any different to the final result at all.

In the second example, on the other hand, failure to measure the temperature even for just a moment may result in catastrophic consequence. Furthermore, it is likely to believe that it is very important that the true sampling times really are the expected ones. Using fast sampling, it does not take much time delay to rapidly decrease the phase margin. This is where the importance of deterministic response times comes in. If the response times are truly deterministic, i.e. it is beforehand possible to get a maximal value of the response time, then the control system may be tested with this delay to be proven stable or not.

The requirements on the equipment are very different in the two examples. In the first one it would be sufficient with a normal computer that would start up nicely a few times a week, run for about 15 minutes and then shut down. The computer that controls the power plant must run perfectly 24 hours a day for a long period of time. It must not, by no manner of means, fail to do its job even for a split-second.

Soft real-time is when the system has to do its job on time, but if it fails now and then it is not that important. The program or system may still work as expected, and the "on time" demand is more of a goal than true requirement.

Hard real-time [3] is when failure to do the job on time means total failure. One single delay is as bad as if the program had not started at all.

Described in terms of correctness, λ, if assumed the delivered value in some sense is correct, so that the correctness depends on the time, the separation yields

[2]Finding this time is not always a trivial task.

[3]It is debated whether hard real-time is in turn separated into critical and non-critical hard real-time.

$$\lambda srt(value, t) = \begin{cases} \text{true} & \text{if } t \le t_{deadline} \\ \delta(t) & \text{if } t_{deadline} < t \ll \infty \end{cases}$$

$$\lambda hrt(value, t) = \begin{cases} \text{true} & \text{if } t \le t_{deadline} \\ \text{not valid} & \text{if } t_{deadline} < t \end{cases}$$

where $\delta(t)$ depends on the individual application. It may have a very steep slope towards "not valid" in e.g. video conference systems, and a very flat slope in for instance a web browser.

The performance is often measured in *latency*. The general definition of latency is the time from that an event occurs to the computer reacts and launches the related ISR. For instance, the latency[4] for a periodic thread is the difference between when it is supposed to execute (internal timer interrupt) and the time when it really does, i.e. the first line of code starts to execute. This latency is of special important in this report.

When using soft real-time it is entitled to look at the average of the performance. If the latency is distributed in a nice way it make sense to look at the mean of this distribution. If a web server fails to deliver pages in time now and then, or the user must have to wait a few seconds longer then usual once in a while, then the web server is still considered to be functioning. These are situations in which the average is of greater interest.

When using hard real-time, the average performance is instead of low interest. The absolute focus is on one thing – the worst case performance. If everything that can disturb the system disturbs it at the same time, what is the performance then? Will the system fulfill its demand? This is the main focus in hard real-time. A superb average can not make up for a bad worst case time, no matter how unlikely (positive probability (> 0), though) it is that it occurs.

2.2 Real-time and Automatic Control

Most control processes are continuous by nature, but almost every modern controller is discrete by nature. Why is this? To get information about the process, the computer has to take samples on a more or less regular basis. No matter how fast the computer samples the process, it will still lack information of what is going on between the samples. Hence, it will not have a continuous function describing the process but instead a discrete series of numbers. By definition, it is thus discrete.

The way to "implement" a continuous controller on a computer is to calculate its discrete time equivalence, and implement this instead. This is done by assuming equidistant observations of the process and then derive the discrete description of the relation between the input to the system and the output, i.e. the discrete time dynamics.

In the process of converting the continuous system to the equivalent discrete one, the "mathematics" without further ado relies on that the sample times demanded are the ones that actually are being used. If this is not the case, the equivalence is broken, and the computer controlled system is not the expected one. It is therefore of greatest interest that the demanded sample times are the ones that actually are being used, or at least that the deviation from these are sufficiently small.

[4] Also known as *jitter*.

Most control applications do not qualify as hard real-time applications, and are in fact developed to be robust to various latency in the sampling time, whereas a small number actually qualify as hard real-time control systems, e.g. aircraft, spacecraft and surgical assistance within medical care.

2.3 Real-time Characteristics for Regular Operating Systems

Before going further, it is important to state that jitter often has its origin in two different sources: in the programmer's code and in the real-time limiting factors of the operating system and the hardware. The former is easier to minimize using different sources of information — there is a lot of good litterature about real-time programming. The latter, however, is not that easy to remove. This thesis deals with these latter limitations.

The pendulum at UPC was delivered with drivers and program to run in a Windows system. The program was an add-in to MATLAB which worked quite well when the computer did not have to do many other things at the same time. Why not settle with this? The program works just as expected and it both brings up the pendulum to up-right position and stabilizes it when there.

In the control-loop that controls the pendulum, the program measures certain values of the pendulum and then uses these to tell the motor how to run to make the pendulum stand up. To be able to control it nicely this program takes samples with a frequency of 100 Hz. When developing the controller it is therefore assumed that the sampling time is 10 ms, but for many reasons the time between two consecutive samples will instead be distributed according to a certain distribution of which the average is expected to be close to 10 ms. However, the standard deviation of this distribution may vary quite a lot. Since the controller is optimized for the sampling time of 10 ms, every time the true sampling time is different, the controller will perform in a suboptimal way. In an operative system like Windows or Linux the controller will perform in a suboptimal way most of the time. Due to the fact that the controller is robust to errors like this and the process is quite nice, the controller still works and it is not until we provoke the computer that we will se really bad performance. Try for instance to drag a window slowly over the screen while running the controller. This will spread the true sampling times far away from the desired, and the worse behavior, made up mainly by two different effects, is clearly visible. One effect is that whenever the samples are delayed, the controlled process, which is non-linear, runs in open loop longer than expected, and the risk of the process moving into areas in which the behaviour of the developed (linear) controller is undefined increases. The other effect comes from the mathematical conversion from continous time to discrete time. Due to the math, the controller "knows" that a certain deviation from the reference (angle, angular velocity, etc) measured over one sample, corresponds to a certain error and thus a certain control signal. If the same deviation would occur during a time which is longer than one true sample time, the controller would overcompensate for this, and if the same deviation would occur during a time which is smaller than one true sample time, the controller would undercompensate for this.

Windows may be many things, but clearly not real-time. In Linux, the results would have been more or less the the same.

2.3.1 Main Goal of Operating Systems

A normal operating systems like Windows or Linux does not have real-time objectives as its main goal. Instead the main goal is *throughput*. This makes these operative systems perfect for normal use, and even rather good at soft real-time, but not suitable in the situations requiring hard real-time.

The basic scheduler, typically based on *Round Robin*, lets a task run for a short period of time[5], then suspends it and lets another task run for a short period of time. The ability for a high priority task to suspend a low priority task which has not finished its time slice is called *preemption*. Most schedulers in normal operative system use priorities and support preemption, and this is one of the reason why they are suitable for soft real-time.

In hard real-time the system must perform an operation in a well-defined time, but for a number of reasons normal operating systems can not, even though they use relative priorities and supports preemption, set an upper limit on this time. Some of the reasons why standard Linux is not suitable for hard real-time are:

- Kernel system calls are not preemeptible. Once a process enters the kernel, it can not be preempted by a higher priority process until it is ready to leave the kernel. If an event occurs that in normal situations would preempt the running process, it simply has to wait.
- The process of paging[6] is very difficult to predict. It is almost impossible to know how long it will take to get a page off a disk, and therefore also impossible to set an upper bound for the time a process may be delayed due to this.
- The Linux scheduler is based on "fairness" with a primary goal of throughput. Thus, a low priority process that has been waiting for a long time may be chosen to execute even though a high priority process is ready to run.
- Linux reorders I/O requests from multiple processes in order to make more efficient use of the hardware. A low priority hard disk read request may be given precedence over a read request from a high priority one to minimize the movement of the disk head.
- "Course Grained" synchronization of data access means that there are long intervals where one task has exclusive use of some data. The opposit is called "Fine Grained" synchronization, and this uses more specific synchronization to the price of more overhead. Standard Linux uses Course Grained synchronization, but real-time would prefer the Fine Grained one. This is one of the essential points where it is obvious that a well functioning compromise between real-time and non-real time objectives is difficult to find.

Keep in mind that all of these "features" are very smart and exist for good reasons. They just do not go hand-in-hand with hard real-time.

[5] This "time slice" is sometimes referred to as *quantum*.

[6] If the RAM is fully used, parts of the RAM content (pages) has to be written to disk to make room for more. When the information written to disk later on is needed, it has to be read from the hard disk to the RAM again, which in terms of I/O speed is a slow process.

	Std No load	Std load	Impr. No load	Impr. load
Min (µs)	−2210	−9615	−3823	−6164
Max (µs)	1301	83388	3836	27510
Mean (µs)	-0.3	46.7	-0.5	5.63
Std deviation	46	1999	103	448

Table 2.1: *Distribution parameters*

2.3.2 Latency Test for Standard Linux

To measure the periodic latency of a standard Linux process a simple test program was written. This program enters a periodic loop where it just before sleep calculates the time when it is supposed to wake up, and when it wakes up it measures the deviation from this time and writes the value to a file. The program was being executed for 60 seconds at two different conditions. In the first execution, the system is under no load (nothing else is done except running this program), and in the second execution there is system load (start/stop programs, read from disk, etc). The period time was chosen to be 10 ms, since this is a sampling time often used when controlling inverted pendulums. Table 2.1 and table 2.2 show data from the test. The plotted distributions are shown in Appendix B. When the system is under no load, the mean value is very close to zero, but of and on the latency either increases or decreases with several milli seconds. The behavior is the same — but amplified — when the system is under load, and latencies of instead several tens of milli seconds are not unusual. Clearly this is not suitable for hard real-time.

In the next test, the periodic process has been modified to hopefully improve the latency. The process has been given a high priority and has been locked in the RAM so that it cannot be paged out. This is a "standard" way of improving the real-time performance of a Linux thread. Surprisingly enough the latency seems to be worse with the improved process when there is no load on the system, but better when there is load. Especially the maximum latency has been decreased with approximately 70 %. The standard deviation for the latencies has also been considerably reduced for the improved process. However, these latency-times are not suitable for hard real-time due to the lack of deterministic behavior. It is still not possible to get a specification on the worst case latency time except by running tests, and who knows if the tests will simulate all possible real-life scenarios?

As mentioned above, a normal sampling time for controlling inverted pendulums of reasonable size is 10 ms. Even using the improved way of programming there is a high risk of getting deviations in this sampling time of up to 275%. Imagine the effect if a process requires even faster sampling for efficient control.

2.3.3 How to Achieve Hard Real-time in Linux

Responsible for handling I/O requests, interrupts, scheduling, etc, and hence responsible for the fact that standard Linux is not suitable for hard real-time is the kernel. If this is where the problem come from, then this is the thing that must be modified to make the real-time performance better. There are at least two different approaches to give Linux a more deterministic behavior.

Preemption Improvement of the standard kernel. It is possible to change the

		(ms)	Std No load	Std load	Impr. No load	Impr. load
$-\infty$	$\leq L_p <$	-10	0	0	0	0
-10	$\leq L_p <$	-5	0	28	2648	1
-5	$\leq L_p <$	0	2619	2826	3352	2887
0	$\leq L_p <$	5	3381	3107	0	3111
5	$\leq L_p <$	10	0	31	0	0
10	$\leq L_p <$	20	0	4	0	0
20	$\leq L_p <$	30	0	2	0	1
30	$\leq L_p <$	50	0	0	0	0
50	$\leq L_p <$	∞	0	2	0	0

Table 2.2: *Data from the latency tests*

standard kernel to make it more deterministic, by e.g. making kernel system calls preemptible. From kernel 2.4 it is possible to patch the standard kernel to get this feature.

Interrupt Abstraction by replacing the standard kernel with a small deterministic real-time kernel. The real-time kernel is placed between the standard kernel and the hardware, and acts as an intermediary for the interrupts. The term "interrupt abstraction" comes from the fact that the real-time kernel abstracts the true hardware interrupts so that the standard kernel does not know that it is not communicating directly with the hardware.

Both ways has drawbacks and advantages. However, after having improved the standard kernel the improved kernel is still too big and advanced to give the true deterministic behavior hard real-time demands. It is a fact that it is difficult to compromise hard real-time with non real-time in a satisfying way. For that reason, and for several others, the way this project achieved hard real-time performance was using interrupt abstraction. There are two major implementations of the interrupt abstraction approach:

RTLinux [7], developed at the New Mexico Institute of Mining and Technology under the direction of Victor Yodaiken. RTLinux exists both as open source (RTLinux/Free) and as a proprietary version (RTLinux/Pro) offered by FSM Labs, Inc.

RTAI [8] is by some considered to be an enhancement of RTLinux, although they have similar performance. RTAI is developed at the Dipartimento di Ingeneria Aerospaziale, Politecnico di Milano under the direction of Prof. Paolo Mantegazza.

For this project RTLinux/Free was chosen. This decition was taken only because more information was found about RTLinux than about RTAI at an early stage. Later information about RTAI did not encourage me to change my mind. However, based on e.g. information in tbe nice book "Linux for Embedded and Real-Time Applications", by Doug Abbot, I am quite sure that similar results to the ones achieved in this report using RTLinux/Free would have been achieved with RTAI as well.

[7] www.rtlinux.org
[8] www.rtai.org

Chapter 3

RTLinux

In standard Linux the kernel is placed between the hardware and the different kind of user processes, and hence separate the user-level tasks from the hardware. User level tasks have to communicate with the hardware through kernel calls, and any time the kernel disables interrupts, signals from and to the hardware are blocked. In the non-real-time world this works fine, but this protection prevents programs from having the total control and access to the hardware that hard real-time requires.

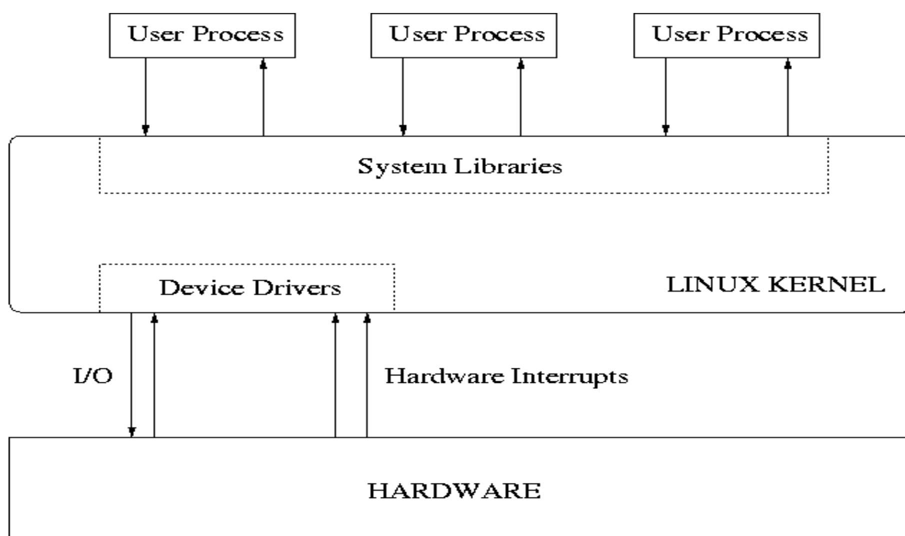


Figure 3.1: *The role of the standard Linux kernel.*

3.1 The Real Time Kernel

RTLinux is a hard real-time operating system based on interrupt abstraction. The RTLinux kernel, RTCore, has the ability to run a secondary operating system (Linux) as a low priority thread, and in the mean time run hard real-time threads with strict timing constraints. The idea for this separation begun with the experimental operating system called *MERT* about 25 years ago.

In RTLinux the RTCore places itself as an intermediary between the standard Linux kernel and the hardware. A far as the standard kernel is concerned, nothing is different. This is possible since RTLinux simulates the true *hardware interrupts* by *software interrupts*. Hardware interrupts not related to real-time activities are held and passed to the standard Linux kernel as software interrupts when the RTCore is idle, and interrupts related to real-time result in start of appropriate ISR. At all times, the hard real-time threads, running as *kernel modules*, are *privileged*, i.e. they have direct access to hardware, and they do not use virtual memory and hence can not be paged out.

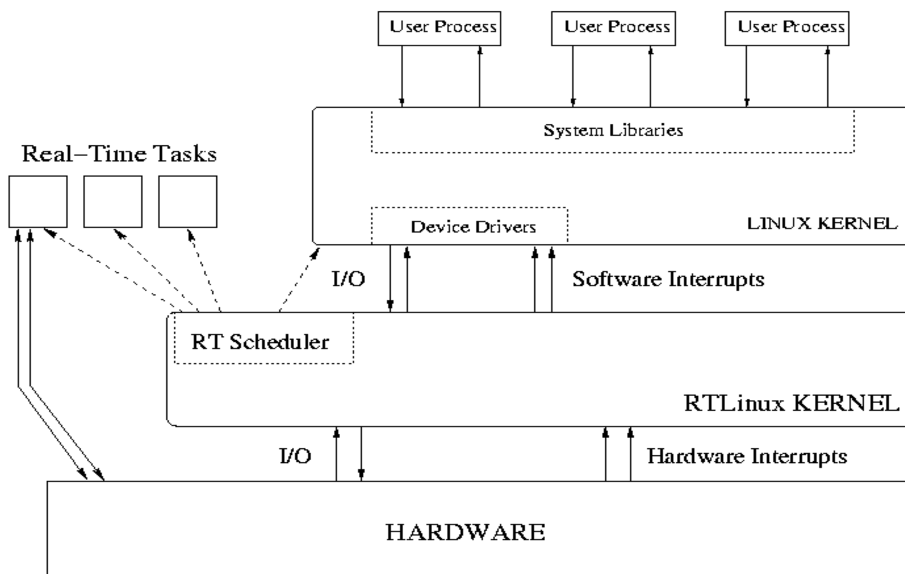


Figure 3.2: *The role of the RT Linux kernel.*

The RTCore itself is nonpreeptible, but unpredictable delays are eliminated by its small size and limited operations.

Often it is only a small part of a real-time program that has to run in hard real-time, while the timing demands on the other tasks do not categorize as hard real-time. In this case it is useful to be able to split the program into two parts: the hard real-time which runs as a module in the RT kernel, and the soft real-time that runs under standard Linux. This is precisely what RTLinux offers. The RT modules can communicate with standard Linux processes using either FIFOs or shared memory.

The RTLinux scheduler is a priority driven scheduler that either uses Round Robin or *FIFO* algorithms. It supports preemption and priority inheritance. The normal scheduler, FIFO, always lets the highest priority thread execute, and if more than one thread has the same priority the FIFO-law is put into practice. With the Round Robin scheduler, the highest priority group of threads execute for a certain time quantum each until all threads are blocked, then lower priority groups get to execute. Other schedulers may be implemented, and, during compilation, it is possible to choose to include an *earliest deadline first* scheduler (EDF). Unfortunately I did not have time to test this one. The priority range is 0 (lowest) to 100000 (highest). There is no limit in the number of running threads, but one should be aware of that the scheduling cost is proportional to the number of threads. Current scheduler code is designed to

efficiently handle a low number of threads (around 10).

An installation guide for RTLinux/Free is written as part of the user manual enclosed to the thesis. For more information about RTLinux, read for instance *The RTLinux Manifesto* by Victor Yodaiken or find information on the FSM Labs Inc. webpage (www.fsmlabs.com).

3.2 Latency Test for RTLinux

To test whether the periodic latency was improved with RTLinux, a similar program used in the test of standard Linux was implemented and executed. During the execution the system was under heavy load; a large file was being copied, many programs were being started at the same time and a rather large compilation was also executed during the 60 second execution time. Figure 3.3 shows the highly improved distribution for the period latency.

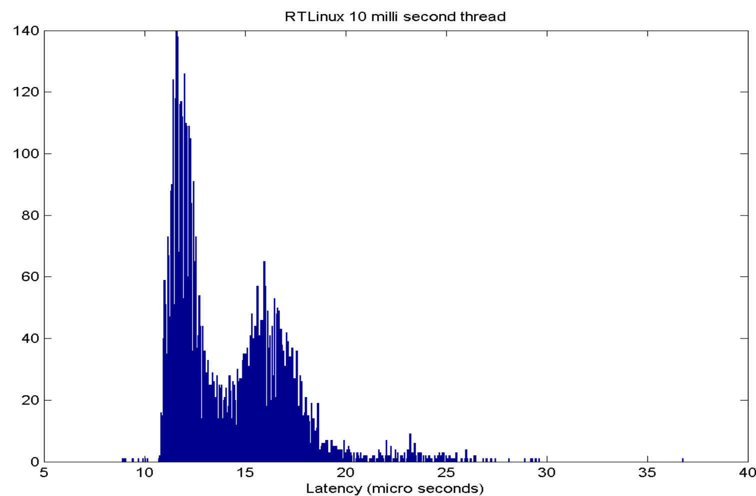


Figure 3.3: *Periodic latency distribution in* µs *for a* 10 *ms RTLinux thread during heavy system load.*

- RTLinux with load
  - Minimal latency: 8.9 µs
  - Maximal latency: 36.8 µs
  - Mean latency : 14.2 µs
  - Standard deviation: 2.85 µs

These numbers are very nice from a real-time point of view. According to specification the worst case periodic latency is around 40 µs using the current equipment, and not surprisingly the maximal latency during the 60 second execution was below this value. Notice that, due to the nature of Linux being a low priority task in RTLinux, the system load during the test does in fact not effect the result. It is possible that

the results would have been a little bit worse if the test would have been performed with many parallel real-time threads since the scheduling time is proportional to the number of threads.

Chapter 4

The Process

4.1 Pendulum Description

The pendulum at UPC (figure 4.1), delivered by the Polish company Inteco[1], is a wall-mounted, regular one where the pendulum is mounted on a cart in such a way that the pole can swing freely only in the vertical plane. The cart moves on a straight track, attenuated by a DC-motor using a belt for transmission. For the sake of balance, two identical pendulum rods and loads are attached to the cart according to figure 4.2. Two optical incremental sensors are connected to the system; one for the pendulum angle, and one for the cart position. Using these signals a computer may compute and set signals to the DC motor so that desired control is achieved.

The pendulum is a process often used in control problems due to its nature of being both a stable process (pendulum down) and an unstable process (pendulum up). Furthermore, it is highly non-linear and often involves practical non-linear phenomena such as friction, saturation and limited displacement for the cart[2].

4.2 Hardware Communication

The pendulum system (pendulum, cart, rack, sensors) is connected to the computer by a combination of a PWM[3] and cable interface (I will refer to it only as PWM), which then is connected to a RT-DAC4/PCI card in the computer. On the PCI card there is an FPGA[4] which converts the optical sensors' pulses to a signal which may be used in the computer. The main purpose for the PWM is to set a PWM-signal to the DC-motor which is proportional to the signal from the computer. The FPGA holds this signal constant until it is changed by the computer, i.e. it uses *zero order hold*.

The optical sensor for the angle measures 4096 pulses on a complete revolution for the pendulum, and hence the resolution in angle is $2\pi/4096$ radians. Furthermore, the optical sensor for the cart measures 18877 pulses on the length of the track which is 1.00 m long. The resolution in cart position is therefore $1/18877 = 5.30 \cdot 10^{-5}$ m. Both signals are delivered as incremental unsigned two-byte values. The signal to the

[1] www.inteco.cc.pl

[2] This is not true when the pivot point moves in a circular orbit. This type of pendulum is called *Furuta pendulum* and is also common in control research.

[3] Pulse Width Modulator

[4] Field Programmable Gate Array

Figure 4.1: *Photo of the actual pendulum.*

DC motor must be set by using signals from 0 (min) to 1024 (max), which implies a resolution in the output signal of $Output_{max}/1024$.

4.3 RT-DAC4/PCI Drivers

RT-DAC4/PCI is a multifunctional programmable PCI-card which is very suitable for control purposes. Unfortunately it was only delivered with drivers for use under Windows 95/98/NT/2000 and despite attempts to find drivers for Linux using various sources (internet, PCI providers) there were none to be found. It is of course absolutely conclusive to be able to communicate with the pendulum in order to control it, so another solution had to be found. The solution was to write the drivers needed. This was not an easy task since I was totally inexperienced in this area before. Despite this, after having found information of how to write Linux device drivers and knowing how the PCI card worked I managed to get the drivers working. They have been updated with small changes along the way, and there have not been any problems using them at all.

According to technical specifications for the PCI-card, the output demanded by the computer will be redirected to the PWM within 6 µs, and signals sent from the PWM to the PCI-card will be available for reading within 1.6 µs.

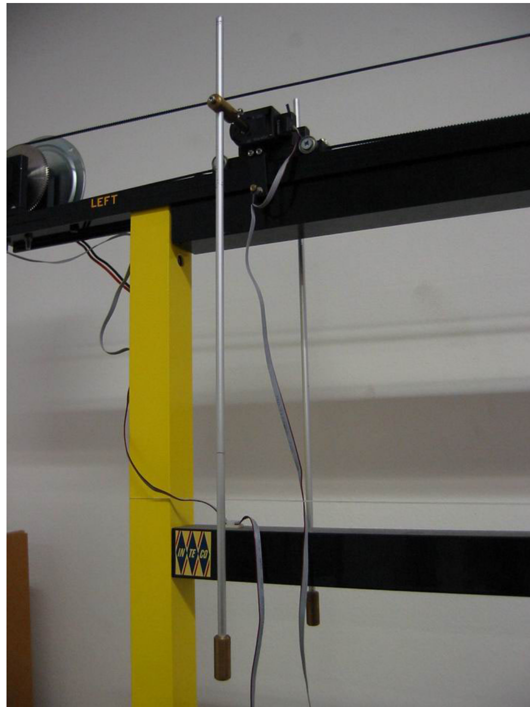


Figure 4.2: *Photo of the pendulum rod and load.*

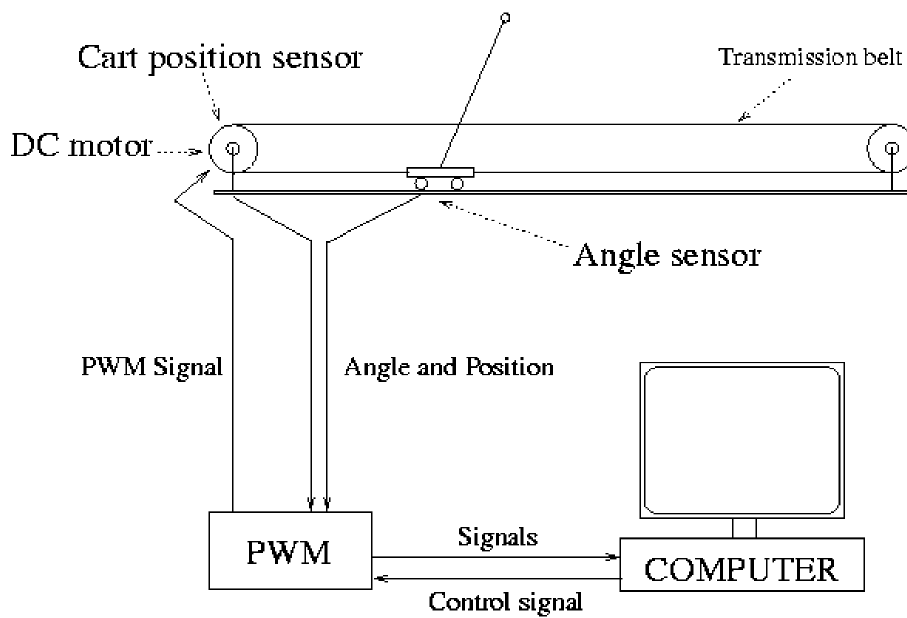


Figure 4.3: *Schematic picture of the total system.*

Chapter 5

The Program

The computer used for control was a Pentium III 1 GHz with 256 MB SDRAM running Linux Debian 3.01 on top of RTLinux/Free 3.2 pre2.

5.1 Program Overview

The program is roughly made up by three different parts:

The Driver communicates with the PCI card in hard real-time. The driver registers itself as a new device on which it is possible to perform `read/write` methods that are executed with the same priority as the calling thread. It is also possible to change other settings according to the API which follows.

The Kernel Modules execute in hard real-time. This is where the main control-loop is implemented. All reading of input signals, computation of output signal and later on writing of these are performed in hard real-time.

The Graphical User Interface (GUI) executes in soft real-time in standard Linux. Its core is a periodic loop that handles the communication with the hard real-time using a number of FIFOs[1]. This is where the state information is displayed and the user can operate the control program.

5.2 RT-DAC4/PCI Drivers

The drivers for the PCI card had to be written to be used with RTLinux. The PCI card contains an FPGA that converts the pulses from the optical sensors to two 16 bits incremental signals written in the memory of the card. It also reads the output signal from a memory card and sends this information to the PWM for further treatment.

Standard Linux drivers register a device under `/dev` to which a process can open a connection and perform standard operations, i.e. `read, write, open, close`, etc. The driver for the RT-DAC4/PCI card does the same thing. When inserted in the module, using `insmod`, the driver registers a device `/dev/pms`[2] to which later on the module for control may open a connection. Using this connection the control module

[1]Byte oriented pipe which is implemented as a FIFO (First In - First Out).
[2]The device name happens to be my unfortunate initials.

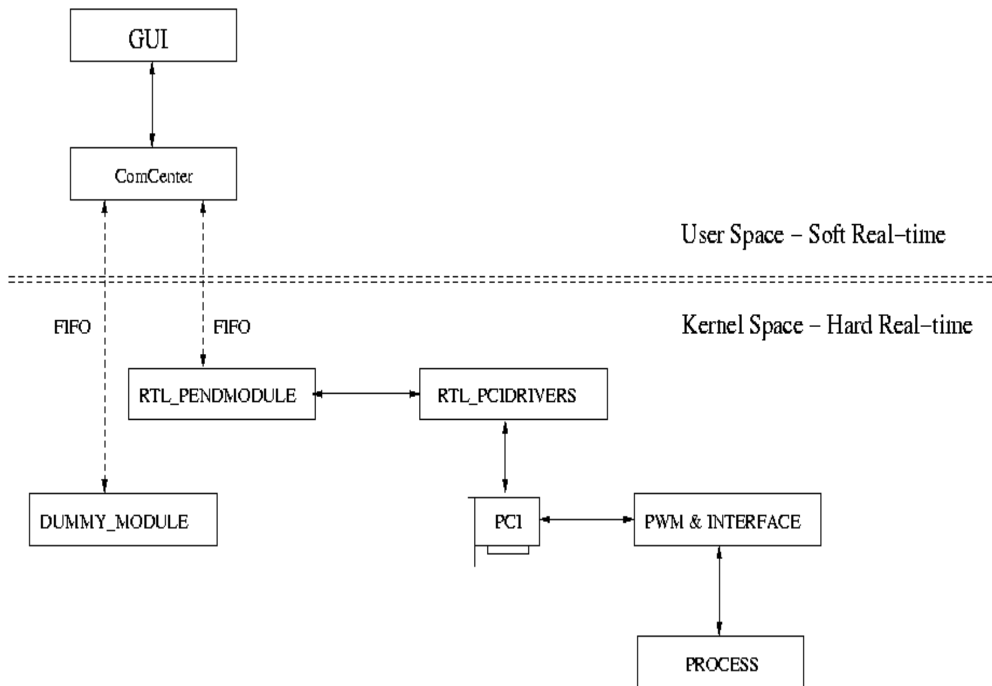


Figure 5.1: *Schematic program overview.*

may read the states and set signals to the DC-motor. It may also use some additional features as resetting the sensors and changing the frequency of the PWM. The driver is written in C and offers the following API (a more complete API list is provided in the Appendix):

```
open(...)       // Open a connection to the device

close(...)      // Close the device

read(...)       // Read position of cart and angle of pendulum

write(...)      // Set signal to the DC motor

ioctl(..., 1)  // Reset the incremental encoders

ioctl(..., 2)  // Set new frequency for the PWM
```

5.3 The Kernel Modules

The real-time tasks run in kernel space (runlevel 0). This is both good and bad. Response times in kernel space are very short. Interrupt response and task switching times are normally below 10 µs, but the bad thing is that the protection in kernel space is little or none, and that errors in a real-time task may bring down the whole system. Unfortunately I experienced this many times during the implementation of

the control module. The kernel modules are also written in C and are being started and stopped using the commands `insmod` and `rmmod`.

The structure of the control module is as follows:

```
init_module()    // Initialize the module. Create and start the thread,
                    the semaphores, etc.

cleanup_module() // Remove the module from the kernel. This is where
                    any necessary ''cleanup'' is done.

thread_code()    // Code assiciated with the started thread
                    which contains the main control loop.

handler_code()   // Code associated to certain FIFOs. This is being
                    launched whenever anything is beeing written to
                    the FIFOs.
```

5.3.1 Calculation of Derivatives

The module uses the signals for the angle and the cart position to calculate their respective derivatives. The method used is *Backward Euler*, i.e. the derivatives are

$$\begin{aligned}\dot{\theta}(k) &\approx \frac{\theta(k)-\theta(k-h)}{h}\\ \dot{x}(k) &\approx \frac{x(k)-x(k-h)}{h}\end{aligned}$$

The variable h is the true system time between two consecutive samples and not the "expected" sampling time. Depending on the sampling time different resolutions in the derivatives will be available. With a sampling rate of 100 Hz the resolution in angular velocity is $2\pi/4096/0.01 = 0.153$ radians per second and the resolution in cart velocity is $1/18877/0.01 = 0.0053$ meters per second.

5.3.2 Faster Floating Point to Integer Conversions

The use of floating point operations is avoided as much as possible in any code that has speed as a main goal, but to not use floating point operations in the controller would make the code highly complicated and very difficult to overview, so not to use floating point operations at all was therefore not an option. However, what can be improved is the cast from a double (or float) to an int, which for example has to be done to set the calculated control signal to the DC motor. Use of standard C cast, i.e.

```
integer_value = (int) double_value;
```

invokes among other things the assembler function `fldcw` (FPU load control word). Whenever the FPU (Floating Point Unit) encounters this instruction it flushes its pipeline and loads the control word before continuing operation. The FPUs of modern CPUs like the Pentium III, Pentium IV and AMD Athlons rely on deep pipelines to achieve higher peak performance.

A solution is to instead use the operation

```
integer_value = lrint(double_value);
```

which instead of truncating the double value (which the standard cast does) rounds it according to standard rounding rules. The (possible) loss of precision is accepted in this case. The change from standard C cast to instead use `lrint` reduced the average computation time by 3 %.

5.3.3 Input-Output Latency and Computation Time

The periodic latency shows almost exactly the same distribution as for the test program used for testing RTLinux, i.e. it has a worst case value of less than 40 µs.

The core functionality of the control module is to run the pseudo-loop

```
while(true) {
  readStates();
  calculateOutput();
  setOutput();
  updateStates();
  misc();
}
```

The time from the reading of input signals (states) to the control signal is set to the DC motor is often referred to as *input-output latency* (IO latency). If this time, T_{IO}, is not taken into account when designing the controller (which is not the case), it is important to make it as short as possible, since any time delay reduces the phase margin. This is done by optimizing the code as much as possible, by for instance updating state information after the output has been set and precalculating loop factors when they are changed only. The values for the IO latency during an execution which involved all modes was distributed according to figure 5.2. Using the maximum value, the IO Latency is below 0.65 % of the sampling time at 100 Hz.

- Input-Output Latency
  - Minimal IO Latency: 42.5 µs
  - Maximal IO Latency: 64.6 µs
  - Mean IO Latency: 45.6 µs
  - Standard deviation: 4.2 µs

The time for a complete revolution in the loop, i.e. the time from that the thread wakes up to that it goes back to sleep, is called *computation time*, T_C. This time is of importance when scheduling many tasks which are to run in parallel, and it is in general of interest to keep this short. This time is T_{IO} plus time for execution of additional code, e.g. update states, write to user space, and write to log. Everything that can be done in user space should be done there. For instance, the recording of data is not written to a file in the module, but instead dumped in a FIFO so that the user space code writes it to a file when the RT kernel is idle. The distribution for T_C during an execution which involved all modes plus recording of data was as follows. Notice that during none of the tests, the possibility to use cache memory, etc have been disabled. Therefore the results can not be used as ”true” worst case times, and their purpose are only for comparison.

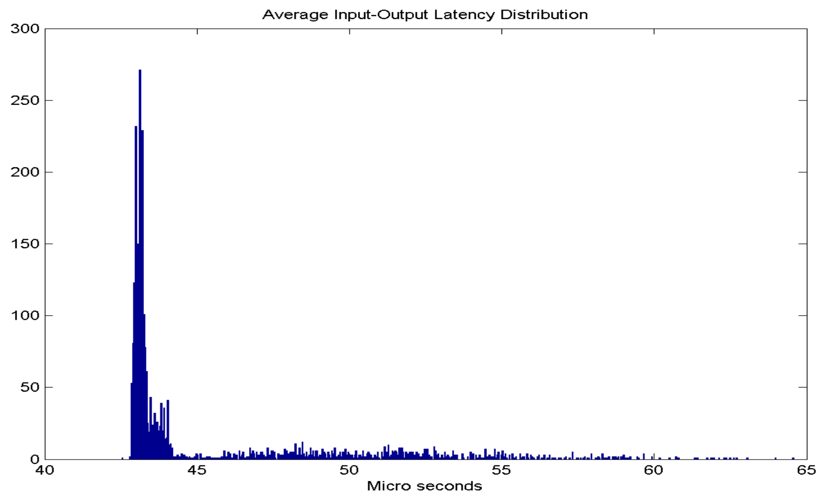


Figure 5.2: *Average Input-Output latency distribution.*

- Computation Time
  - Minimal Computation Time: 52.8 µs
  - Maximal Computation Time: 106.0 µs
  - Mean Computation Time: 60.5 µs
  - Standard deviation: 11.0 µs

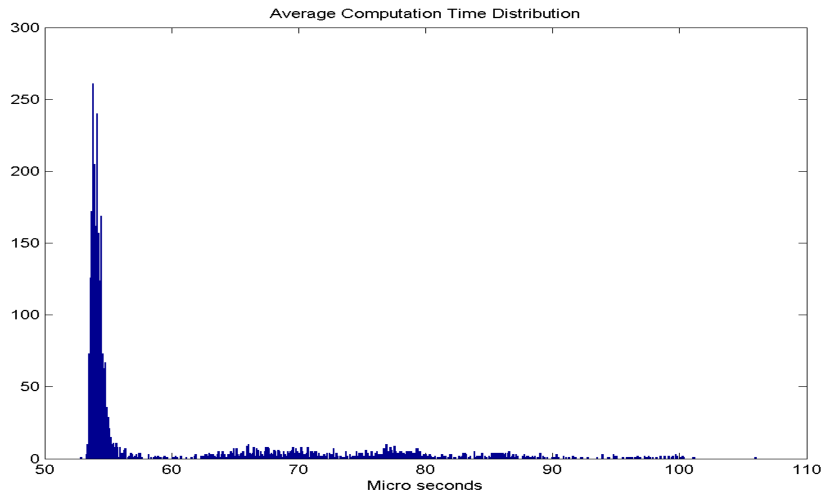


Figure 5.3: *Average computation time distribution.*

If the control module was the only running RT thread, it would be possible to sample, with equally strict timing constraints, at sampling rates as high as 4.8 kHz

before the CPU load would reach 50%. I believe this is impossible to with a normal operating system.

5.3.4 Dummy Module

Besides the control module, a module for dummy real-time threads was written. This was to be able to disturb the real-time control program in a deterministic way. Using this module, hard real-time threads with various period time and computation time may be started and stopped. It is also possible to set different priorities on the started threads. Chapter 8 gives some examples of tests. With minor changes in the code it is possible to use these threads to test different scheduling algorithms. However, even though this is an interesting topic, it was not part of this project so no tests have been done in this area.

5.4 The Java Program

When designing the GUI, flexibility was a main goal. Personally I like to choose for myself whether to have a lot of information on the screen or not. The GUI (see Appendix B.3) is written in JAVA Swing, and is made up by several parts:

The main control window connects to the kernel modules and starts/stops them. It is designed for state feedback in up/down position, but can easily be changed for other controllers. The state feedback vector can be set for the controller in up-mode and for the controller in down-mode. The reference signal for the cart is as default in the middle, but can also be changed.

The State and Control Viewer gives continuous information about the four states and the control signal. The user may freeze any number of plotters and save the image to disk. The user may also freeze all current plotters at the same time and save them as one single image. The axis of the plotters can be modified while running and it is possible to choose any composition of displayed plotters. The time used for the time coordinates is the time used by the RT scheduler. To produce nice and smooth plots, buffers are used. I also made a slight change in the class library `se.lth.control.plot`[3] which allows more control of when the plotters are invoked. Besides the obvious signals, the cart reference is plotted in red as well as the calculated output signal, which is plotted in the same plotter as the friction compensated signal (black).

The RT Thread Table gives the user the possibility to start different real-time threads. These threads are dummy threads with a period time, computation time and a priority. This feature gives the possibility to measure control performance in the case when the controller is disturbed in a controlled way.

The Control Settings set a number of control related parameters, e.g. sampling time, static friction, swing-up gain, etc.

The Device Settings contains information to which devices the program connects. It is useful if some devices are busy and need to be changed.

[3] Developed at the Department of Automatic Control at LTH.

The Data Recorder records full state information and information about latency, current priority, reference signal, computation time and control signal during a period of max 60 seconds with an optional step of size Y cm in the cart reference at time Z.

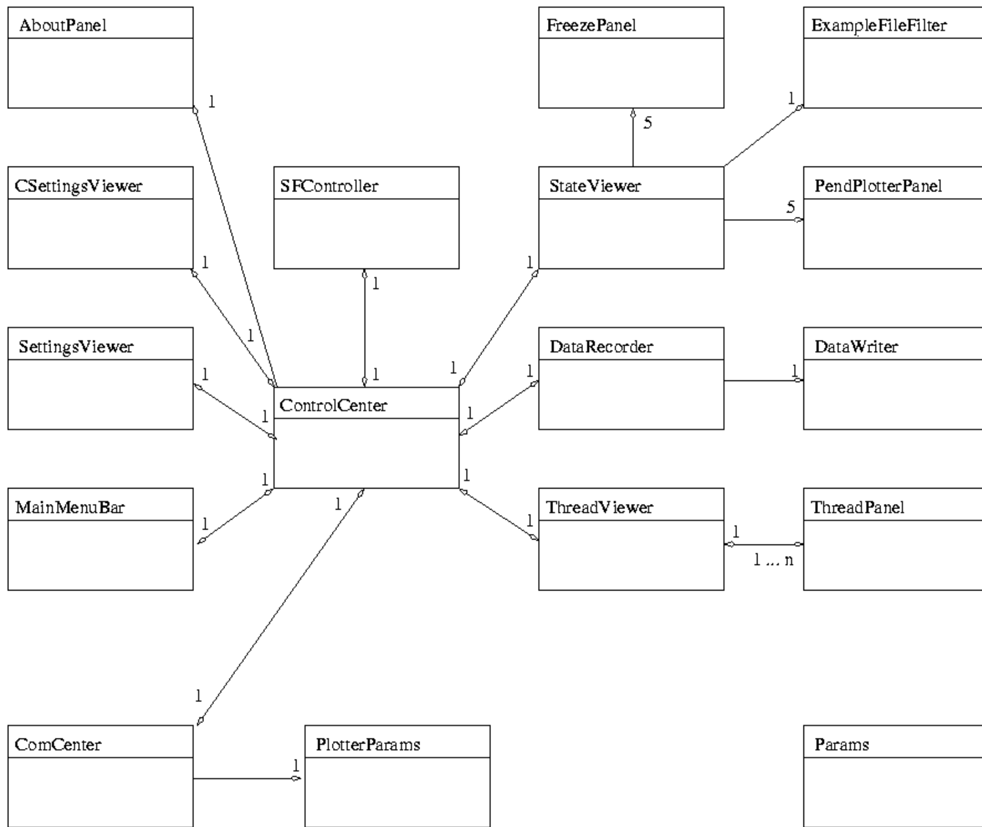


Figure 5.4: *UML diagram over Java Program*

The main functionality for the Java program is to read the FIFOs containing information about the states, control signal, etc. This is done periodically with a frequency of 33 Hz. To avoid a visual delay due to the lower frequency than the control module, the loop tries to read up to 5 data points every time. With these settings, sampling rates up to 165 Hz (6 ms sample time) are possible in the control module without visual delay. It is of course easy to change these values if faster sampling is required. To save CPU the plotters only print 50 data points per second. More information of how to use the program and how to achieve additional information (Javadoc) is found in Appendix A.

5.5 Putting Everything Together

As mentioned in chapter 3, communication between the real-time kernel and the soft real-time user space can either be done using FIFOs or shared memory. I choose to

use FIFOs for communication both from user space to the real-time kernel and vice versa.

The rather simple protocol for communication from user space to the real-time kernel has an upper limit of 2^{15} parameters set with double precision (64 bits), and possibility to use 2^{15} different modes. The protocol from the real-time module to user space also has an upper limit of 2^{15} parameters but only with integer precision (32 bits). All parameters are protected by semaphores since there is a possibility that they are being accessed at the same time by the control module and the handler connected to the FIFO that the user space writes to.

The dummy module has an upper limit of 2^{15} possible threads each having 2^{15} parameters of integer precision. It is however unlikely that any of the modules will require many more parameters than they currently use. All parameters are protected by semaphores, and semaphores are also used to start/stop the real-time threads as well as the control module.

As default, Java uses *network byte order*[4] while C uses *host byte order*[5], so any communication must consider this.

Since the signal for angle measurement is incrementing and delivered as a signed byte, the signal will be discontinuous at some point. An extra function (invoked when reading the states) converts the signal to a continuous one.

When the control module is running at the sampling rate 100 Hz and the Java program is running without the plotters, the CPU load according to `top` is less than 4%. Depending on the number of plotters running the CPU load may increase to up to 95%.

Besides from a randomly appearing bug when going from ”OFF No Plotters” to ”OFF Run Plotters”, the program runs stable and without any problems in the control. It does not appear to have any memory leaks and has been controlling the pendulum for full days without problems. The Java program is somewhat less stable, and the graphics show some unpredictable behaviour occasionally. This is however not a problem that effects the control. Due to the separation of the different software parts, it is actually possible to close the Java program by force (ctrl + c), restart it and re-connect to the module without disturbing the ongoing control.

When changing the parameters while controlling the pendulum, there is a risk of increasing the different latencies the same amount of time as the protecting semaphores may be locked by the changing code. This time is however extremely small.

[4] Also known as Big Endian

[5] Also known as Small Endian

Chapter 6

Model of Pendulum

To later be able to control the pendulum, it is important to have a description of the system dynamics. These dynamics are often described as a number of transfer functions from various inputs to outputs. There are at least two ways of finding these transfer functions — either theoretically, or by system identification. I personally like the theoretical approach, so this report uses the first method. Furthermore, I believe that the theoretical derivation gives a greater understanding of the process then by using system identification. Using system identification may however reveal behavior caused by unmodelled dynamics.

6.1 Mathematical Model of Pendulum

Parameter	Description
x_p	X-coordinate for center of gravity of pendulum
y_p	Y-coordinate for center of gravity of pendulum
x_c	X-coordinate for center of gravity of cart
y_c	Y-coordinate for center of gravity of cart
y_{pp}	Y-coordinate for pivot point
l	Distance from pendulum center of gravity to pivot point
θ	Pendulum angle from positive Y-direction
m	Mass of pendulum (load and rod)
M	Cart mass
F	Force applied to cart
V	Vertical reaction force from pendulum
H	Horizontal reaction force from pendulum
I_p	Moment of inertia for pendulum with respect to pivot point
f_p	Viscous friction for pendulum at pivot point
f_c	Dynamic friction for cart

Basic trigonometry applied to figure 6.1 shows that

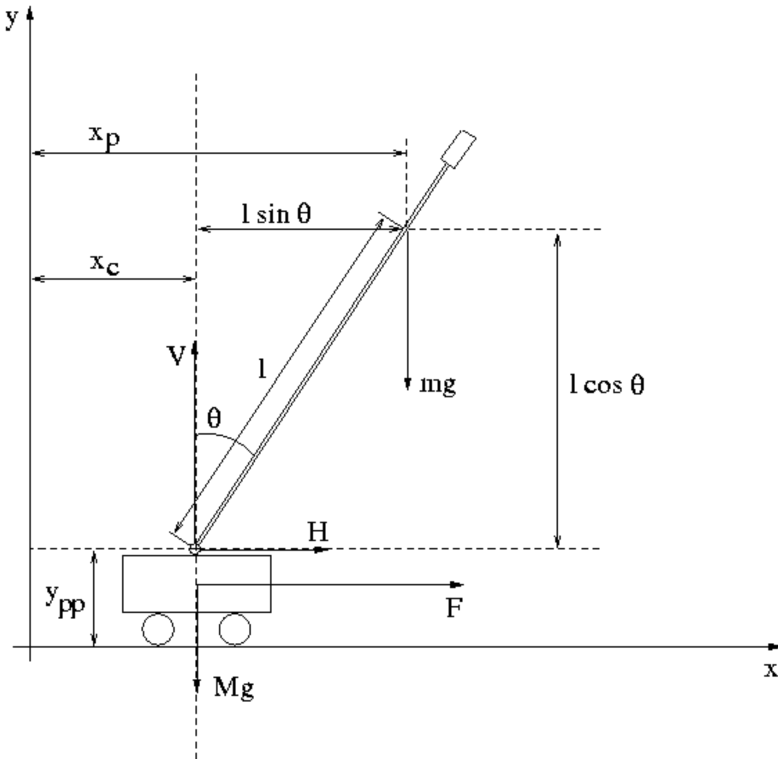


Figure 6.1: *Schematic picture of the pendulum and cart.*

$$\begin{aligned} X_c &= x \\ \dot{X}_c &= \dot{x} \\ \ddot{X}_c &= \ddot{x} \\ Y_c &= y = 0 \\ \dot{Y}_c &= 0 \\ \ddot{Y}_c &= 0 \end{aligned}$$

$$\begin{aligned} X_p &= x + l\sin\theta \\ \dot{X}_p &= \dot{x} + l\dot{\theta}\cos\theta \\ \ddot{X}_p &= \ddot{x} + l\ddot{\theta}\cos\theta - l\dot{\theta}^2\sin\theta \\ Y_p &= Y_{pp} + l\cos\theta \\ \dot{Y}_p &= -l\dot{\theta}\sin\theta \qquad (\dot{Y}_{pp} = 0) \\ \ddot{Y}_p &= -l\ddot{\theta}\sin\theta - l\dot{\theta}^2\cos\theta \end{aligned}$$

Sum of forces in the X-direction:

$$F = H + M\ddot{X}_c + \dot{X}_c f_c \qquad \text{where} \qquad H = m\ddot{X}_p \tag{6.1}$$

Sum of forces in the Y-direction:

$$V = mg + m\ddot{Y}_p \Leftrightarrow V = mg - ml(\ddot{\theta}\sin\theta + \dot{\theta}^2\cos\theta) \tag{6.2}$$

since $\dot{Y}_c = \ddot{Y}_c = 0$.
Sum of torques around the pivot point X_c, Y_{pp}:

$$\sum \tau = I_p\ddot{\theta} \tag{6.3}$$

Equation 6.1 yields

$$\begin{aligned} F &= M\ddot{X}_c + f_c\dot{X}_c + m\ddot{X}_p \\ &= M\ddot{x} + f_c\dot{x} + m(\ddot{x} + l\ddot{\theta}\cos\theta - l\dot{\theta}^2\sin\theta) \\ &\Updownarrow \\ (M+m)\ddot{x} &= F - ml(\ddot{\theta}\cos\theta + \dot{\theta}^2\sin\theta) - f_c\dot{x} \end{aligned}$$

Equation 6.3 yields

$$\begin{aligned} I_p\ddot{\theta} + f_p\dot{\theta} &= Vl\sin\theta - Hl\cos\theta \\ &= (mg - ml\ddot{\theta}\sin\theta - ml\dot{\theta}^2\cos\theta)(l\sin\theta) - m(\ddot{x} + l\ddot{\theta}\cos\theta - l\dot{\theta}^2\sin\theta)(l\cos\theta) \\ &= m(gl\sin\theta - l^2\ddot{\theta}(\sin^2\theta + \cos^2\theta) - \ddot{x}l\cos\theta) \\ &= m(gl\sin\theta - l^2\ddot{\theta} - \ddot{x}l\cos\theta) \qquad \textit{since} \qquad \sin^2\theta + \cos^2\theta = 1 \\ &\Updownarrow \\ (I_p + ml^2)\ddot{\theta} &= mgl\sin\theta - ml\ddot{x}\cos\theta - f_p\dot{\theta} \end{aligned}$$

The dynamic relations for the cart position and the pendulum angle are thus

$$\ddot{x} = \frac{F - ml(\ddot{\theta}\cos\theta + \dot{\theta}^2\sin\theta) - f_c\dot{x}}{(M+m)} \tag{6.4}$$

$$\ddot{\theta} = \frac{mgl\sin\theta - ml\ddot{x}\cos\theta - f_p\dot{\theta}}{(I_p + ml^2)} \tag{6.5}$$

6.2 Equilibrium Points

The non-linear equations 6.4 and 6.5 describe the system with the four states $x, \dot{x}, \theta, \dot{\theta}$. An equilibrium is defined as a point (or curve) at which all states remain unchanged, i.e. their derivatives are zero. It is interesting to investigate whether such a point exists when the system's input signal is zero ($F = 0$);

After $\ddot{x}$ in 6.5 has been substituted for by 6.4 and vice versa for $\ddot{\theta}$, a new system of equations appears:

$$\begin{aligned} \beta\ddot{x} &= (I_p + ml^2)(F + ml\dot{\theta}^2\sin\theta - f_c\dot{x}) - ml\cos\theta(mgl\sin\theta - f_p\dot{\theta}) \\ \beta\ddot{\theta} &= (M+m)(mgl\sin\theta - f_p\dot{\theta}) - ml\cos\theta(F + ml\dot{\theta}^2\sin\theta - f_c\dot{x}) \\ \beta &= (M+m)(I_p + ml^2) - (ml\cos\theta)^2 \end{aligned}$$

The conditions $\dot{x} = 0$, $\dot{\theta} = 0$ and $F = 0$ give

$$\begin{aligned} \beta\ddot{x} &= -ml\cos\theta(mgl\sin\theta) = 0 \\ \beta\ddot{\theta} &= (M+m)(mgl\sin\theta) = 0 \end{aligned}$$

Since $\beta \neq 0$ the only points satisfying these equations are $\theta = 0$ and $\theta = \pi$. This result makes sense; the pendulum can hang straight down and remain here (states not changing) and the pendulum can also (at least in theory) stand straight up and remain

here if nothing disturbs it. Notice the fact that there are not any restrictions on x, the cart position. The pendulum may find it self in an equilibrium anywhere on the horizontal bar.

Without the conditions $\dot{x} = 0$ and $F = 0$ there exist infinite many equilibrium trajectories. It is in theory possible to keep the pendulum at any angle by applying a constant acceleration. Each possible angle would then result in a constant acceleration and hence a trajectory for $\dot{x}$ to follow. The pendulum in this report is very limited in terms of displacement of x, so the only equilibrium points of interest are $(x, \dot{x}, \theta, \dot{\theta}) \in \{(x, 0, 0, 0), (x, 0, \pi, 0)\}$.

6.3 Linearization in Up-position

To be able to use normal stabilization theory equation 6.4 and 6.5 must be linearized. In up-position it is natural to chose the equilibrium point $(\theta_0 = 0, \dot{\theta}_0 = 0)$ and linearize around this. This is the region in which the stabilizing control will be applied, and if this works satisfying the deviation from $(\theta_0 = 0, \dot{\theta}_0 = 0)$ is small, and hopefully the linearization is therefore valid.

Define θ as the deviation from $\theta_0 = 0$. This does not differ from the previous definition due to the choice of linearization point. Linearization using Taylor series results in:

$$\sin\theta \approx \sin\theta_0 + \frac{d}{dt}\sin\theta_0\bigg|_{\theta_0=0} \cdot \theta + \varepsilon \approx 0 + 1 \cdot \theta = \theta \tag{6.6}$$

$$\cos\theta \approx \cos\theta_0 + \frac{d}{dt}\cos\theta_0\bigg|_{\theta_0=0} \cdot \theta + \varepsilon \approx 1 + 0 \cdot \theta = 1 \tag{6.7}$$

where ε means higher order terms. Around $(\theta_0 = 0, \dot{\theta}_0 = 0)$ the linearized versions of equation 6.4 and 6.5 therefore are

$$\ddot{x} = \frac{F - ml\ddot{\theta} - f_c\dot{x}}{(M + m)} \tag{6.8}$$

$$\ddot{\theta} = \frac{mgl\theta - ml\ddot{x} - f_p\dot{\theta}}{(I_p + ml^2)} \tag{6.9}$$

6.4 Linearization in Down-position

In down-position the control will be focused in the area of $(\theta_0 = \pi, \dot{\theta}_0 = 0)$ (the other equilibrium point), and it therefore makes sense to linearize around this point. Define θ as the deviation from $\theta_0 = \pi$.

As above, linearization using Taylor series results in:

$$\sin\theta \approx \sin\theta_0 + \frac{d}{dt}\sin\theta_0\bigg|_{\theta_0=\pi} \cdot \theta + \varepsilon \approx 0 - 1 \cdot \theta = -\theta \tag{6.10}$$

$$\cos\theta \approx \cos\theta_0 + \frac{d}{dt}\cos\theta_0\bigg|_{\theta_0=\pi} \cdot \theta + \varepsilon \approx -1 + 0 \cdot \theta = -1 \tag{6.11}$$

Around ($\theta_0 = \pi, \dot{\theta}_0 = 0$) the linearized versions of equation 6.4 and 6.5 therefore are

$$\ddot{x} = \frac{F + ml\ddot{\theta} - f_c\dot{x}}{(M+m)} \tag{6.12}$$

$$\ddot{\theta} = \frac{-mgl\theta + ml\ddot{x} - f_p\dot{\theta}}{(I_p + ml^2)} \tag{6.13}$$

6.5 State Space Description

To get equation 6.8 and 6.9 into valid linear state space, and get the state space description in up-position, they must be functions of lower order terms only. Hence $\ddot{x}$ must be substituted for in 6.9 by 6.8 and vice versa for $\ddot{\theta}$.

Equation 6.8 in 6.9 result in

$$\begin{aligned}
\ddot{\theta}(I_p + ml^2)(M+m) &= (M+m)(mgl\theta - f_p\dot{\theta}) - ml(F - ml\ddot{\theta} - f_c\dot{x}) \\
&\Leftrightarrow \\
\ddot{\theta}((I_p + ml^2)(M+m) - m^2l^2) &= (M+m)(mgl\theta - f_p\dot{\theta}) - ml(F - f_c\dot{x}) \\
&\Leftrightarrow \\
\ddot{\theta} &= (M+m)(\frac{mgl\theta}{\alpha} - \frac{f_p\dot{\theta}}{\alpha}) + ml(\frac{f_c\dot{x}}{\alpha} - \frac{F}{\alpha})
\end{aligned}$$

after introducing $\alpha = (I_p + ml^2)(M+m) - m^2l^2$. In the same way 6.9 in 6.8 results in

$$\begin{aligned}
\ddot{x}(M+m)(I_p + ml^2) &= (I_p + ml^2)(F - f_c\dot{x}) - ml(mgl\theta - ml\ddot{x} - f_p\dot{\theta}) \\
&\Leftrightarrow \\
\ddot{x}((M+m)(I_p + ml^2) - m^2l^2) &= (I_p + ml^2)(F - f_c\dot{x}) - ml(mgl\theta - f_p\dot{\theta}) \\
&\Leftrightarrow \\
\ddot{x} &= ml(\frac{-mlg\theta}{\alpha} + \frac{f_p\dot{\theta}}{\alpha}) + (I_p + ml^2)(\frac{-f_c\dot{x}}{\alpha} + \frac{F}{\alpha})
\end{aligned}$$

Using the state vector

$$X = [x, \dot{x}, \theta, \dot{\theta}]$$

the linearized system in up-position is described by

$$\begin{cases} \dot{X} = A_{up}X + B_{up}U \\ Y = CX + DU \end{cases}$$

with

$$A_{up} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{(I_p+ml^2)f_c}{\alpha} & -\frac{m^2l^2g}{\alpha} & \frac{mlf_p}{\alpha} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{mlf_c}{\alpha} & \frac{(M+m)mgl}{\alpha} & -\frac{(M+m)f_p}{\alpha} \end{bmatrix}$$

$$B_{up} = \begin{bmatrix} 0 \\ \frac{(I_p+ml^2)}{\alpha} \\ 0 \\ -\frac{ml}{\alpha} \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad D = 0 \quad U = F$$

In a similar manner it is possible to show that the state space description in down-position is

$$\begin{cases} \dot{X} = A_{down}X + B_{down}U \\ Y = CX + DU \end{cases}$$

with

$$A_{down} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & -\frac{(I_p+ml^2)f_c}{\alpha} & -\frac{m^2l^2g}{\alpha} & -\frac{mlf_p}{\alpha} \\ 0 & 0 & 0 & 1 \\ 0 & -\frac{mlf_c}{\alpha} & -\frac{(M+m)mgl}{\alpha} & -\frac{(M+m)f_p}{\alpha} \end{bmatrix}$$

$$B_{down} = \begin{bmatrix} 0 \\ \frac{(I_p+ml^2)}{\alpha} \\ 0 \\ \frac{ml}{\alpha} \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad D = 0 \quad U = F$$

6.6 Finding the Parameters

The dynamics of a model of the system is described in algebraic terms. To be able to use the model all these algebraic terms must be given numerical values. Some of the parameters were given in the technical specifications of the pendulum, while other call for experimental determination. Parameters that were given were the mass of the cart, pendulum rod, and the pendulum load.

6.6.1 Center of Gravity of Pendulum

The pendulum is made up of a rigid pole of aluminium and has a load attached at the end. By using math and physics it is possible to describe the center of gravity, but the easiest way to find the center of gravity is to use a simple experiment:

With the pendulum dismounted from the system, balance it horizontally on a sharp edge and find the point of balance. This point is the location of the center of mass.

6.6.2 Moment of Inertia for Pendulum

The moment of inertia with respect to a point can be calculated by taking careful measurement of the object and knowing the density of the containing parts. As in the previous example, there is an easier way.

When the pendulum swings back an forward with a small angle, it may be treated as a simple harmonic system, i.e. the force that brings the pendulum back to its equilibrium is proportional to the distance from this point. Such a system is said to move in a simple harmonic motion producing an oscillation with period τ. The expression for the period is

$$\tau = 2\pi\sqrt{\frac{I}{mgD}}$$

with I being the moment of inertia with respect to the pivot point, m the equivalent mass of the pendulum (pole + load) and D being the distance from the pivot point to the center of gravity of the pendulum. Rearranging the expression yields

$$I = \frac{\tau^2 mgD}{4\pi^2}$$

The only thing need to be found is thus the period time. This is simply done by counting the number of periods during a certain amount of time. If this time is chosen sufficiently big, this will give a very good estimate of the moment of inertia. Notice that the period time approximately[1] remains the same even though it will decrease in amplitude due to friction.

During 60 seconds the pendulum made 49 periods, yielding $T = 1.225$. This gives an estimated value of the moment of inertia for the pendulum with respect to the pivot point of 0.0116 kg/rad.

6.6.3 Pendulum Friction

If the pendulum hangs straight down and is pushed away from its equilibrium, it will exhibit simple harmonic motion. The decay of the oscillation is described by the function $f(t) = ae^{-bt}$, with a as an initial condition and $b \geq 0$ depending on the viscous friction. Pushed away from the equilibrium the pendulum will oscillate and decrease in amplitude by a factor 0.5 during a period of 60 seconds. This results in

$$\begin{aligned} 0.5a &= ae^{60b} \\ &\Leftrightarrow \\ b &= \frac{\ln(0.5)}{60} = -0.01 \end{aligned}$$

Thus the viscous friction for the pendulum is 0.01 Ns/rad. The static friction is assumed to be small enough to be neglected.

6.6.4 Cart Friction

When trying to move the cart on the track it is obvious that there is a level of force that must be reached to get the cart to move at all. This force is due to static friction, f_{sc} which is often referred to as *stiction*. If an increasing force is applied to the non-moving cart, the stiction is equal to the force applied in the moment the cart starts to move. After calibration of the output signal so that the unit is in Newton the stiction was measured at several points of the track. Unfortunately the stiction varied quite significantly depending on the position on the track, so instead of a static friction being a function of material factors only, it is also a function of the position. The stiction is not included in the model of the pendulum, so this has to be taken in account when controlling it. Typically this is done using some sort of *friction compensation* (see section 7.9).

[1] Valid in the area where $sin\theta \approx \theta$.

Besides stiction there is a friction force which depends on the speed of the cart. A heuristic proof for this is the observation that with a constant signal to the DC-motor, which in turn affects the cart with a constant force, there is not a constant acceleration of the cart, but instead a more or less constant speed that depends on the input signal. Without a speed depending friction there would be a constant acceleration instead. A common model (at low speeds) for this friction force is that it is directly proportional to the speed of the cart. This implies that the friction on the cart may be expressed as

$$f_c(x, \dot{x}) = f_{sc}(x) + f_{kc}\dot{x}$$

For simplicity, the static friction f_{sc} is approximated with an average of the measured values along the track. This have to be compensated for to get the resulting force on the cart applied by the motor. When this assumption is made, an approximation of the kinetic friction can be made by measuring the cart speed at different input signals. At constant speed of the cart, the resulting force is zero and thus the kinetic friction is equal to the input force. So with a known input force, F, and a measured speed, $\dot{x}$ the expression for the kinetic friction coefficient is

$$f_{kc} = \frac{(F - F_{sc})}{\dot{x}}$$

This value was measured at different speeds and the final approximation of the kinetic friction coefficient is an average of these values.

6.6.5 Numerical Matrixes

After having found all parameters, the following table may be presented:

Parameter	Value	Unit
l	0.260	m
m	0.120	kg
M	0.572	kg
I_p	0.0116	kg/rad
f_p	0.01	Ns/rad
f_c	1.1	Ns/m

This yields the following numerical system description:

$$A_{up} = \begin{bmatrix} 0 & 1.00 & 0 & 0 \\ 0 & -1.71 & -0.75 & 0.02 \\ 0 & 0 & 0 & 1.00 \\ 0 & 2.71 & 16.70 & -0.55 \end{bmatrix} \qquad B_{up} = \begin{bmatrix} 0 \\ 1.56 \\ 0 \\ -2.46 \end{bmatrix}$$

$$A_{down} = \begin{bmatrix} 0 & 1.00 & 0 & 0 \\ 0 & -1.71 & -0.75 & -0.02 \\ 0 & 0 & 0 & 1.00 \\ 0 & -2.71 & -16.70 & -0.55 \end{bmatrix} \qquad B_{down} = \begin{bmatrix} 0 \\ 1.56 \\ 0 \\ 2.46 \end{bmatrix}$$

Chapter 7

Controlling the Pendulum

7.1 Control Objectives

The control objectives were the following:

- To get the pendulum from down-position to up-position using a *Swing-Up*.
- To stabilize the pendulum in the up-position. It must be able to deal with load disturbances and impulses without falling down. The desired closed loop natural frequency was set to approximately 40 rad/s.
- To stabilize the pendulum in down-position, i.e. *Crane Control*.
- To get the pendulum from the up-position to the down-position in a fast and smoth way using a *Swing-Down*.
- In all situations and modes also control the position of the cart. The track is limited, and it is desired to keep the cart from "hitting" the endpoints.

7.2 Discretization

The continuous model of the plant is described by the matrixes

$$A_{up} = \begin{bmatrix} 0 & 1.00 & 0 & 0 \\ 0 & -1.71 & -0.75 & 0.02 \\ 0 & 0 & 0 & 1.00 \\ 0 & 2.71 & 16.70 & -0.55 \end{bmatrix} \qquad B_{up} = \begin{bmatrix} 0 \\ 1.56 \\ 0 \\ -2.46 \end{bmatrix}$$

Since computer control is used, the model of the plant has to be discretized. To do so, the sampling time and the principle method for the D/A has to be known. The D/A for the pendulum (FPGA and PWM) uses *zero order hold* for setting signals to the DC motor, i.e. the signal is constant between the sampling times.

The sampling time was chosen to be 10 ms (100 Hz). This is often sufficient for a system with a natural frequency of 1 Hz, and with the desired closed loop natural frequency of 40 rad/s this is in line with the rule-of-thumb saying that $0.1 \leq \omega_c h \leq 0.6$, h being the sampling time.

Based on this information the discret time state space description in the up-position, calculated in MATLAB using the command `c2d`, is

$$\Theta_{up} = \begin{bmatrix} 1.00 & 0.01 & 0 & 0 \\ 0 & 0.98 & -0.01 & 0 \\ 0 & 0 & 1.00 & 0.01 \\ 0 & 0.03 & 0.17 & 1.00 \end{bmatrix} \qquad \Gamma_{up} = \begin{bmatrix} 0 \\ 0.02 \\ 0 \\ -0.02 \end{bmatrix}$$

where, with h being the sampling time,

$$\Theta_{up} = e^{A_{up}h} \qquad \Gamma_{up} = \int_0^h e^{A_{up}h}\, dsB$$

In the same manner the discrete system in down position is described by

$$\Theta_{down} = \begin{bmatrix} 1.00 & 0.01 & 0 & 0 \\ 0 & 0.98 & -0.01 & 0 \\ 0 & 0 & 1.00 & 0.01 \\ 0 & -0.03 & -0.17 & 1.00 \end{bmatrix} \qquad \Gamma_{down} = \begin{bmatrix} 0 \\ 0.02 \\ 0 \\ 0.02 \end{bmatrix}$$

The eigenvalues of Θ_{up} (1.00, 0.98, 0.96, 1.03) indicate an unstable system, just as expected, and the eigenvalues of Θ_{down} (1.00, 0.98, 0.99 + 0.04i, 0.99 - 0.04i) indicate a stable but higly oscillating system without asymptotic stability on the cart position. These values seem resonable.

7.3 State Machine Description

The fact that an intermediate mode is needed to get the pendulum from down-position to up-position, and that it probably is not a good idea to get it from up-position to down-position without using some kind of "slow-it-down" procedure, i.e. Swing-Down, justify that the pendulum is treated as a *state machine* according to figure 7.1. A further motivation for this is that the controllers, which are developed in belief that the system can be considered linear in a small area of the linearization points, will have uncertain behaviour if used in areas that clearly are far away from these points. Therefore it would be unsuitable (and perhaps dangerous) to use them here. Figure 7.2 show $e(\theta) = \theta - \sin\theta$ as a function of the angular deviation from the point of linearization. The error is rapidly increasing for any angular deviation larger than 1 radian.

A state machine description also facilitates the computer implementation of the controller.

7.4 Calibration and Assumptions

Before control can begin, the signals have to be calibrated so that they are given and set according to SI[1]-units, since the model is described in these. The scale for the angle and the cart position is given by the technical specification, and this is beeing corrected for in the module when reading the inputs. What remains to be calibrated is the output signal to the DC motor, which has to result in a force on the cart in Newton. Before this is done an important decition has to be taken.

If the sub-system consisting of the DC motor, the rotational axis connected to the gear wheel, and the the rubber belt is treated as a system of its own (S1), the process description in which C is the controller and P the process in figure 7.3 is given.

[1] Le Systeme international d'Unites

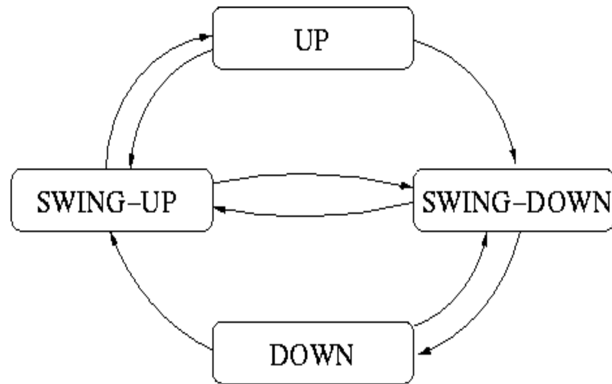


Figure 7.1: *Pendulum system regarded as a state machine.*

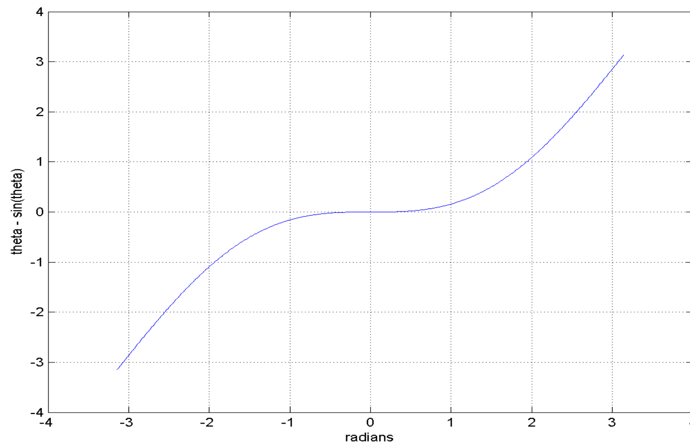


Figure 7.2: *Linearization error for* $\theta = \sin\theta$ *as a function of* θ*.*

It is clear that S1 contains much dynamics — there is a transfer function from voltages to torque due to the characteristics of the DC motor, and there is a transfer function from torque to output force on the cart due to ”spring” behaviour of the rubber belt. The decision of whether to include these dynamics in the model or not, has to be taken. Based on the following, I chose not to include them:

- The states for S1 (current, speed of gear wheel, and change in length for the rubber belt) are not given directly, and therefore have to be estimated using e.g. an observer. I find it unlikely[2] that such an observer would give sufficient estimations of the states fast enough for them to do more good than bad. Besides, measuring the parameters of the DC motor (resistance and inductance) would require surgical operation on the process which, given the price of the equipment, would have been more fun for me than for UPC.
- The speed of the gear wheel will at stabilizing control be very low. Its effect on

[2]Not an engineerish way to do it. Unfortunately I did not have time to test it.

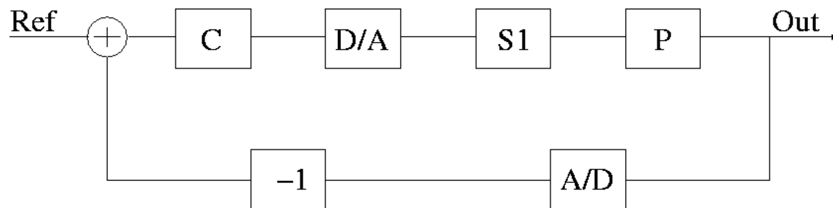


Figure 7.3: *Block diagram of the computer controlled system.*

the dynamics in terms of *EMF*[3] hence is minimal, and also the effect in terms of friction torque is minimal.

- The effect from the inductance is very fast, and since the EMF is assumed to be neglectable, a steady-state situation ($U = RI$), is rapidly reached. In this situation the output current (which in turn is directly proportional to the torque) is just a scaled version of the input voltage.
- The rubber belt seemed less elastic than to be able to disturb the control (this was however later on proven to be wrong).

Using these paragraphs the assumption was made that $S1 \approx k$ during the circumstances for control. The parameter k was given by measuring the output force (taking into account the stiction) on the cart given a certain signal to the DC motor. This was measured for several different input signals and the final approximation of k was an average of the different values, resulting in $k = 0.0145$ N per signal unit.

7.5 State Feedback

For the stabilizing control in up-position and down-position *state feedback* was used. State feedback is an intuitive and powerfull control strategy which is well astablished by now. It is also quite easy to implement in a computer. The control law is

$$u = -LX = -[k_1 \quad k_2 \quad k_3 \quad k_4] \begin{bmatrix} x_{ref} - x \\ \dot{x}_{ref} - \dot{x} \\ \theta_{ref} - \theta \\ \dot{\theta}_{ref} - \dot{\theta} \end{bmatrix} \tag{7.1}$$

The angle and the cart position are given directly from the pendulum, and the angular velocity and the cart velocity are approximated by Backward Euler in the kernel module. In the first attempt no further observation of the derivatives are done, and it is assumed that all states are given.

Due to limitations in the DC motor the output signal will saturate at a given value. The maximum force on the cart is limited to 12.86 N according to specifications.

To be able to use state feedback without limitation in possible states to reach, the system has to be *reachable*. Reachability is given by the rank of the reachability matrix

$$W_c = \begin{bmatrix} \Gamma_{up} & \Theta_{up}\Gamma_{up} & \dots & \Theta_{up}^3\Gamma_{up} \end{bmatrix} \tag{7.2}$$

[3]Electro Magnetic Force

It is easily verified that the rank of $W_c = 4$ which is the order of the system. Hence the system is reachable (which implies controllability) and, in theory, any desired closed loop behaviour is possible to acheive by applying the corresponding control vector L.

Given the desired positions of the four poles, it is possible to solve the linear system of equations that may be set up and get the corresponding control vector L. It is however convenient to use the command `place`[4] in MATLAB for this.

7.5.1 Stabilizing Control Up

Aware of the fact that the model has its limitations, the goal of the pole placement based on the model was to get reasonable behaviour, and then later fine tune on the real process. Since the swing-up controller at this point was not implemented, the control was tested by keeping the pendulum at up-right position and starting the controller. Poles chosen as $|z| < 0.9$ gave very aggressive behaviour with saturated control signals and, although stable, not a pleasant show. When the theoretical poles were moved closer to the unit circle the pendulum behaved much better.

In the testing phase some strange result came up. Allthough the pendulum appeared to be stabilized correctly, a high frequency component seemed to be picked up during the control. This frequency increased in amplitude until the pendulum eventually fell down. The strange thing about this was that the frequency was significantly higher than the natural frequency of the pendulum. Figure 7.4 show the behavior of the pendulum. The strange look of the plots at t $\approx$ 3197 is due to a the programming "error" which has been corrected later on. The control was stopped at t = 3204.

This behaviour was very confusing. When the feedback gain on the angular speed was reduced significantly, the behaviour was better, but only in the sense that the high frequency increased slower — after a while it still made the pendulum fall down.

Soon it became clear that the oscillation must have its origin in some of the unmodelled dynamics. From analyzing plots of the behaviour (figure 7.4) the disturbing oscillations seemed to have a frequency of around 16 Hz. It seemed more likely that the transmission, being a rubber belt, had something to do with this than that the DC motor was involved. The rubber belt can be treated as a spring, and therefore its behaviour can be described by its spring constant and the mass of the load it is moving. By applying a force of 30 Newton the belt expanded approximately 6 mm, hence the spring constant is $k_t = 30/0.006 = 5500$ N/m. The cart has a weight of $M = 0.572$ kg, and this yields the differential equation

$$\ddot{x}M = xk_t \tag{7.3}$$

The solution to this equation has a resonance peak at 100 radians/second $\approx$ 16 Hz. This explained from where the oscillation came. However one thing remained to explain — why did this frequency get amplified by the system? The pendulums natural frequency was much lower, but yet it was clearly affected by the 16 Hz oscillation. The answer was to be found by analyzing the pendulum closer. It is true that it had a natural frequency close to 1 Hz, but it also had another "natural" frequency — the frequency due to the fact that the aluminium rod is not perfectly stiff. By attaching the pendulums end to a screw vice and measuring the displacement of the end when a certain force was applied, the "spring constant" of the pendulum, assumed it can be treated undertaking harmonic motion, can be estimated. The spring constant was measured at three different lenghs of the pendulum: 0.45, 0.40, and 0.35 m. By

[4] Use "help place" to get mor information

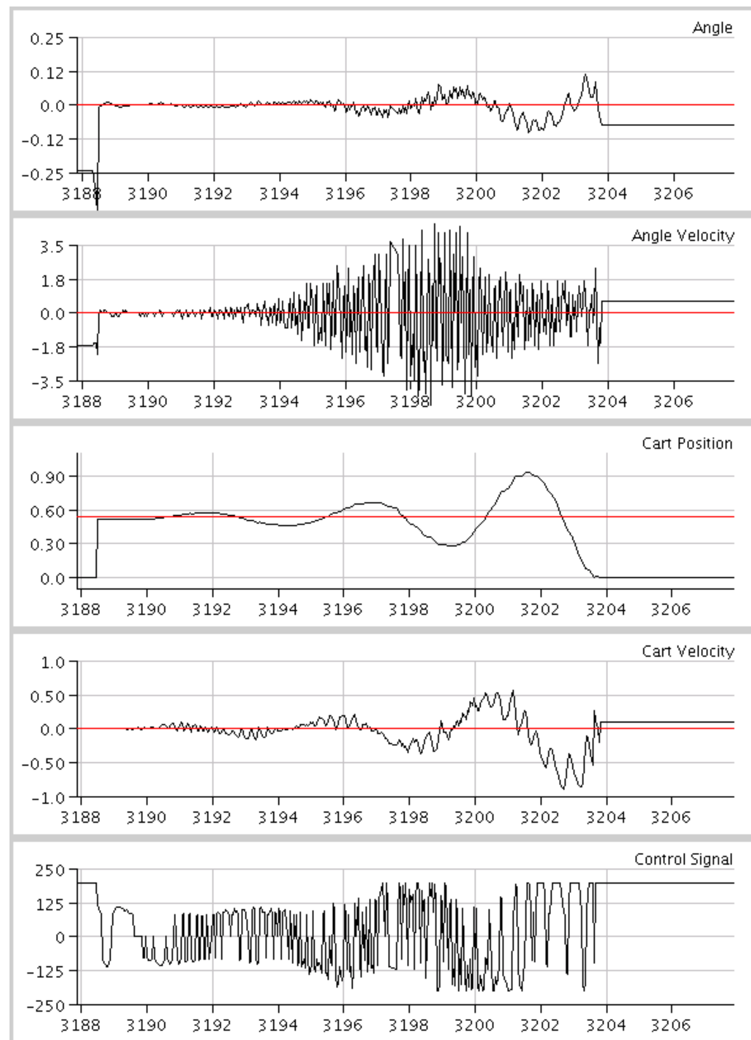


Figure 7.4: *State plots when the high frequency oscillation was present.*

finding the moment of inertia with respect to the pivot point for the different lenghts (see chapter 6 on Modelling), a differential equation similar to equation 7.3 may be set up:

$$\ddot{x}I_p = xk_p \tag{7.4}$$

with I_p being the estimated moment of inertia for the different pendulum lenghts and k_p being the different spring constants.

Plotting the bode diagrams for the different cases in the same diagram as the bode diagram for the transmission revealed what was expected. When the pendulum has a length of 0.45 m, which it initially had, it has a natural frequency due to the aluminium which is very close to 100 radians/second. This explains the scenario. In some way frequencies close to 16 Hz was generated by the control law and was picked up by the transmission. These frequencies was then picked up by the pendulum and feedbacked

(fed back) in to the control law. When the control law later on started sending signals to the DC motor with this frequency, all components that make up an unstable system were present. Figure 7.5 show the different bode plots.

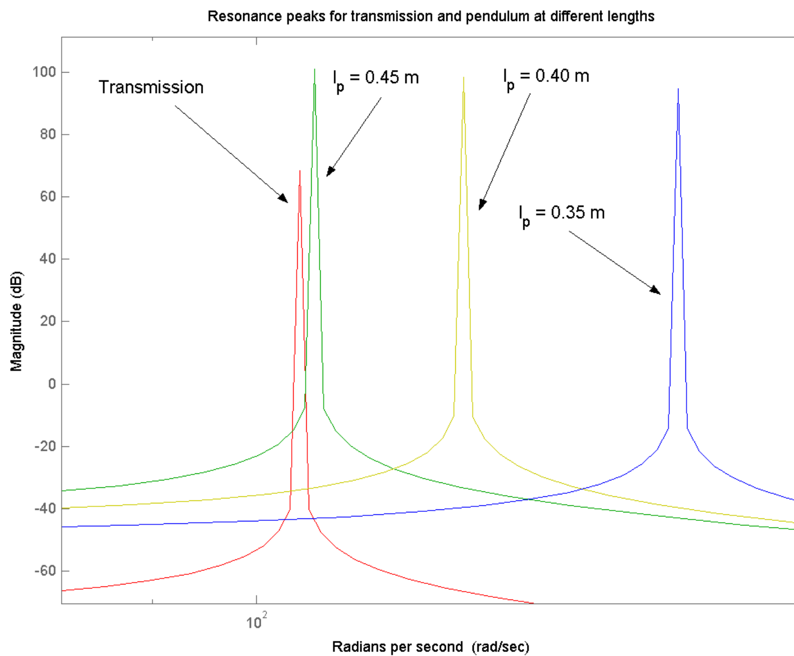


Figure 7.5: *Bode plots for transmission and pendulum at different lenghts.*

By changing the tension of the belt just a little bit, and at the same time making the pendulum shorter (by changing the point where it was attached) the natural frequency of the transmission was separated from the natural frequency of the pendulum. After recomputing the value of the moment of intertia and finding a new feedback vector in MATLAB the controller worked much better. The estimated values in chapter 6 are the one from using the shorter pendulum. After tuning the parameters manually, the best feedback vector was found to be $L = (-9, -8, -48, -4)$, but ironicly, using this feedback vector the controlled system's theoretical poles are $(1.0003 + 0.09i, 1.0003 - 0.09i, 0.99+0.01i, 0.99-0.01i)$, and thus indicate an unstable system. Figure 7.6 shows the states and control signal with two impulses in either direction on the pendulum angle.

7.5.2 Stabilizing Control Down

When hanging straight down, the pendulum system has an stable equlibrium. With the viscous friction of the pendulum present, the equilibrium is asymptotically stable, which would not be the case if the viscous friction of the pendulum would have been zero. When perturbed, the pendulum will thus return to quiet after a certain amount of time. Due to poles at $-0.3 \pm 4.1i$ the system is poorly damped, and faster damping is desired. Using the feedback vector $L = (25, 15, 35, 0.1)$ the poles of the theoretical

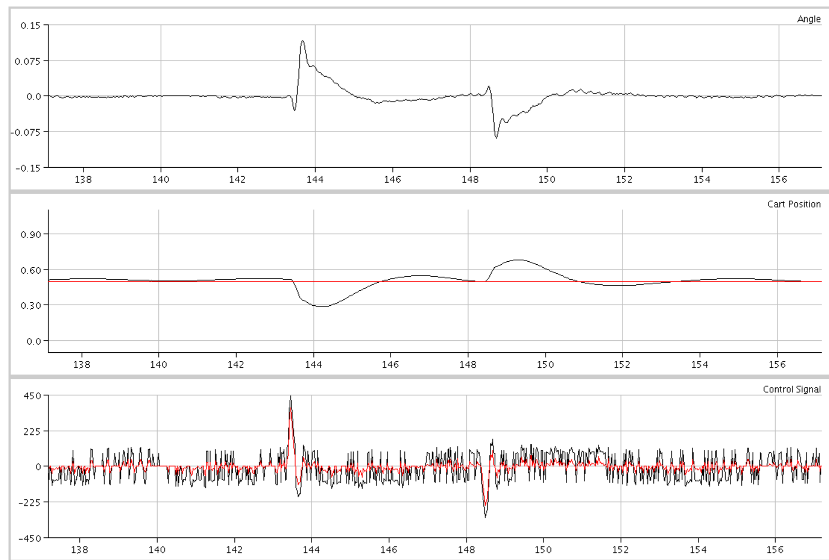


Figure 7.6: *State plots with two impulses on angle when stabilizing in up position.*

closed loop system are $(0.81, 0.96, 0.99 + 0.02i, 0.99 - 0.02i)$, and this resulted in much better performance. Figure 7.7 shows the states and control signal with two impulses in either direction on the pendulum angle at times t = 62 and t = 69. The closed loop natural frequency is however probably lower than 40 rad/s, which was the approximate objective.

7.6 Model vs Real Pendulum

To test whether the model behaved similar to the real pendulum a simple test was set up. The model of the controlled pendulum was implemented in Matlab Simulink according to figure 7.8.

The simulation time was 10 seconds with an impulse in the control signal after 2 seconds. The model starts in stabilizing mode using the same feedback vector, and to make the simulation more realistic, white noise was added to the control signal. This was compared with the real process and a quite large impulse on the cart. An impulse direct on the cart is more or less the same as an input on the control signal. The simulated states are shown in figure 7.9 and 7.10, and plots from the real process are shown in figure 7.11. Except for the small oscillation for the cart in the real process, the behaviour of the states are very similar.

7.7 Swing-Up and Swing-Down

A common way to get the pendulum from down-position to up-position is to use a controller based on the energy for the system. The basic idea is that the pendulum must ”slow down” and eventually hang quiet if the system's energy constantly is decreasing, and show the opposit behaviour if the energy is increasing. Often a simplified model of the pendulum is adequate for using this approach.

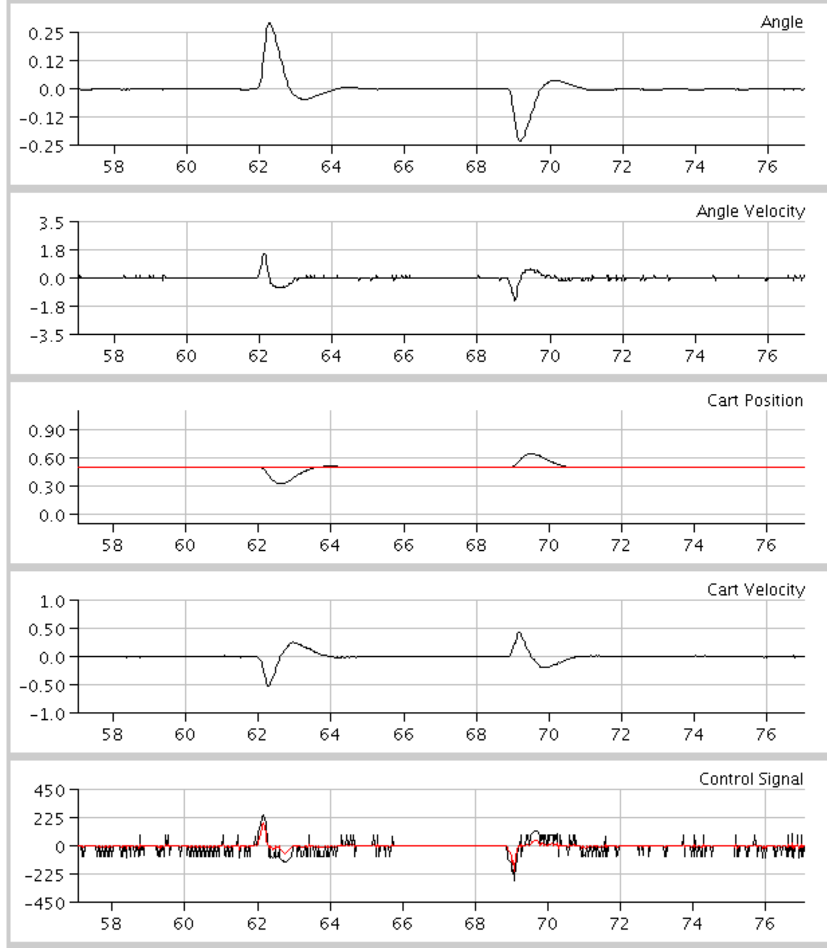


Figure 7.7: *State plots with two impulses on angle when stabilizing in down position.*

If the dynamics of the cart and the friction of the pendulum is neglected, the equation for the pendulum motion is described by

$$\ddot{\theta} = \frac{mgl}{I_p}\sin(\pi - \theta) - \frac{ml}{I_p}u\cos(\pi - \theta) \tag{7.5}$$

with u beeing the force on the pivot point (moving the cart).

The kinetic energy (E_k) and the potential energy (E_p) of the pendulum are

$$\begin{aligned} E_k &= mgl(1 - cos(\pi - \theta)) \\ E_p &= \frac{I_p\dot{\theta}^2}{2} \end{aligned}$$

and $E_t = E_k + E_p$ is thus the total energy of the pendulum, defining E_k as zero when the pendulum hangs straight down. The time derivative of E_t is

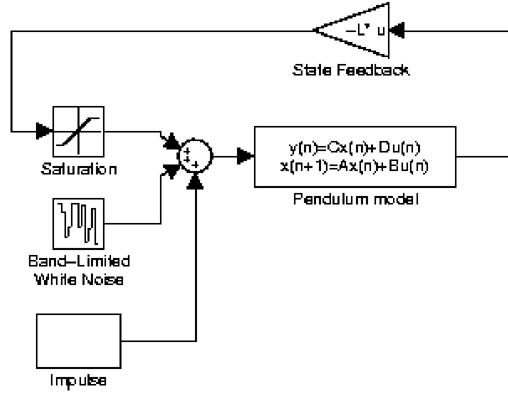


Figure 7.8: *Matlab model of the controlled pendulum*

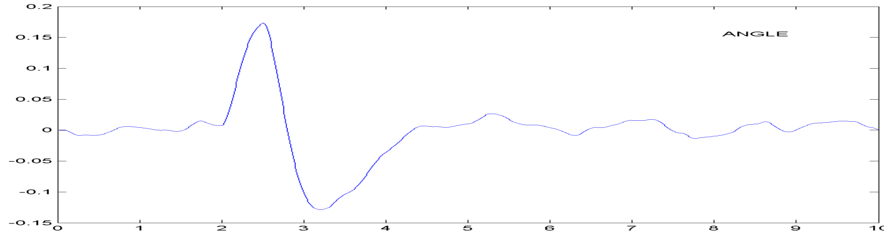


Figure 7.9: *Simulated angle with an impulse on the control signal.*

$$\begin{aligned}
\dot{E}_t &= I_p\dot{\theta}\ddot{\theta} + mgl(-(-\sin(\pi-\theta)(-\dot{\theta}))) \\
&= I_p\dot{\theta}\ddot{\theta} - mgl\sin(\pi-\theta) \\
&= I_p\dot{\theta}(\frac{mgl}{I_p}\sin(\pi-\theta) - \frac{ml}{I_p}u\cos(\pi-\theta)) - mgl\sin(\pi+\theta) \\
&= -\frac{\dot{\theta}ml}{I_p}u\cos(\pi-\theta)
\end{aligned}$$

Thus, by choosing

$$u = k_{su}\mathrm{sign}(\dot{\theta}\cos(\pi-\theta)) \Rightarrow \dot{E}_t \geq 0 \quad (7.6)$$

$$u = -k_{sd}\mathrm{sign}(\dot{\theta}\cos(\pi-\theta)) \Rightarrow \dot{E}_t \leq 0 \quad (7.7)$$

with $k_{su} \geq 0$ and $k_{sd} \geq 0$, it is possible to force the total energy to either increase or decrease. For Swing-Up input according to equation 7.6 is used, and for Swing-Down input according to equation 7.7 is used.

To avoid getting a system energy which results in a angular velocity being too high to later on catch the pendulum, a control signal in Swing-Up is only set if the total energy $E_t = E_k + E_p \leq 1.2E_{up}$, with E_{up} being the energy of the pendulum standing

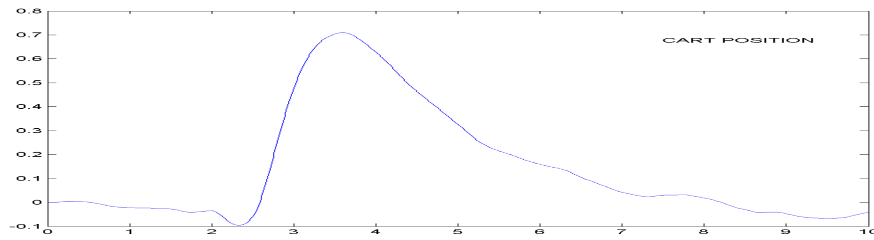


Figure 7.10: *Simulated cart position with an impulse on the control signal.*

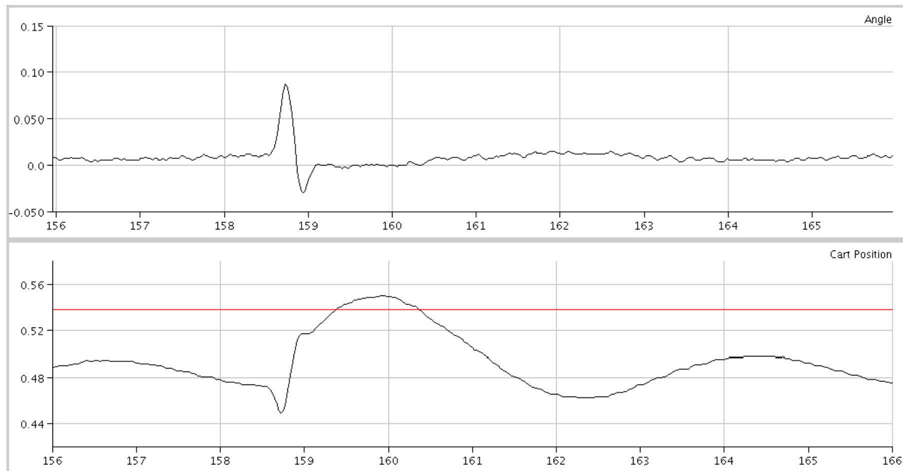


Figure 7.11: *Real process with an impulse on the cart.*

up having zero angular velocity. If the energy is higher than the threshold a zero-signal is set to the DC motor.

In the points where the stabilizing controllers take over, there are certain conditions based on *fuzzy control* that must be fulfilled. In general, the switching to a stabilizing controller is allowed if the pendulum angle is within a predefined area at the same time as the angular velocity is within a predefined range. Close to the endpoints, there are additional conditions to avoid "hitting" the endpoints. The rules are roughly that a swith is only allowed to take place if the stabilizing controller initially will stabilize by moving to the center of the track. 0.4 radians as maximum allowed angle together with 4.0 radians/second as the maximum allowed angular velocity resulted in reliable switchings.

An efficient Swing-Up is a trade off between getting the pendulum up fast and to keep the cart away from the endpoints. From the pendulum being quiet, the normal time for the Swing-Up to swing it up and catch it is less than four seconds. Figure 7.12 show a typical scenario. The Swing-Up is initialized at t = 0, and at the same time the angle zero is changed from initialy being straight down to instead straight up. For every turn of the cart, the pendulum angle increases (closer to 0) until it, at t = 3.4, switches to stabilizing control.

To get the pendulum from up position to stabilizing down position takes about the same amount of time, which is shown in figure 7.13. At t = 42.8 the Swing-Down is

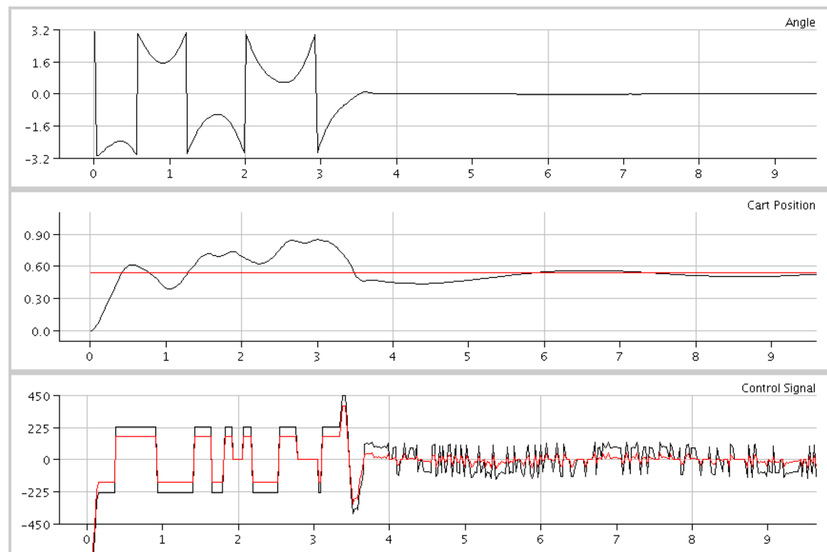


Figure 7.12: *State plots of a switch from Swing-Up mode to stabilizing control in Up mode.*

initialized and the pendulum falls freely until the angle is larger than $\pi/2$. Then the energy absorbing control is activated until the switching to stabilizing control down is taken place at t = 45.3. When the user orders the controller to get the pendulum down, the reference angle is changed from being zero straight up to being zero straight down to avoid not seeing the new reference angle in the plotter window. Without this information, the plotts may seem confusing. Compared to the inverted pendulums in litterature, these times seem to be quite good. Notice that it will take a few extra seconds to get the cart to the cart reference.

7.8 Cart Control

The objective of keeping the cart away from the ends is rather easily obtained when the pendulum is in stabilizing mode either up or down, due to the feedback of cart position error and cart velocity error. An example of a reference step in the cart position, and thus an example of the performance of the cart control, is shown in figure 7.14. Notice that the cart initally moves in the wrong direction, which implies a non-minimum phase transfer function from input to cart position.

The difficult part is to avoid hitting the ends in Swing-Up mode and in Swing-Down mode. In these modes the control signal is only based on the angle and the angular velocity, and without further cart control the cart will hit the endpoints sooner or later. One solution to the problem is to send any output signal through a "filter" of rules. Most of the rules are based on simple logic — find the critical areas and find the rules to either avoid these or, in worst case, get the cart out of these. Next section explains one way of implementing such rules.

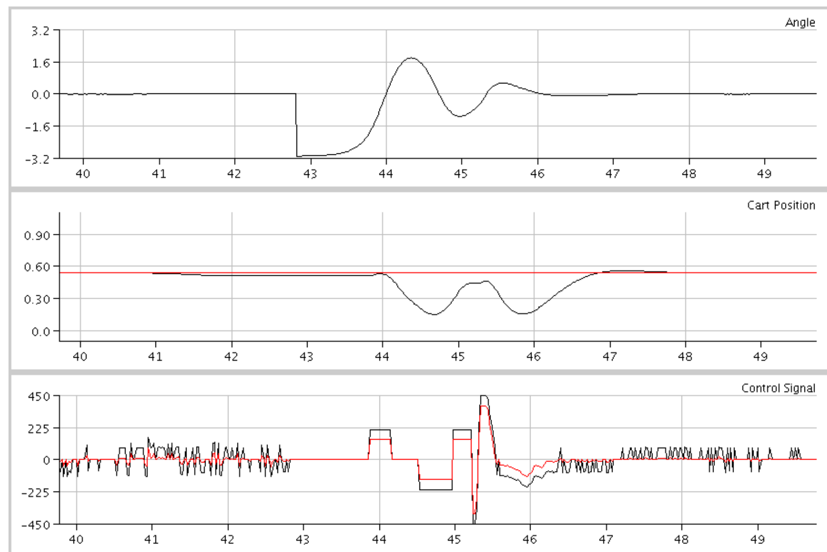


Figure 7.13: *State plots of a switch from stabilizing control up via Swing-Down to stabilizing control down.*

7.8.1 Software Spring

At the endpoints small shock absorbing pads are mounted to stop any wild cart. However, these pads are quite small and it is prudent to add some extra safety. Instead of adding more shock absorbing material the system will at all control modes be protected by a *software spring*. Any time the cart enters an area protected by the spring, i.e. close to the endpoints, a force, u_{spring}, proportional to the distance of entrance is applied in the opposit direction. The total signal to the DC motor is thus

$$u = u_c + u_{spring} \tag{7.8}$$

As an extra security, the last 10 cm on each side of the track are protected even more. If, in control mode, the cart enters these areas, maximal signal is set to get the cart out of there. This length is not possible to change from the GUI — only by changing it in the code. Figure 7.15 shows the function for u_{spring}.

The software spring works satisfying, but in systems requiring extra protection it is of course inadequat to rely on protection like this. If the program would hang with the latest signal to the DC motor set as a high value in either direction, all that is left for protection are the two shock absorbing pads.

7.9 Friction Compensation

The model for the friction on the cart is

$$f_c(x, \dot{x}) = f_{sc}(x) + f_{kc}\dot{x}$$

with f_{sc} being the stiction and f_{ks} being the friction depending on the cart velocity.

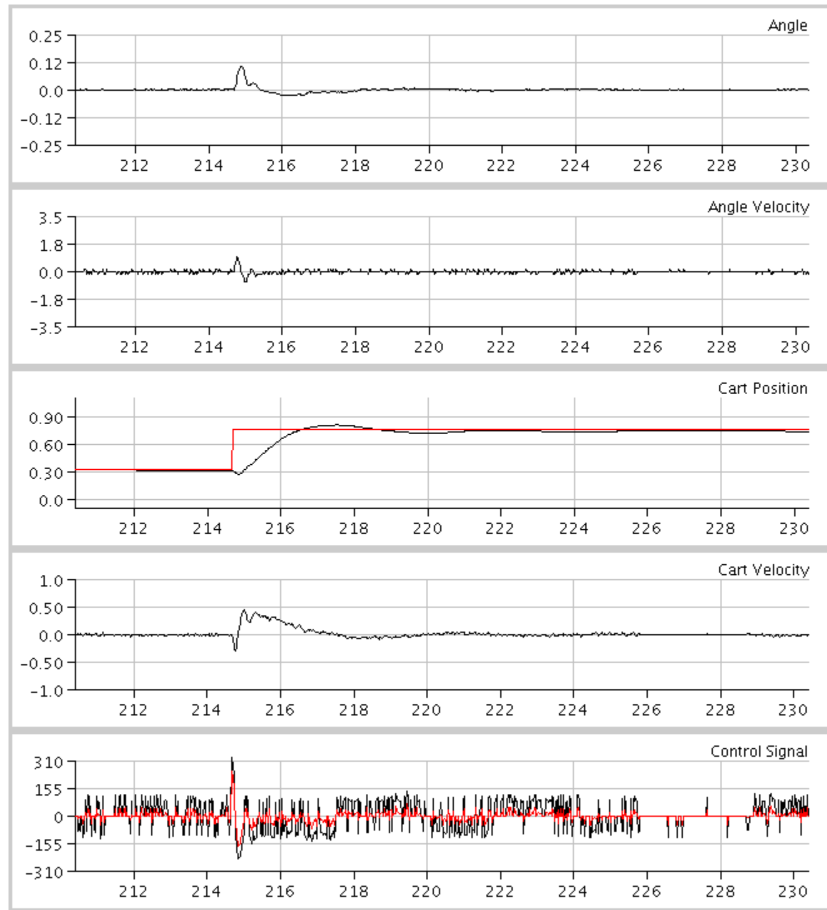


Figure 7.14: *Change of Reference for the Cart Position.*

Depending on the cart's position on the track, the stiction varies between 0.75 N and 1.2 N. A common consequence of friction is the *limit circle* phenomenon. This is a non-linear phenomena which is explained using describing function analysis.

The velocity depending friction is included in the model, so this does not require any friction compensation. An easy way to reduce the effects of the stiction is to use a friction compensation according to the following:

$$\text{output} = \begin{cases} s_c + f_{sc} & \text{if} \quad s_c > 0 \\ s_c - f_{sc} & \text{if} \quad s_c < 0 \end{cases}$$

where s_c is the calculated control signal and f_{sc} is the average stiction force on the track. Using this compensation the limit circle, which else is very obvious, is significantly reduced. Figure 7.16 show plots from the stabilizing control. At time t = 981 the friction compensation is activated.

Despite tuning the parameters and the friction compensation, I could not remove the small oscillation which remained and at the same time retain the same control performance. The oscillation decreased if the feedback on the cart velocity was increased, but in my oppinion this deteriorated the control of impulses. I also experienced that

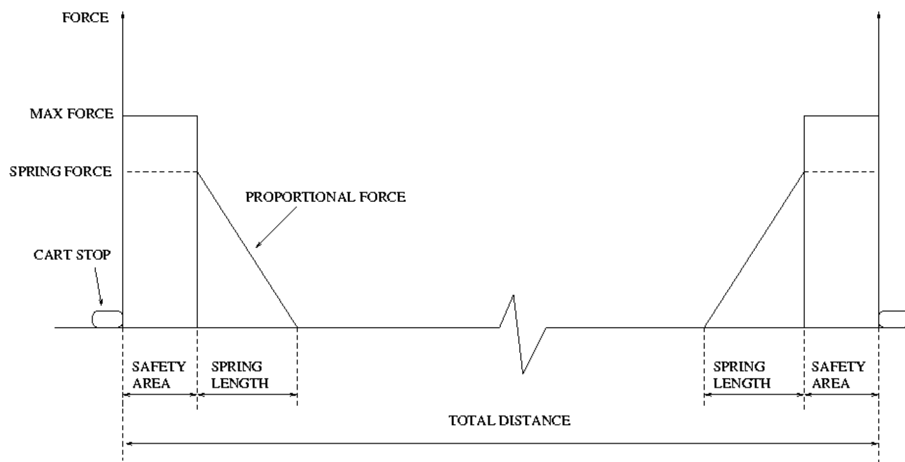


Figure 7.15: *Different areas for the software spring.*

the oscillation was marginally reduced if the zero limit in the implemented filter was set to a lower value, but then the pendulum was more ”fiery” than before. A good compromize was therefore to set the zero limit to 3 signal units and the static friction to 73 signal units (1.06 N).

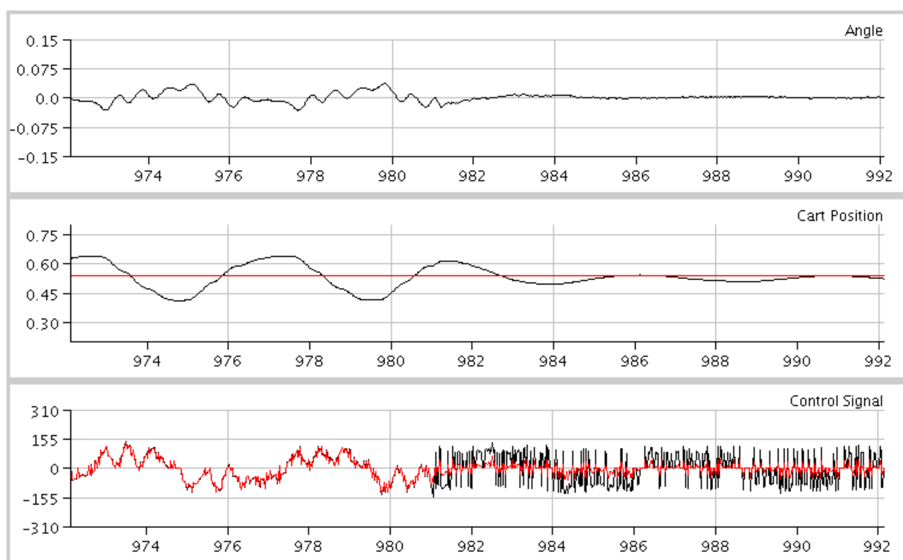


Figure 7.16: *Friction compensation activated at* $t = 981$.

Chapter 8

Deterministic Disturbance

As an extra feature, besides writing a program for hard real-time pendulum control, another kernel module was implemented. This module gives the possibility to start other hard real-time threads, i.e. *dummy threads*, that run in parallel with the control thread. The dummy threads each have the following parameters:

Period Time set in micro seconds.

Computation Time set in micro seconds. The dummy thread's ”computation” is a simple time conditioned while loop.

Priority set according to the RTLinux priorities, i.e. from 0 (minimum) to 100000 (maximum).

By setting the priority of a dummy thread higher than the priority of the control thread, the control can be disturbed in a deterministic way. This feature was not used nearly as much as it could have been if extra time would have been availabe, so this chapter will only work as an introduction to the many tests that can be done.

When hard real-time is not available, it is left to computer simulations to prove any theoretically derived result. I expect it to be useful to perform practical tests as well.

This chapter gives example of two different test:

- Evaluate the control when the sampling time is different than the sampling time the controller was developed for.
- Measure the control error when the controller is being disturbed by a RT thread with different parameters.

8.1 Variable Sampling Time

The controller developed in chapter 7 is intended to be used at the sampling rate 100 Hz (10 ms sampling time), and it is interesting to investigate how the controlled system behaves if this controller is used at sampling times different from 10 ms. By recording data from different control session which uses the same controller but different sampling times, it is possible to compare and get a visual overview of how much the control is destroyed by wrong sampling time. If the data from the different sessions are going to

be compared, it is important to have as similar conditions as possible. To get similar conditions the data recording feature of the control program was used with the same settings through all tests. Every control session was executed for 20 seconds with a 10 cm step in the cart reference reference set after 2 seconds. All sessions were started from the center of the track with the pendulum being as stabilized as possible with the original sampling time. Figure 8.1 shows the weighted sum of the state errors as function of sampling time. This sum is the absolute values of the errors weighted in some sense after their respective importance. The weights used were $(10, 1, 20, 1)$ using the state error vector $(x_{err}, \dot{x}_{err}, \theta_{err}, \dot{\theta}_{err})$. The weighted sum of the errors when using the original sampling time is set to zero, and all the others are presented relative to this, i.e. the plotted points are $t_{si}, (\sum E_{t_{si}} - \sum E_{10\ ms})/\sum E_{10\ ms}$.

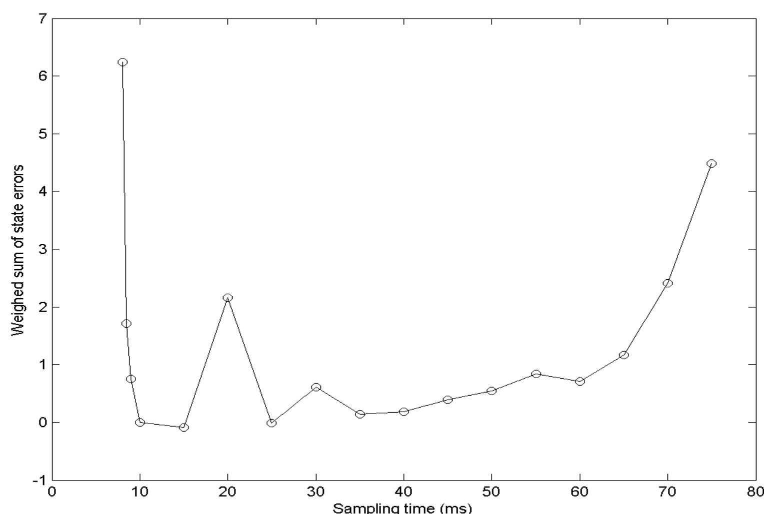


Figure 8.1: *Weighted sum of state errors as function of sampling time.*

Somewhat surprising, the results in figure 8.1 indicate that the controller would work marginally better if used at the sampling time 15 ms, and dramatically worse if the sapmling times are smaller than 10 ms. The results from the latency tests for standard Linux show that it is common with too short sampling times as well as too long. It is interesting that the same deviation from the true sampling time obviously is more harmful when it shortens the sampling time than makes it longer. It is also clear that in a non-real-time operating system, it will require burst errors of long delays of the sampling time to really jeopardize the stability of the pendulum. However, in a critical situation, e.g. if the pendulum would face a direct impulse just after the control signal had been set, and this would have been followed by long delays, then the width of the delay burst required to jeopardize the stability would of course increase.

The small peak at 20 ms is likely due to some unmodelled dynamic. At this sampling rate the entire rack is vibrating, and with more firm attachment to the wall it is possible that this peak would have disapeared. Notice that the control signal is not included in the error since the control signal used in control always is below 50 % of its maxium and thus much power is available. Also notice that these test does not measure the ability to restrain disturbances that affects the pendulum directly. Even

though the sampling times 10, 15 and 25 ms seem to be equally good, it is possible that tests that involve impulsed directly on the pendulum (similar to the argument above) would be advantageous for the faster sampling time

8.2 Variable Disturbance

The CPU utilization is defined as

$$U = \sum_i U_i = \sum_i \frac{T_{Ci}}{T_{Pi}} \tag{8.1}$$

with T_C and T_P being the computation time and period time for the different tasks. If the control module is executed at the same time as the RT thread, the utilization is

$$U = U_{control} + U_{RT} + U_{Linux} = \frac{100}{10000} + U_{RT} + U_{Linux} \tag{8.2}$$

since the worst case computation time for the control module is approximately 100 µs and the computation time is 10 ms. It is assumed that the kernel overhead, i.e. CPU required to schedule the different tasks, is zero. The CPU utilization for Linux will not disturb any tests since it runs with the lowest of RT priorities.

It is interesting to investigate whether the same CPU utilzation for a RT thread disturbs the pendulum differently depending on its composition of pairs of $(T_C, T_P)|T_C/T_P = U_{RT}$. The pairs used in the tests were $U_{RT} = T_C/T_P \mid (T_P, T_C) \in$ {(1.00, 0.20), (0.80, 0.16), (0.6, 0.12), (0.4, 0.08), (0.2, 0.04)}, i.e the CPU utilization for the RT thread was always 0.2. It is not useful to lower the computation time below 0.01 s, since the controller then (with a worse case computation time of approximately 100 µs would be able to perform control once every 10 ms anyway.

There exist at least two different hypotheses about the outcome of the tests:

- If the plant would be truly linear, one possibility would be that the different pairs would result in approximately the same error, assuming the errors was added for a sufficiently long time. The argument for this is that a large computation time for the RT thread, and hence the same long time for the controller to run open loop, would be compensated for by performing normal control for a long time in between. This would be compared to the situation when the computation time for the RT thread instead would be small, but when normal control almost never could be applied since the time between the open loop races would be small, i.e. the control would be disturbed (almost)at all time. However, the theoretical function of this type of disturbance on linear plant would probably result in a curve with an "optimum", i.e. point where the control was least "destroyed", instead of a constant error.
- With the knowledge that the plant is non-linear, and that the approximation of the pendulum being linear only is valid for small deviations from the angle of linearization, the probable outcome is instead that the control will perform worse the longer it will have to run in open loop. The longer time the control runs in open loop, i.e. the feedback is broken, the larger is the probabilty that the pendulum will fall outside the "linear" area and to correct this would be more and more difficult.

The practical result are most likely a combination/superposition of the two effects (and perhaps several more).

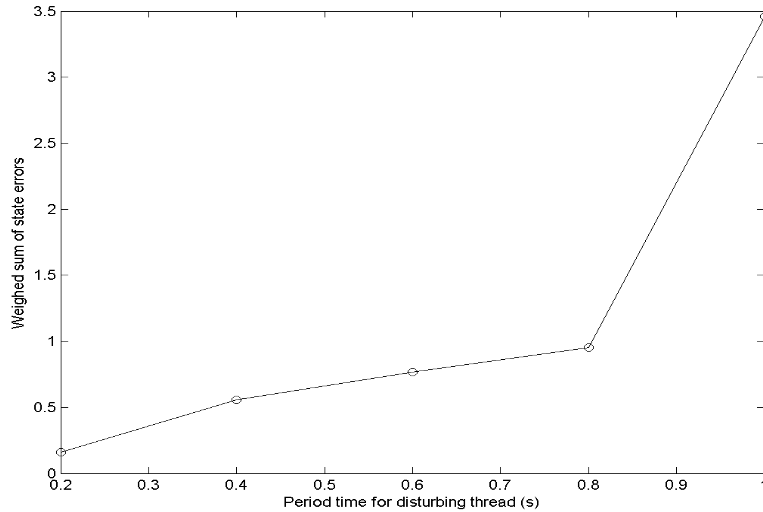


Figure 8.2: *Weighted sum of state errors as function of period time for* $U_{0.2}$ *disturbing thread*

8.3 Possible Future Tests

With very small changes in the code for the dummy-modules and for the control module it would be possible to do the following tests:

- Let the computation time for the RT thread come from a known random distribution.
- Let the sampling time for the control module be $t_s + t_r$ where t_s is the original sampling time and t_r is from a known random distribution.
- Let the sampling time for the control module depend on current CPU status and/or control error, and use a look-up table to choose control law.
- Control two pendulums (or other processes) in parallel. This was done very briefly at the end of the project with very encouraging results.
- More thoroughly investigate if desired control may be achieved using lower sampling rates. The test with variable sampling time indicate that this may be possible, and with the ongoing work of trying to control the pendulum with computer vision (video), this is highly relevant.

Chapter 9

Result and Summary

The general results were very good. The objectives were to be able to control the plant in hard real-time, and these have been accomplished. Also, the control results were good with satisfying results in all the control modes. More simulations could have been done before the actual implementation on the controller, but due to the main objectives, it would have been wrong to start with simulations before the control platform was finished. The priorities of hard real-time control were higher than on simulations. However, I stongly recommend to begin with simulations in any other case, and preferably keep running simulations in parallel with the real implementation as the work advances.

This is version 1 of the program. The first version always contains hidden errors which will be found when other people start to use it. If this is the case, please let me know. Furthermore, Java Swing is slow and there are most likely things that could have been improved here as well. I do not consider myself being a good programmer of GUIs. I think this is tricky and it is more difficult to find out how to actually get something the way you want it, than to come up with the idea. The programming I like is the absolute opposite. Despite this, the GUI works and offers good and usable features and the freedom to choose what kind of information to be displayed.

It is clear that it is possible to gain very good results with a much simpler model of the pendulum. Both the pendulum friction and the viscous friction of the cart could have been neglected. This would have resulted in easier computations of the different transfer functions.

Future work on the program could involve the use of EDF scheduling of the real-time tasks, adding code so that it would be possible to visualize the different threads executions, and using the program for controlling different plants. To visualize the execution of different RT threads would have been especially interesting to combine with real process control performance and the use of other schedulers, e.g. EDF. As mentioned in the report, it is possible to sample with rather high sampling rates with the same strict real-time contstraints on the sampling time, and therefore control of (much) faster processes is possible.

Appendix A

User Manual

The user manual is enclosed at the end.

Appendix B

Miscellanous

Information about the project, source code, Javadoc, etc. may be downloaded from www.efd.lth.se/˜d98mse/

B.1 RT-DAC4/PCI Driver API

```
/**
 * Open a connection to the specific device
 */
open(struct rtl_file *filp)

/**
 * Close the connection to the specific device
 */
close(struct rtl_file *filp)

/**
 * Reads angle value and cart position as two incremental
 * unsigned two byte values. The values are being written
 * to buffer buf.
 */
read(struct rtl_file *filp, char *buf,
     size_t count, off_t* ppos)

/**
 * Write the two byte signed value in buffer buff to the
 * DC motor.
 */
write(struct rtl_file *filp, const short *buff,
      size_t count, off_t* ppos)

/**
 * request = 1: Reset encoders
 * reguest = 2: Set frequency l as new PWM frequency
 */
```

```
ioctl(struct rtl_file *filp, unsigned int request,
      unsigned long l)
```

B.2 Latency Test Diagrams

Figure B.1 and B.2 show the distribution for the periodic latency of a normal thread executed in standard Linux.

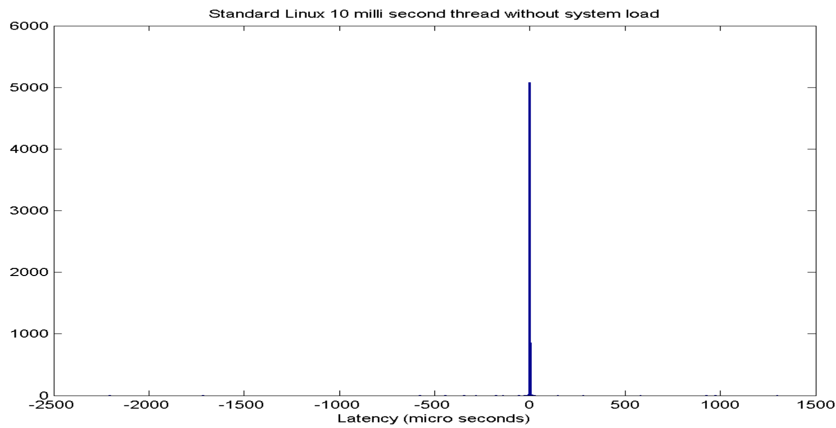


Figure B.1: *Periodic latency distribution in* µs *for a* 10 *ms standard Linux thread without system load.*

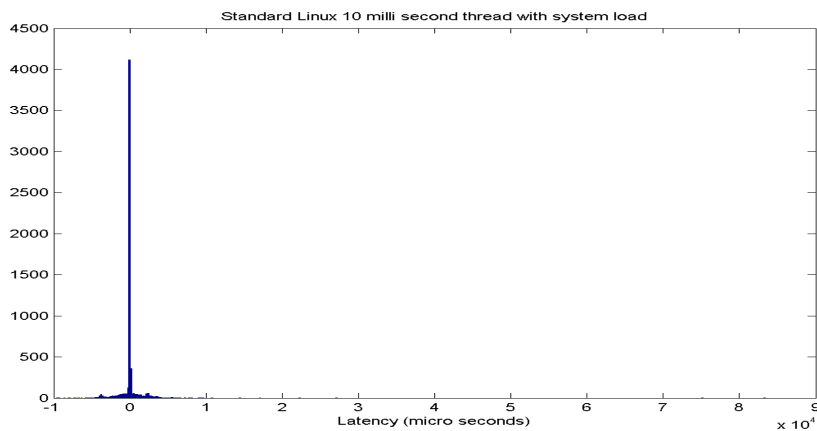


Figure B.2: *Periodic latency distribution in* µs *for a* 10 *ms standard Linux thread during system load.*

Figure B.3 and B.4 show the distribution for an improved thread when executed in a no-load situation and in a situation with load in standard Linux.

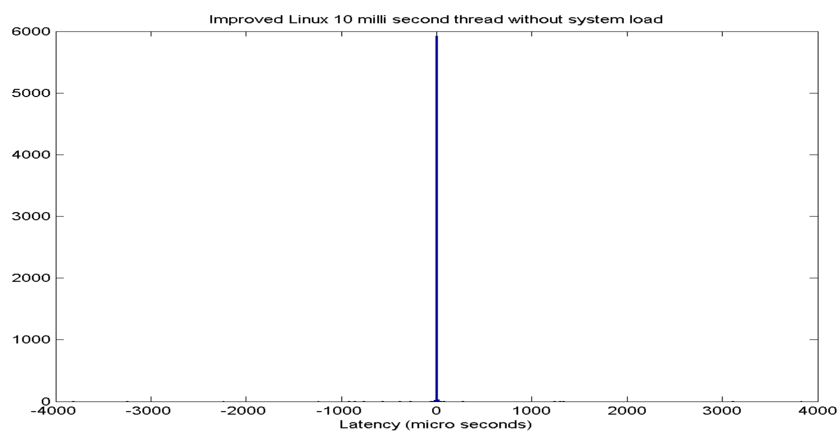


Figure B.3: *Periodic latency distribution in* µ*s for a* 10 *ms high priority Linux thread locked in RAM without system load.*

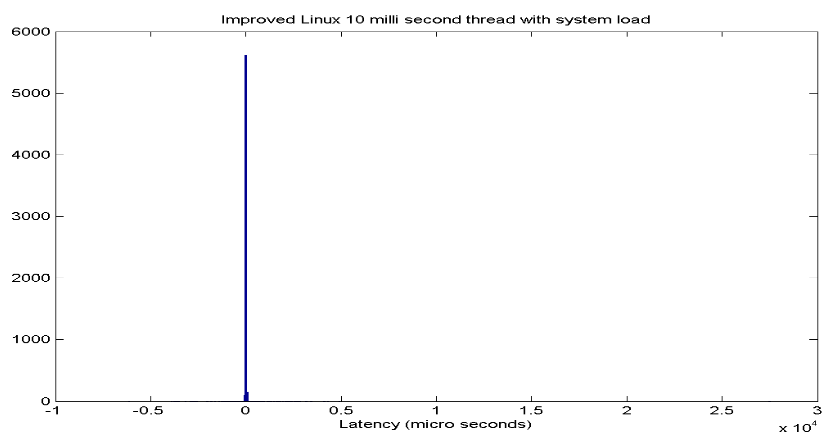


Figure B.4: *Periodic latency distribution in* µ*s for a* 10 *ms high priority Linux thread locked in RAM during system load.*

B.3 Screen Dump of the Java Program

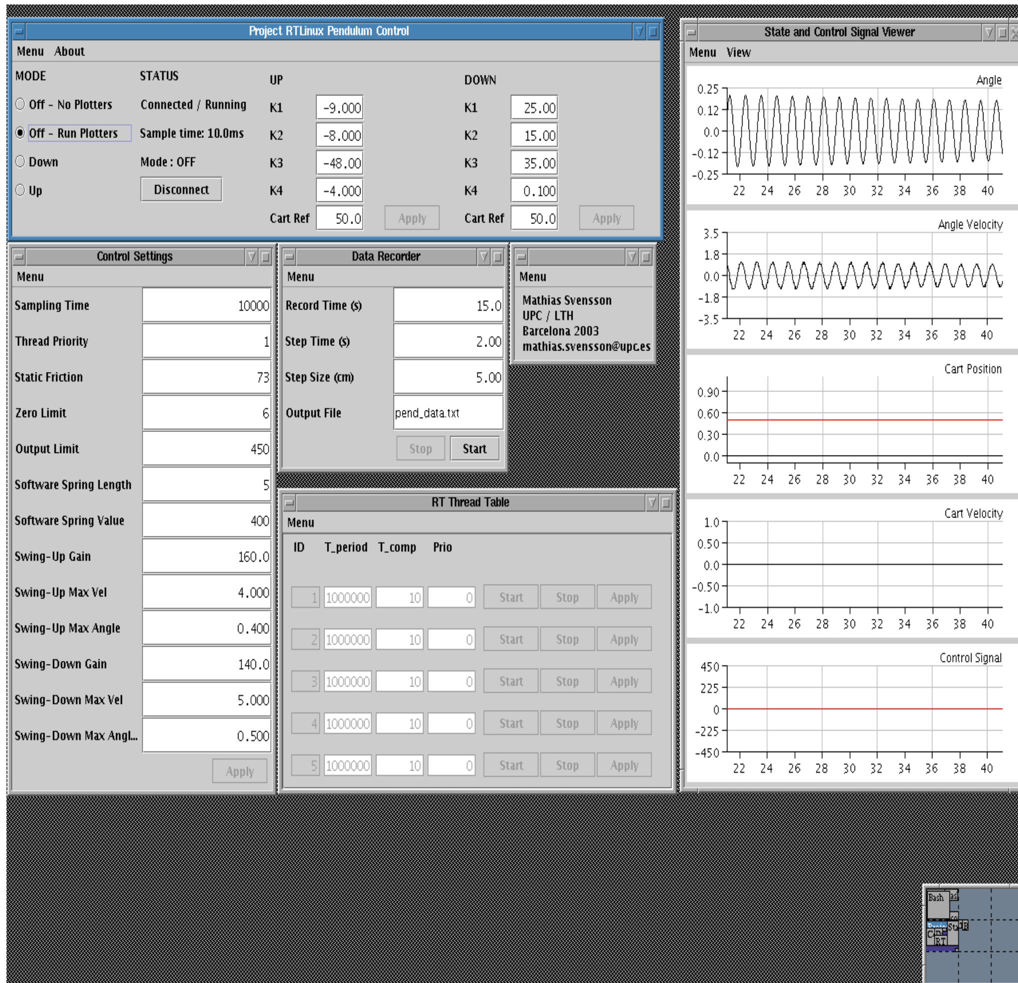


Figure B.5: *Screen dump of the Java program.*

Bibliography

[1] Michael Barabanov
Getting Started with RTLinux
2001

[2] Victor Yodaiken
The RTLinux Manifesto
Department of Computer Science
New Mexico Institute of Technology

[3] Matt Sherer
Writing Applications with RTLinux
FSM Labs, Inc.
2002

[4] *RTLinux Installation Instructions*
FSM Labs, Inc.
2001

[5] *Introduction to Linux for Real-Time Control*
National Institute of Standards and Technology
Intelligent Systems Division

[6] Doug Abbott
Linux for Embedded and Real-time Applications
Elsevier Science (USA), 2003

[7] Karl-Erik Årzén
Real-Time Control Systems
Department of Automatic Control
Lund Institute of Technology, Lund 2001

[8] Jean-Jacques E. Slotine / Weiping Li
Applied Non-Linear Control
Prentice-Hall Inc., 1991

[9] Helmut Kopka & Patric W. Daly
A guide to LaTeX
Addison Wesley, Great Britain, 1999

[10] Steven Holzner
C++ Black Book
The Coriolis Group, 2001

[11] Karl J Åström / Björn Wittenmark
Computer Controlled Systems, theory and design, third edition
Prentice Hall, 1997

[12] Torkel Glad / Lennart Ljung
Reglerteknik - Grundläggande teori
Studentlitteratur, Lund, 1989

[13] Erik De Castro Lopo
Faster Floating Point to Integer Conversions
Version 1.1, 2001/11/02
http://www.mega-nerd.com/FPcast/

Manual
V.1.2

Real-Time Control Program for Pendulum using RTLinux/Free

Mathias Svensson

UPC / LTH

Contents

1 Preface **3**

2 Installation of RTLinux **5**

2.0.1 Kernels . . . 5

2.0.2 Patching and Compilation of Kernels . . . 5

3 Settings before Starting **7**

3.1 Load the RTLinux Modules . . . 7

3.2 Set User Rights . . . 7

3.3 Installation of Java . . . 8

3.4 Compilation and Loading of RT Modules . . . 8

3.4.1 rtl_pcidriver . . . 8

3.4.2 rtl_modules . . . 9

3.4.3 Java Program . . . 9

4 To Use the Program **10**

4.1 To Control the Pendulum . . . 10

4.2 To Use the RT-threads . . . 14

Chapter 1

Preface

This manual is primarily intended to help with the installation and set up of the RTLinux/Java control program for the Inteco Pendulum at UPC/ESAII. Some of the paragraphs are specific to the computer used at UPC, whereas the other paragraphs may work as a general RTLinux installation guide. It is difficult to write a manual that will cover every possible scenario in the installation, and for that reason is assumed that the user of this manual is familiar with Linux and with kernel compilation, etc. If this is your first experience with Linux, then please ask someone for help, otherwise it might be difficult.

The cd contains the following:

- java
  - javalib
  - j2sdk-1_4_2_02-linux-i586.bin
- kernels
  - linux-2.4.20.tar.bz2
  - rtlinux-3[1].2-pre2.tar.bz2
- program
  - java
  - rtl_modules
  - rtl_pcidriver
  - set_fifo_rights
- graphics_driver
- docs

If there is an excisting installation of Linux then it is possible to use this one with the RT-kernel as base.

If instead a clean installation is going to be performed. Begin with the installation of any Linux distribution. The system is tested with Debian 3.0 r1, but should perform

equally good with any other distribution. The installation of RTLinux will however be slightly different. I recommend using a journaling file system, e.g. ext3.

From this point it is assmued that a standard Linux distribution is installed and functioning with the most common programs, e.g. make, gcc, patch, etc. Compiling the kernel and loading modules require root-rights, so make sure you either have this or the possibility to use ”su”.

Chapter 2

Installation of RTLinux

2.0.1 Kernels

Download a fresh copy of a Linux kernel and a fresh copy of RTLinux to /usr/src. Unpack and unzip them. Make sure that the version of RTLinux contains patches for the chosen kernel.

The cd contains Linux kernel 2.4.20 and RTLinux-3.2_pre2 which has been tested together. Copy these from the folder /cdrom/kernels to /usr/src. Unpack and unzip them. The program bunzip2 is needed for unzipping files in .bz2 format.

Put the cd in the cd-reader and start up a terminal window. Mount the cdrom so that the cd can be used.

```
>> mount /cdrom
>> cp /cdrom/kernels/* /usr/src
>> cd /usr/src
>> bunzip2 rtlinux-3[1].2-pre2.tar.bz2
>> tar -xvf rtlinux-3[1].2-pre2.tar
>> bunzip2 linux-2.4.20.tar.bz2
>> tar -xvf linux-2.4.20.tar
```

This will create two new folders, ”rtlinux-3.2-pre2” and ”linux-2.4.20”.

2.0.2 Patching and Compilation of Kernels

The following is valid using the kernels that came with the cd. If other versions are used, the directories will have different names.

```
>> cd /usr/src/rtlinux-3.2-pre2
>> ln -s ../linux-2.4.20 linux
>> cd /usr/src/linux-2.4.20
>> patch -p1 < /usr/src/rtlinux-3.2-pre2/patches/kernelpatch-2.4.20...
>> make mrproper
>> cd /usr/src/linux-2.4.20
>> make menuconfig (or make config or make xconfig)
```

Configure the kernel as wanted, but make sure not to enable APM (Automatic Power Management), and make sure to enable Joliet support under File systems. The two following paragraphs (make dep, make bzImage) may take several minutes to complete.

```
>> make dep
>> make bzImage
>> make modules
>> make modules_install
>> cp arch/i386/boot/bzImage /boot/rtzImage
```

Now edit /etc/lilo.conf and add the following lines below the lines with a similar structure:

```
image=/boot/rtzImage
      label=RTLinux
      read-only
```

also change

```
# propmt
```

to

```
  propmt
```

Save the file and inform LILO of the changes.

```
>> lilo (or /sbin/lilo)
```

Restart the computer (init 6). The boot-choice ”RTLinux” should appear. Choose this.

*** VALID FOR COMPUTER AT UPC BEGINS ***
If the graphics driver did not load properly (no graphical login) the drivers have to be installed. Use the following:

```
>> mount /cdrom
>> cp /cdrom/graphics_driver/NVIDIA-Linux-x86-1.0-4496-pkg2.run /usr/src
>> cd /usr/src
>> chmod 755 NVIDIA-Linux-x86-1.0-4496-pkg2.run
>> sh NVIDIA-Linux-x86-1.0-4496-pkg2.run
```

Just answer ”yes” to everything. When this is done use

```
>> /etc/init.d/xdm restart
```

This will get the graphics back.
*** VALID FOR COMPUTER AT UPC ENDS ***

Continue to set up the RTLinux kernel.

```
>> cd /usr/src/rtlinux-3.2-pre2
>> make menuconfig
>> make dep
>> make
>> make devices
>> make install
```

Restart the computer (init 6)and choose ”RTLinux”. If everything went ok, RTLinux is now installed but NOT ready to be used. Continue to next section.

Chapter 3

Settings before Starting

Mount the cdrom so that the cd can be used.

```
>> mount /cdrom
```

Copy the folder /cdrom/program to any destination on the disk. This manual will use /usr/local/pendulum/ for that folder. The folder /cdrom/program contains three folders and one script. The names should be self-explanatory.

```
>> mkdir /usr/local/pendulum
>> cp -r /cdrom/program/* /usr/local/pendulum
```

3.1 Load the RTLinux Modules

To load the modules require root-rights. Either log in as root or use ”su” when needed. Make sure that the rtlinux modules have been loaded.

```
>> rtlinux status
```

If not started (minus sign after the modules) use

```
>> rtlinux start
>> insmod /usr/src/rtlinux/debugger/rtl_debug.o
```

It is convenient to have the rtlinus modules loaded at start. If this is desired, use the script ”load_rtlinux”. This will copy a script which loads the modules upon boot.

```
>> cd /usr/local/pendulum/
>> ./load_rtlinux
```

3.2 Set User Rights

The devices used for communication must be set so that the user who starts the Java-program has right to read and write on the devices. The script ”set_fifo_rights” fixes this.

```
>> cd /usr/local/pendulum/
>> ./set_fifo_rights
```

3.3 Installation of Java

Copy the file "j2sdk-1_4_2_02-linux-i586.bin" to the directory to which will contain the installation of Java. This directory could be anyone, but consider /usr/local/ as an option. Installation in this directory require root rights. Move to the chosen directory.

```
>> cd /usr/local
>> cp /cdrom/java/j2sdk-1.4.2_02-linux-i586.bin .
>> chmod 755 j2sdk-1.4.2_02-linux-i586.bin
>> ./j2sdk-1_4_2_02-linux-i586.bin
```

To be able to compile and run the Java control program the "PATH" have to include the Java binaries.

```
>> export PATH=$PATH:/usr/local/j2sdk1.4.2_02/bin/
```

Also Java must be informed that the java library on the cd is going to be used. Copy the folder /cdrom/java/javalib/se to anywhere you want. Consider the folder "lib" in the installation folder for Java.

```
>> cp -r /cdrom/java/javalib/se /usr/local/j2sdk1.4.2_02/lib
>> chmod -R 755 /usr/local/j2sdk1.4.2_02/lib/se
>> export CLASSPATH=.:/usr/local/j2sdk1.4.2_02/lib/
```

Do not forget the ".". Ask the system administrator how to make the changes upon boot.

3.4 Compilation and Loading of RT Modules

Each folder except the Java folder contains a make-file that contains compilation information. "cd" to the correct folder and write "make" to compile the .c file(s). On success this will produce a .o file(s). Although, read more before doing it.

3.4.1 rtl_pcidriver

The I/O base address for the PCI card has to be known. Use

```
>> cat /proc/pci
```

This gives a list of PCI information. Look for the text:

```
Bus  0, device   9, function  0:
    Bridge: PCI device 10b5:2497 (PLX Technology, Inc.) (rev 1).
      IRQ 5.
      Non-prefetchable 32 bit memory at 0xe7001000 [0xe700107f].
      I/O at 0xd800 [0xd87f].
      I/O at 0xdc00 [0xdcff].
```

This means that the I/O base address of the PCI card is 0xdc00. This must be set in the code for "rtl_pendmodule.c" Change the line

```
#define BASE_ADDRESS 0xdc00
```

to the new (perhaps same) I/O base address.

If any changes have been done, or if it is the first time the driver is being used, write

```
>> cd /usr/local/pendulum/rtl_pcidriver
>> make clean; make
>> insmod rtl_pcidriver.o
```

3.4.2 rtl_modules

```
>> cd /usr/local/pendulum/rtl_modules
```

If any changes have been done, or if it is the first time the modules are beeing loaded, write

```
>> make clean; make
```

IMPORTANT! Place the cart to the left of the track and wait until the pendulum is hanging straight down. The first thing "rtl_pendcontrol.o" does is to calibrate the cart position and the angle value. Make sure the PWM is switched on and started (red button). When the cart is to the left and the pendulum is not moving write

```
>> insmod rtl_pendmodule.o
```

And if the RT dummy threads are to be used write also

```
>> insmod dummy_module.o
```

To later remove the modules write

```
>> rmmod rtl_pendmodule
>> rmmod dummy_module
```

Notice that the extension ".o" is removed. The system will not allow the removal of the module if it is used, i.e. the Java program is using it. Stop and close any programs using the module before removing it.

3.4.3 Java Program

Compile the program by writing

```
>> cd /usr/local/pendulum/java
>> javac rtlcontrol/rtlpend/*.java
```

Chapter 4

To Use the Program

It is possible to use the features independant of each other, i.e. it is possible to use the RT-thread feature without having to connect to the control module, allthough it is primarily developed to test the behaviour of the controller when it is being disturbed in a controlled way. It is in fact possible to use the the RT-thread feature without even having the pendulum driver or the pendulum hardware connected to the computer. For that reason the user manual is separated in two parts: The use of the control feature, and the use of the RT-thread feature.

4.1 To Control the Pendulum

Due to the separation of the programs it is possible to close the Java program using force (ctrl + c) without disturbing any started control. Therefore, if the Java program for some reason would hang during control, it may be savely closed and restarted. Move to the folder for the file "RunPend.class". To start the program write

```
>> java RunPend
```

This will start up the main control window.

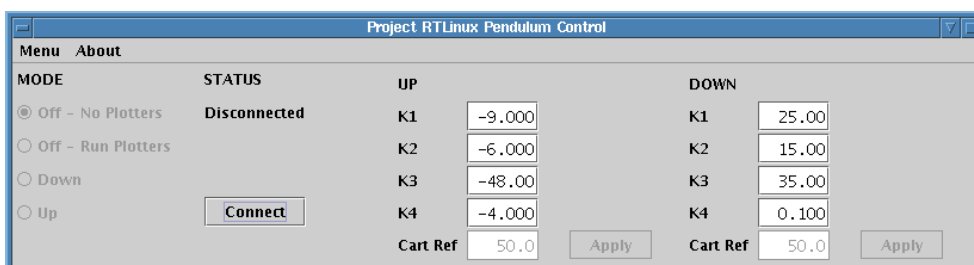


Figure 4.1: *Main control window*

Initially the Java program will not be connected to either the control module or the dummy module.

1. Make sure that the module "rtl_pendmodule.o" is loaded before continuing. Click on the connect-button to connect to the RT control module. If the connections was set up properly the window will change to figure 4.2

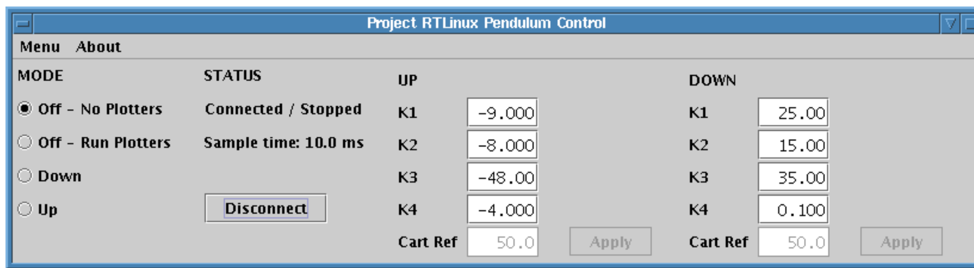


Figure 4.2: *Main control window connected to the RT control module*

Upon connection the program syncronizes all settings to the RT control module. The predefined values is proven to be good.

2. Click ”Menu” and choose ”Control Settings”. This will bring up the window

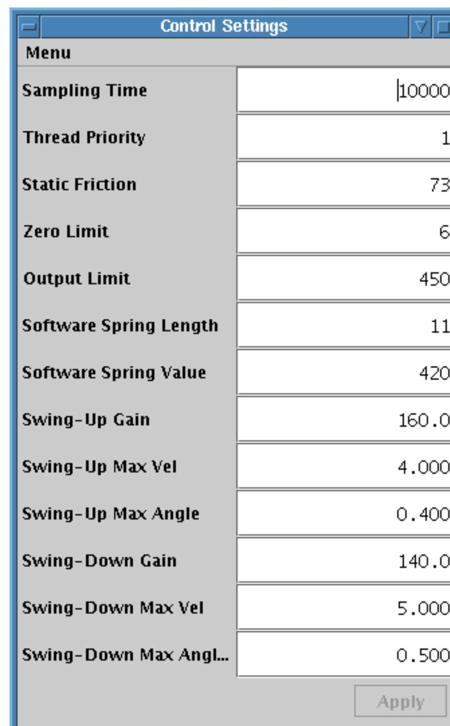


Figure 4.3: *Control Settings*

This is where the basic control settings are set. The following settings are available:

Sampling Time The sampling time for the RT control module expressed in micro seconds.

Thread Priority The priority of the RT control module. The priority range is 0 (min) to 1000000 (max). The priority 0 should in theory execute the thread at the same priority as standard Linux.

Static Friction The minimal force to move the cart (i.e. stiction) expressed in signal unit (su). With the current DC motor the scale is $1su = 0.0145$ N. This is added (with respect to sign) to all control signals.

Zero Limit Any control signal less than this is treated as zero. This is to spare the DC motor from noice.

Output Limit Maximal control signal to the DC motor.

Software Spring Length Length in cm for the software spring. When the cart is within an area covered by the software spring, a force proportional to the distance within the spring-area is applied in direction towards the center of the track.

Software Spring Value Maximal Spring Force.

Swing-Up Gain The control signal used in the Swing-Up.

Swing-Up Max Velocity Maximal allowed angle velocity for the pendulum when switching from Swing-Up to stabilizing control in radians/second.

Swing-Up Max Angle Maximal allowed angle for the pendulum when switching from Swing-Up to stabilizing control in radians.

Swing-Down Gain The control signal usedn in the Swing-Down.

Swing-Down Max Velocity Maximal allowed angle velocity for the pendulum when switching from Swing-Down to stabilizing control in radians/second.

Swing-Down Max Angle Maximal allowed angle for the pendulum when switching from Swing-Down to stabilizing control in radians.

For security reasons many of the fields have limited input ranges.

3. Click "Menu" and choose "View States/Signals" to launch the window with the different plotters (figure 4.4).

   Left-click in a plotter will immediately freeze the plotter, and left-click again will un-freeze it. Right click in a plotter will bring up a menu where it is possible to do the following

   Freeze All Will immediately freeze all plotters.

   Save Current Plotter Save current plotter as a .png or .jpeg file.

   Save All Plotters Save all plotters as a .png of .jpeg file.

   Set Axis of Current Plotter Set the axis of the current plotter and hence the desired resolution. This option is not available for a frozen plotter.

   Click "Up" to start the Swing-Up of the pendulum. Status information will be displayed in the main control window while running. This also show the internal modes, i.e. Swing-Up and Swing-Down. All status information comes directly from the RT module.

   It is possible to go from control in up position directly to control in down position by clicking "Down". It is not necessary to use any off mode to get it down. An energy absorbing mode, Swing-Down, will make the transition smoth and fast.

4. Click "Menu" and chose "Record Data" to bring up the data recorder window (figure 4.5).

   From here it is possible to record data according to the following:

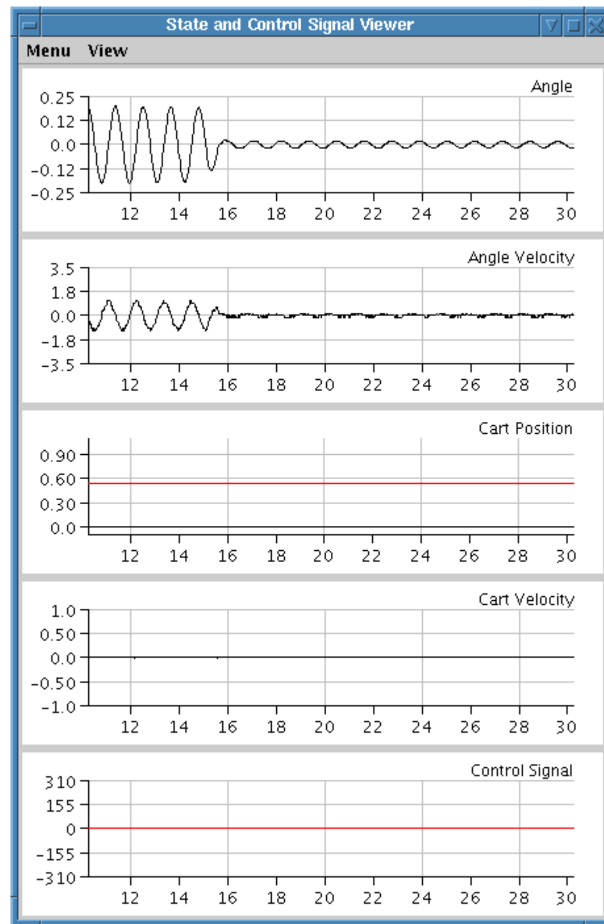


Figure 4.4: *State and Control Viewer*

Record Time Time of recording in seconds. Using the current buffer setting maximal time for a recording is 60 seconds.

Step Time Time of an optional step change in cart position.

Step Size Size of step in cm. A zero step will result in a clean recording without any step change in cart position.

Output File Name of output file. The file will be written to the directory holding the file ”Main.class”.

5. Click ”Menu” and choose ”Settings”. This will bring up the window for setting of devices. Do not change anything here without making the same changes in the RT module (Params.h).

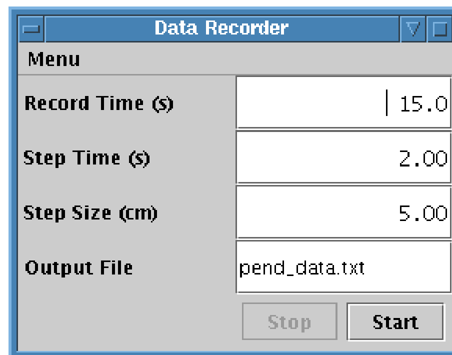


Figure 4.5: *Data Recorder*

4.2 To Use the RT-threads

The main purpose of this feature is to disturb the control thread in a controlled way. It is however possible to use independently, to disturb any other real-time / non-real-time application.

Click "Menu" and chose "View RT Thread Table". This will bring up the RT Thread Table (figure 4.6) from which it is possible to start and stop threads that execute in hard real-time in the kernel. It is also possible to change their parameters, i.e. execution time, computation time and priority.

To connect to the module "dummy_module.o", click "Menu" and chose "Connect". The default number of possible threads to start is 5, but it is possible to change the two files /rtlpend/Params.java and /rtl_modules/Params.h before compilation to get up to 32767 possible threads.

IMPORTANT! It is possible (and very easy) to start RT threads which result in a CPU load that will make standard Linux appear as "hung". This is due to the fact that standard Linux at all times execute with a lower priority than any started RT thread. Avoid this.

For security reasons it is not possible to start a thread which has an execution time longer then the period time.

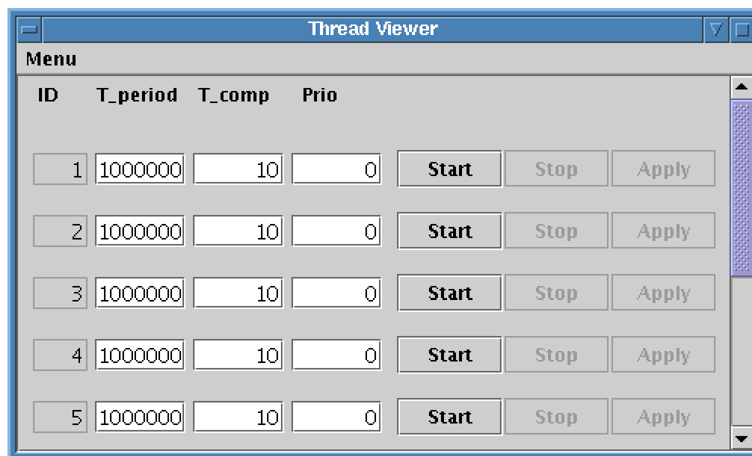


Figure 4.6: *Real-Time Thread Table*