Development of a Software Platform for Real-time Remote Control of an Inverted Pendulum

Mikael Johansson

Master of Science Thesis

Stockholm, Sweden, 2007

ICT/ECS-2007-21

Development of a Software Platform for Real-time Remote Control of an Inverted Pendulum

Mikael Johansson

Supervisor
HIRANO Satoshi
AIST, Tsukuba, Japan

Examiner
Vladimir Vlassov
KTH, Stockholm, Sweden

Master of Science Thesis

Stockholm, Sweden, 2007

ICT/ECS-2007-21

Abstract

Traditionally network middleware for robotic systems are implemented using the C programming language for reasons such as performance and predictability. However, C has a number of problems such as safety and the lack of object-orientation. Using Java for development would solve some of the problems, but there are questions regarding the performance and real-time capabilities of the language.

In this master thesis a remote balanced inverted pendulum system was developed as a proof-of-concept for the use of standard Java in robotic applications. The applications for balancing the inverted pendulum as well as the network middleware used Java. A Xilinx Virtex-II based FPGA-board was used for interfacing with the inverted pendulum hardware.

Acknowledgements

This master thesis project was performed at the Network Middleware Group in AIST. I would like to express my gratitude to the following people:

Dr. HIRANO Satoshi for allowing me to study in the NMG lab, coming up with the idea of the remote balanced inverted pendulum and for his guidance throughout the project.

Dr. OHKAWA Takeshi for all his help, valuable advices and many contributions to the project such as the FPGA design.

Dr. QU Runtao for all his advices and contributions to the project.

Mr. KUBOTA Takaya for his help and contributions to the project, for example, by enabling console-less operation and software re-configuration over a network.

Dr. ANDO Noriaki for his advices on control theory and letting me borrow the equipment that was used in the project.

Dr. Vladimir Vlassov for his guidance on writing this thesis paper.

Finally I would like to thank my family for their support.

Contents

List of Figures

1 Introduction

Traditionally robotic systems have been developed in the programming language C. By using C it’s possible to create very efficient programs. C is also a good language for interfacing directly with hardware. However, programs written in C have the disadvantage of not being very portable and cannot be easily reused. Another problem is the lack of safety and the difficulty of finding bugs which can be easily created due to the memory model of C.

Robotic systems are getting increasingly complex. This creates a need for using object-oriented languages, such as Java, in some parts of the software. Using Java in these parts of a robotic system would give benefits such as re-usability, portability and safety. The problem with using Java in robotics is due to questions concerning efficiency and real-time capabilities of the language.

Robots and other embedded systems need a way to communicate. A network middleware is a type of software that creates an infrastructure for applications to communicate. To reduce the complexity and increase the extensibility of a robotic system various parts of the software are often modularized. These parts can then be connected by using a middleware such as CORBA. Today's middleware for robotics is often written in C. Java might be a better choice for writing this type of software.

The purpose of this project is to explore topics such as:
- Real-time capabilities of Java in general
- Real-time capabilities of Java-based middleware
- Performance of Java in embedded and robotic systems

An inverted pendulum can easily be described as a robot-arm which balances a pen in the palm of its hand. The goal is to keep the pen in the up-right position and thus keep the pen from falling down. The inverted pendulum is commonly used in control theory experiments. In control theory research the inverted pendulum is used to test new control algorithms in a safe environment.

For this project an inverted pendulum will be built for the purpose of demonstrating the performance of various Java-based network middleware. The inverted pendulum will be balanced remotely over a network using a middleware for communication. A Xilinx Virtex-II Pro FPGA-board will be used for interfacing with the pendulum. Java software will be used both within the FPGA-board and in the remote computer which implements the balancing algorithm.

1.1 Motivation

The project will examine whether Java and Java-based network middleware can provide the performance and real-time capabilities needed for robotic applications. Moreover, the inverted pendulum will make an excellent visual demonstration tool for real-time characteristics of a network middleware. If the control loop of the inverted pendulum would miss its deadline the pendulum will fall down.

Another purpose of the project is to explore how Java can be used in embedded systems, and in particular in a Xilinx Virtex-II Pro based FPGA-board. During the project a platform for distributed Java-based robotic systems will be developed. The platform will be used for developing the inverted pendulum and can also be used for developing future Java-based robotic applications.

1.2 Thesis Objectives

Building the Inverted pendulum. An inverted pendulum will be built from scratch. The work will involve design of the electrical and mechanical parts of the system as well as developing the control-algorithm for balancing the pendulum.

Development of the Java-based Platform. The main issue will be to find a way to run Java within the CPU-core of the FPGA-board. The problems are that the FPGA-board has a very small memory and that Linux-variant used is not compatible with standard JVMs.

Remote balancing the Inverted Pendulum. The inverted pendulum will be remote balanced using a Java-based middleware for communication. Two questions should be answered. The first question is whether Java can be efficient enough and provide the real-time needed for balancing the pendulum even locally. The second question is whether a Java-based middleware can provide the round-trip time needed for remote balancing.

1.3 Requirements

At the start of the project the following requirements were specified.

- The inverted pendulum should be built from scratch using standard electrical components.
- Standard Java SE should be used in all software.
- A Xilinx Virtex-II Pro based FPGA-board should be used for interfacing with the pendulum.
- The inverted pendulum system should be implemented and evaluated in two different Java-based network middleware-technologies: HORB and JavaRTM.

1.4 Structure of Thesis

This report is based on eight chapters.

Chapter one describes the motivation and objectives of the project.

Chapters two gives background to embedded system, real-time systems and programming languages.

Chapter three describes the overall design of the remote balanced inverted pendulum system.

Chapter four describes the development of the platform.

Chapter five explains how the inverted pendulum was built; the electrical work and the balancing controller design and implementation.

Chapter six shows how the remote balancing was implemented in two network middleware: HORB and JavaRTM.

Chapter seven presents an evaluation of HORB and JavaRTM.

Chapter eight provides conclusions of the project.

2 Background

The chapter gives an introduction to subjects which were important in the development of the inverted pendulum platform: embedded systems, real-time systems and a survey of issues in implementing such systems using the programming languages C, Java and Real-time Java.

2.1 Embedded Systems

This section describes what an embedded system is, explains some of the challenges in embedded system design and gives an overview of the choice of hardware for implementing an embedded system.

Let's start by giving a definition of an embedded system: An embedded system is a computer which is designed to perform a specific function.

When describing an embedded system it's helpful to contrast it with a general-purpose computer system such as a desktop PC. Whereas a desktop PC can be used for running various applications an embedded system is built for performing a specific function. Software for a general-purpose computer can run in any computer of the same architecture. However, in an embedded system the software and hardware is tightly coupled to create a unique device.

A developer of software for a general-purpose computer is only concerned with the function of the application – the computer is just seen as a platform. On the other hand, a designer of an embedded system must decide which microprocessor that should be used, design the hardware platform with I/O devices to support a required task and implement the software. [5]

2.1.1 Challenges of embedded system design

The designer of an embedded system faces many challenges. In [5] Wolf describes some of the problems that must be taken into account in embedded system design:

- **Choice of hardware.** Hardware must be chosen both to meet the performance requirements and minimize the cost of the product.
- **Minimize power consumption.** In some embedded system, for example battery-powered ones, the power consumption is very important.
- **Design for upgradeability.** The hardware platform might be used for several product generations. The platform needs to be designed with future software changes in mind.
- **Complex testing.** It's often not possible to separate the testing of an embedded computer from the machine in which it is embedded.
- **Limited observability.** Since embedded systems usually don't have keyboards and monitors it becomes more difficult to know what goes on inside the system. The system often has to be observed for example by watching the values on the bus.
- **Restricted development environments.** The development environments for an embedded system are usually more limited than those for a general-purpose computer. Debugging also becomes more difficult.

2.1.2 Hardware design

In many embedded system a microprocessor is used. However, there are many ways of implementing a digital system: custom logic, FPGAs, and so on.

In [10] Deschamps discusses the selection of hardware for an embedded system. The selection of hardware should answer the following question: “How do we get the desired behaviour at the lowest cost, while fulfilling some additional constraints?” Cost can cover several aspects such as: the unit production cost, the nonrecurring engineering costs, or a cost for late introduction of the product to the market.

Deschamps further describes how to choose between microprocessor, DSP, FPGA and ASIC when implementing an embedded system.

A microprocessor should be used for systems that require little data processing capability. For such systems microcontrollers and low-range microprocessors are usually the best choice.
A DSP should be used for systems which has a greater computation need.

Using a DSP is a flexible solution since the development work mainly consists of developing a program. A more powerful microprocessor could also be used for these systems.

When even higher performance is needed a specific circuit might be necessary. A first option can be to use an FPGA. It’s a good option for prototyping and production of small series. For large series an application-specific integrated circuit (ASIC) should be developed.

As mentioned, today the most common choice for implementing an embedded system is by using a microprocessor. In [5] and [8] the reasons for implementing an embedded system using a microprocessor is explained in detail:

- **Cost.** Implementing a digital design using a microprocessor is often much faster and less expensive than implementing it in custom logic (such as FPGA or ASIC).

- **Flexibility.** Some changes might be needed after the product has been finished. For a hardware-based system the electronics would have to be re-designed. If the system is implemented using a microprocessor only a few lines of code would have to be changed.

- **Programmability.** A robotic arm might be used to paint car doors one day and sort packages the next. By using a microprocessor it’s possible to enable the user to re-program the system to perform different tasks.

- **Design families of products.** High-end products can be created by extending the software without changing the hardware.

2.1.3 FPGA

This section gives an introduction of FPGA technology since it’s used for implementing the inverted pendulum platform.

An FPGA is a programmable logic chip that can quickly be customized to a particular function. FPGAs are powerful enough to implement almost any hardware design. In the past FPGAs were marketed primarily for two purposes [11]:
1. For prototyping of hardware that will later be implemented as an ASIC.
2. To achieve quick time-to-market since implementing hardware in FPGA in much faster compared to implementing the hardware as an ASIC.

As the speed of FPGAs increased and power consumption decreased, FPGAs also started to be used in final product designs.

2.1.3.1 FPGA Cores

An FPGA Core can be described simply as a self-contained digital function. There are two types of cores. A soft core, also known as an IP Core, describe a logical function but without a physical implementation. Soft cores are described in a hardware description language (HDL). In constrast, embedded cores are the physical implementation of a function. In FPGAs these are often physically embedded into the FPGA chip.

IP Cores: IP cores are often sold by third-party vendors that specialize in a specific field. There are simple IP Cores implementing functions such as a multiplier and more advanced ones, such as a JPEG decoder/encoder or an Ethernet controller. By taking advantage of IP Cores the FPGA designer can save much time since the IP cores are already designed and verified. Many IP cores are also modifiable which allows the designer to make custom changes. An FPGA designer working on some project may choose to encapsulate new functions as IP Cores which gives him re-usable hardware components for future projects. There are also many free IP Cores available from websites such as Opencores[12].

Embedded Cores: An example of an embedded core can be an analog device. Some analog devices cannot be designed in an FPGA so therefore they are instead embedded into the FPGA chip. By choose an FPGA with the needed devices already embedded will save space for chips that would otherwise be needed outside the FPGA. An embedded core also typically offers good performance and power consumption compared to an IP core since the embedded core is often optimized for a particular FPGA chip.

Processor Cores: Processor cores consist of a full-fledged CPU which is run inside the FPGA. Processor cores are available both as IP Cores and embedded cores. Embedded processor cores have the advantage of being optimized for a particular FPGA and thus are more predictable and have lower power consumption than IP core ones.

2.2 Real-time Systems

This section introduces some important concepts of real-time systems.

2.2.1 Definition of a real-time system

In a real-time system the response time has to be guaranteed. Usually a timing constraint specifies the deadline of the response time. If the real-time system could not meet the timing constraint it’s as worse as producing the wrong response. [1]

2.2.2 Soft versus Hard real time

Real-time systems are often distinguished between hard and soft real-time systems. In hard real-time systems it’s critical that the response time is within the specified deadline, or else a complete failure will occur. In a soft real-time system the timing constraints are still important, but missing a few deadlines will not cause the system as a whole to fail. In [2] Liu, explains that missing a few deadlines in a soft-realtime system do not serious harm the system but the overall performance of the system becomes poor.

2.2.3 Characteristics of a real-time system

The characteristics of real-time systems are summarized by Wellings in [4] as:

- **Large and complex.** Real-time systems vary from small embedded systems to large multi-platform distributed systems.

- **Extremely reliable and safe.** Many real-time systems are critical systems such as air traffic control. These systems need to attempt to tolerate faults and continue to operate.

- **Real-time facilities.** The language and run-time environment should enable the programmer to access a clock, delay execution, program timeout and specify scheduling.

- **Interaction with hardware interfaces.** Most real-time systems need to interact with the real word, for example, to monitor sensors or control actuators.

- **Efficient implementation and a predictable execution environment.** Since real-time systems are time-critical the efficiency of implementation is very important.

2.3 Programming Languages for Real-time Embedded Systems

This section describes the advantages and disadvantages of C and Java. It also gives an introduction to the new features of Real-time Java.

2.3.1 C

C has been the most popular programming language for embedded systems since the 1980. There are several reasons why [9]:

- C has a simple execution model. There's a constant cost for each C construct. A good C programmer can approximate the assembly code that will be generated for a specific C construct.

- Since the language has been around for a long time there are many tools available for many different architectures. For the same reason the language is also stable and standardized.

- The language is closely tied to hardware execution. For this reason C is often called a "low-level" high-level language. C gives the programmer direct hardware control. It's easy to control for example memory-mapped peripherals and registers from C, without resorting to assembly language.

C also has several disadvantages:

- C has no type-safety. This means it's possible to assign variables of different types without any warning or compilation error. The lack of type-safety is a big barrier to productivity [9]. For example, to debug a pointer which is "lost" can be very time-consuming. Burns and Wellington argues in [1] that: "Strong typing is universally accepted as a necessary aid to the production of reliable code. The absence of strong typing in C is one drawback to the use of that language in high-integrity-environment." However, most C compilers can be tuned to catch typing mismatches. [3]

- The absence of object-orientation makes reuse difficult. Another disadvantage is that the memory allocation model in C (malloc-based) is too primitive. This makes identifying memory leaks very difficult.

- C also has no explicit exception-handling such as for example Java. However, in [3] Laplante argues that C provides a type of exception handling through the use of signals.

2.3.2 Java

Java typically describes the entire platform consisting of the language, the virtual machine and the collection of library APIs. Java's execution model is very different from C. Java is designed to be compiled to an architecture-neutral representation, called Java byte code, which can be executed by the Java virtual machine running on a target system. Even though Java was designed to be interpreted other ways of compiling the language for performance have appeared.
Just-in-time (JIT) compilers compile methods as they are loaded and run on the target system. Ahead-of-time (AOT) compilers compile programs in the more traditional way. In AOT compilers the byte code is precompiled to native code before it is run in the target system.

Many of Java's features address the problems of C in terms of safety and developer productivity. These are the most important advantages of Java:

- Object-oriented programming, which provides one of the main tools that programmers can use to manage large software systems. [1]

- Target independence is achieved by using Java byte code. This enables reuse of the same software between different platforms and CPUs.

- Automatic memory management greatly increases the productivity since the developer does not have to worry about forgetting to free up objects and reclaim memory. This also increases safety of the software since memory leaks are less likely to occur.

- Type safety is important since it removes the cause for many bugs. It enables protection against overwriting arrays which is a dangerous security bug which can occur in C.

- No explicit pointers avoid many types of bugs which can occur in C. Without pointers programs become much safer and easier to debug.

- Java offers built-in security in the form of byte code-verification. Applications can run in a restricted environment known as a sandbox to protect systems that need high integrity.

However, Java has some issues which make it difficult for use in embedded systems. These are the issues Java face in embedded and real-time systems:

- Portability: For Java to be able to run in some target system a JVM has to exist for that platform. However, for many embedded systems a JVM might not be available. In practice a JVM has to be ported to a specific CPU and operating system.

- System requirements: For good performance Java requires a fast CPU. Another problem is the size of the JVM and the Java class-libraries. These often require a few megabytes in size (for J2SE) which is too much for most embedded systems.

- Low-level programming: Since Java is designed for safety and platform independence operations such as access to memory cannot be done in Java. The Java native interface (JNI) provides a solution for this issue.

- Garbage collection: Automatic memory management comes at a price. While the garbage collection algorithm run the Java program will completely stop. This makes it difficult to use Java in real-time systems.

- Thread management: The thread scheduling in Java is not suitable for real-time systems since it is missing features such as preemption, priority-based synchronization of serial resources (to handle priority inversion), handling of asynchronous events, fast asynchronous transfer of control, and safe thread termination [9].

2.3.3 Real-time Java

To solve the issues regarding the use of standard Java in real-time systems the RTSJ were specified. The RTSJ is defined as a new API with support from the JVM [16].

RTSJ enhances Java in the following areas: memory management, time values and clocks, schedulable objects and scheduling, real-time threads, asynchronous event handling and timers, asynchronous transfer of control, synchronization and resource sharing and physical and raw memory access [4].

3 System Design

This chapter describes the overall design of the system. At first the important parts of the system are identified and introduced. The system is then de-composed into layers and the responsibility of each layer is described. The dynamic behavior of the system is also described.

3.1 Overview

As shown in figure 1 the system consists of three main parts: 1) the inverted pendulum, 2) the FPGA-board which interfaces with the electronics of the pendulum and 3) the balance controller which is a program running in a PC. The pendulum interface and the pendulum controller are connected to a network and use a Java-based middleware for communication.

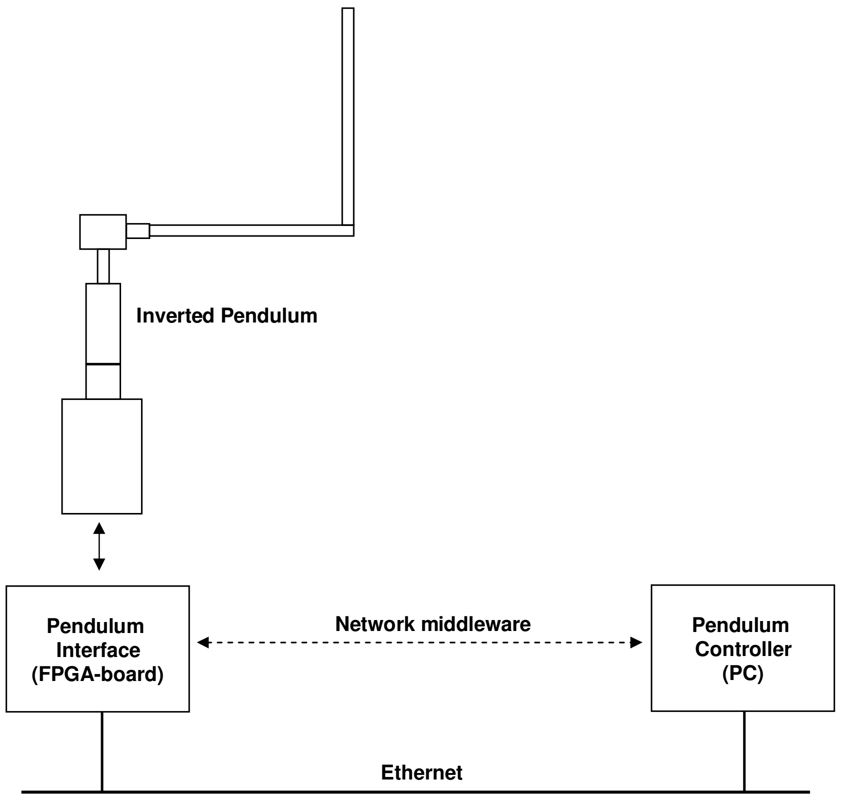


Figure 1: System overview

Each part of the system is briefly described below.

Inverted Pendulum. An inverted pendulum is a device that will to balance a stick in an upright position. The stick is attached to a rotary encoder which senses the angle of the stick. The pendulum arm is driven by a motor. Since the stick can rotate freely the motor has to adjust the position of the pendulum arm to balance the stick. The type of inverted pendulum that will be built for this project is known as a rotary inverted pendulum (or Furuta pendulum).

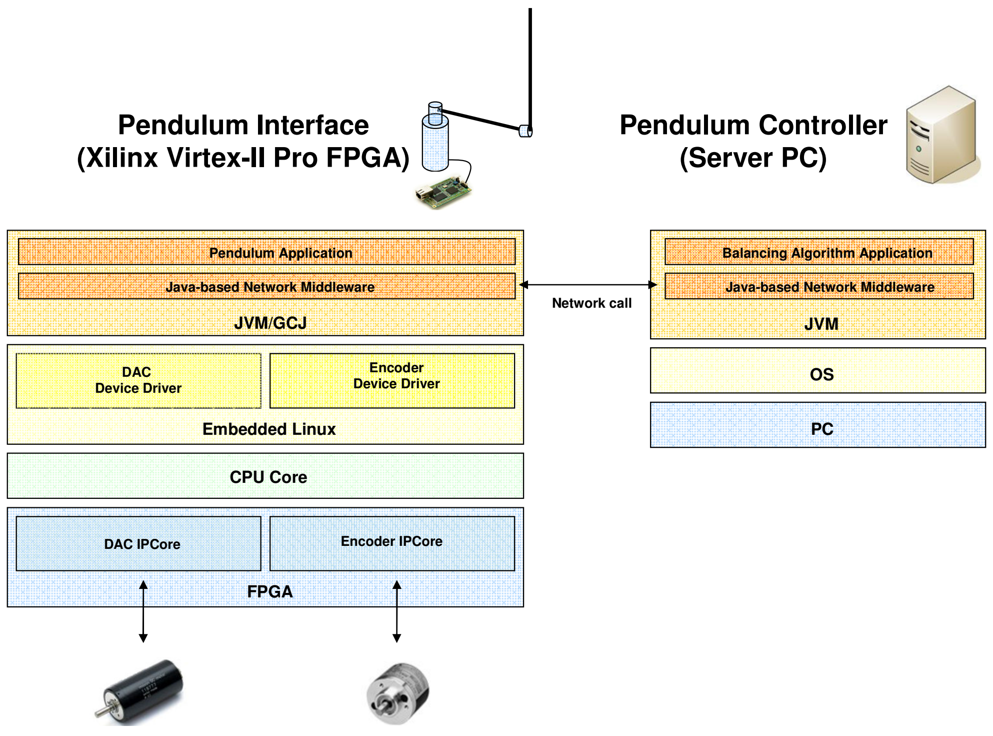
Pendulum Interface. The electronics of the inverted pendulum is interfaced from a FPGA-board. Software running within the CPU core of the FPGA-board will read the state of the pendulum and send it to the balance controller. The balance controller will return a value which describes how the motor should move to balance the stick.

Balance Controller. The Balance Controller is a Java-software which implements the algorithm for balancing the inverted pendulum. The algorithm is presented in section 5.3.

Java-based Network Middleware. A network middleware is used to connect the pendulum interface software and the balance controller software. The purpose of the network middleware is to provide the communication infrastructure for the applications. To different network middleware are used: HORB and JavaRTM.

3.2 Layers

This section shows a more detailed design of the system. Figure 2 shows the layers of the pendulum interface and the balance controller.



Pendulum Interface (Xilinx Virtex-II Pro FPGA)
Pendulum Controller (Server PC)
Pendulum Application
Java-based Network Middleware
JVM/GCJ
Network call
Balancing Algorithm Application
Java-based Network Middleware
JVM
OS
PC
DAC Device Driver
Encoder Device Driver
Embedded Linux
CPU Core
DAC IPCore
Encoder IPCore
FPGA

Figure 2: System layers

3.2.1 Pendulum Interface

The pendulum interface logically consists of four layers: FPGA, CPU Core, Embedded Linux and JVM.

FPGA Layer. The FPGA-layer is the FPGA-chip of the FPGA-board. Within this layer electrical circuits for interfacing with external hardware, such as actutators and sensors, are located. In the solution two IPCores are used. The DAC IPCore does digital-to-analog conversion for the purpose of controlling the motor. The Encoder IPCore read the quadrature-signal of the encoders.

Embedded Linux Layer. On top of the CPU Core of the FPGA-board an embedded Linux operating system is running. Within this layer the device drivers for the DAC and encoders are defined.

JVM Layer. The JVM layer provides a Java runtime for the pendulum interface. Within the JVM-layer two Java-applications will run. A Java-based middleware provides the pendulum interface with a communication method. The other application is the pendulum interface application. The pendulum interface application uses the middleware to communicate with the pendulum controller.

3.2.2 Pendulum Controller

The pendulum controller logically consists of three layers: PC, OS and JVM.

Within the JVM-layer two Java-applications will run. As for the pendulum interface a Java-based middleware will provide a communication method. The other application is the pendulum controller application. This application has the purpose of implementing a balancing algorithm. The pendulum controller application will receive the state of the pendulum and calculate a value which describes how the motor should move.

3.3 Remote Balancing Design

This section describes how the pendulum interface and the pendulum controller interact to remote balance the inverted pendulum. The internal behavior of the two components is also described.

3.3.1 Overview

Figure 3 shows the interaction between the pendulum interface and the pendulum controller.

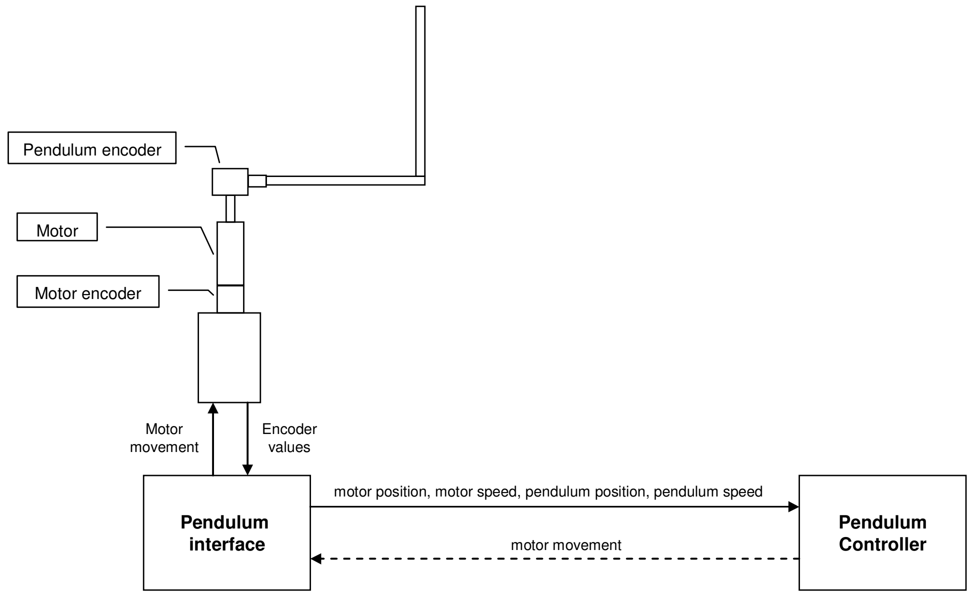


Figure 3: Interaction between pendulum interface and controller

As shown in Figure 3, the pendulum interface interacts with both the inverted pendulum hardware and the pendulum controller. The pendulum interface will continuously read the encoders of the pendulum and then send the state of the pendulum to the controller. The pendulum controller will retrieve the pendulum state from the interface, and return a value which describes how the motor should move.

3.3.2 Pendulum Interface

Figure 4 shows the internal behavior of the pendulum interface software.

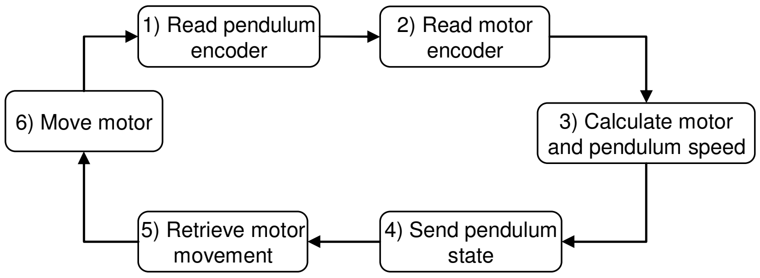


Figure 4: Activity diagram of pendulum interface behavior

The pendulum interface software will first read both the pendulum and motor encoder. It will then calculate the speed of the motor and pendulum by using previous encoder values. The pendulum controller will send the pendulum state (consisting of motor position, motor speed, pendulum position, pendulum speed) to the pendulum controller. The pendulum controller will then wait until it has retrieved a value which describes the motor movement. When the motor movement value has been received the pendulum controller will move the motor. After the motor has moved the procedure is repeated.

3.3.3 Pendulum Controller

Figure 5 shows the internal behavior of the pendulum controller software.

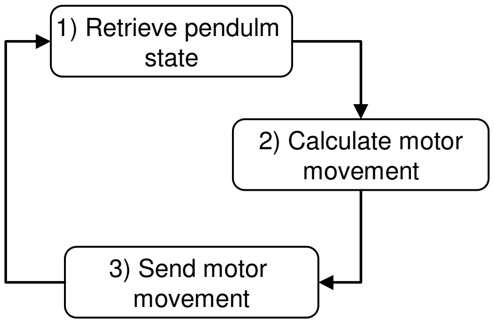


Figure 5: Activity diagram of the pendulum controller behavoir

The pendulum controller will first wait until it receives a pendulum state from the pendulum interface. When it has been received it will use the pendulum state to calculate a new motor movement. The design and implementation of the calculation is a control theory problem which is explained in sections 5.3 and 5.3. Once the motor movement has been calculated the pendulum controller will send the motor movement value back to the pendulum interface.

3.4 Protocol

The communication protocol between the pendulum interface and the pendulum controller consist of three commands:

- Start
- Stop
- Remote balancing command

Figure 6 shows how the commands will be typically used. At first the start command is sent to start remote balancing. The remote balancing loop starts. The pendulum state is continuously sent and the motor movement is returned for balancing the pendulum. After some time the stop command is sent to stop the balancing loop.

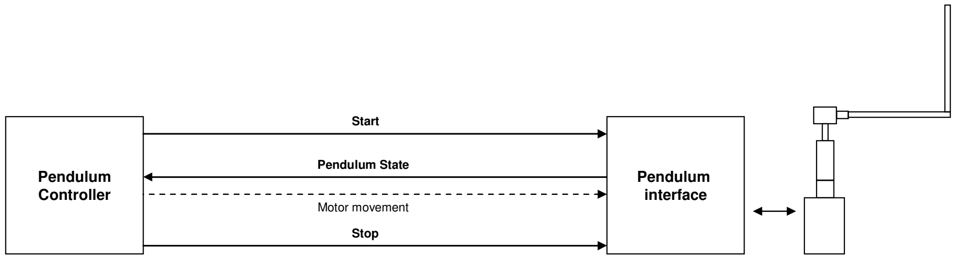


Figure 6: Protocol outline

The protocol commands are described further below.

3.4.1 Start command

The start command is sent from the pendulum controller to the pendulum interface to start the remote balancing loop.

3.4.2 Stop command

The stop command is sent from the pendulum controller to the pendulum interface to stop the remote balancing loop.

3.4.3 Remote balancing command

The remote balancing command has the purpose of remote balancing the inverted pendulum. During remote balancing the command is continuously sent. The command is made up of two messages: a request and a response. In the request message the state of the pendulum is sent. The pendulum state contains four values: motor position, motor speed, pendulum position and pendulum speed. In the response message a value which describes motor movement is sent back.

Request (Pendulum state)	***Motor position***	***Motor speed***	***Pendulum position***	***Pendulum speed***
	4-bytes integer	4-bytes float	4-bytes integer	4-bytes float
Response (Motor movement)	***Motor movement***			
	4-byte integer			

4 Platform

This chapter focuses on the development of a Java-based platform which will run on the FPGA-board. The purpose of the platform is to interface with electronics of the pendulum and provide a Java runtime environment for the pendulum software. The platform should enable Java-applications to easily make use of hardware connected to the FPGA-board.

In detail, the platform which was developed consists of the following parts:

- **Java runtime on FPGA.** The platform provides a solution for how to efficiently run Java within a Xilinx Virtex-II Pro based FPGA-board.
- **Cross-development environment.** The platform also provides a cross-development environment for Java-programs. The environment includes an IDE as well as the cross-compiler for producing native code for the PowerPC 405 CPU.
- **Access hardware from Java.** A solution for how to access hardware of the FPGA-board from a Java-program is provided.
- **Device drivers.** Device drivers for motor control and sensor measurements were developed.
- **Network middleware.** Two Java-based network middleware were ported to the FPGA-board.

Once the platform was built it was used in the development of the remote balanced inverted pendulum.

4.1 Overview

Figure 7 shows how the platform is structured.

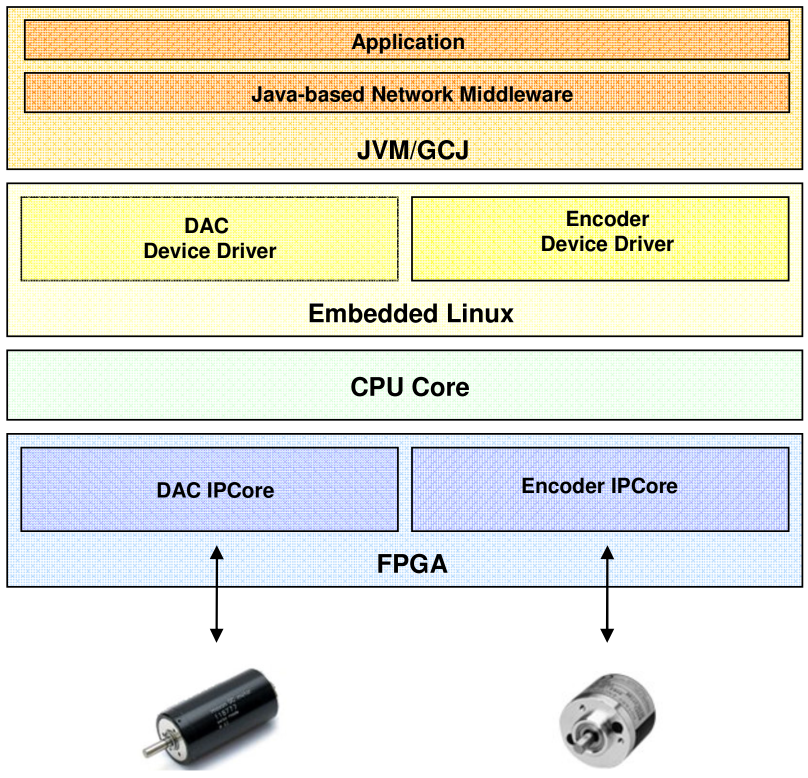


Figure 7: Platform overview

As mentioned in the previous chapter the platform consists of four layers: FPGA, CPU Core, Embedded Linux and JVM. The following sections describe what was done in each layer in the development of the software platform.

4.2 FPGA-board

This section introduces the SUZAKU-V FPGA-board which is hosting the platform.

The SUZAKU-V FPGA-board is manufactured by Atmark Inc. The most important feature of this FPGA-board is that it combines a FPGA-chip and CPU core on the same board. The FPGA-chip and the CPU Core are tightly integrated and are both connected to the internal bus of the FPGA. This makes it possible to efficiently use the features of the FPGA from a program running in the CPU. Another important feature of the board is that it comes prepared with an embedded Linux. Moreover, the board includes a 100MBit network chip. These features make it possible to create exciting applications:

- The FPGA-chip enables integration to any electrical components.
- The Linux programming environment makes it easy to quickly develop programs which make use of the power of the FPGA.
- The network chip enables the device to be used in a network environment.

The FPGA-board is based on the Xilinx Virtex-II Pro FPGA. The FPGA has a CPU core with a 270MHz PowerPC 405. The board has 32MB of RAM and 8MB flash memory. As mentioned, an embedded Linux operating system, uCLinux, comes installed with the board.

Figure 8 shows the various parts of the FPGA-board.

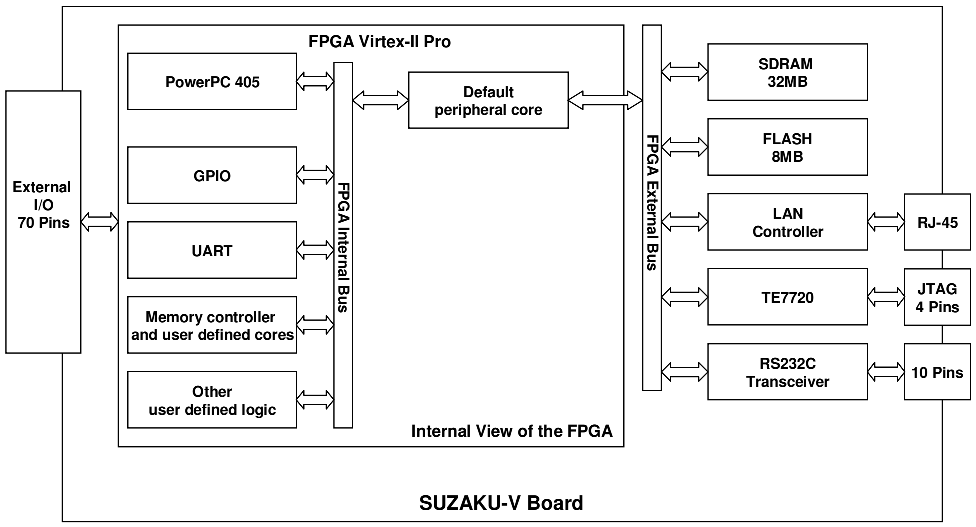


Figure 8: SUZAKU-V FPGA-board

As shown to the left in the figure the board has 70 I/O pins which can be used to integrate external circuits. To the right in figure the standard connectors of the board are shown: RJ-45 for the network connection which is connected to the LAN controller, JTAG for programming the FPGA-chip which is connected to the TE7720 and RS232C for serial-communication. The block to the left in the figure shows the internal view of the Virtex-II Pro FPGA. Inside the FPGA is embedded CPU Core. The logic of the FPGA and the CPU Core are connected through the internal bus of the FPGA. The external bus of the FPGA integrates components such as memory and LAN controller. The GPIO (general-purpose input/output) is as a general way of connecting physical pins on the FPGA to custom-logic.

4.2.1 Embedded Linux

The FPGA-board comes with an embedded Linux distribution called uCLinux. uCLinux is a Linux distribution made particularly for embedded devices which often has a very small memory. The uCLinux-package that comes with SUZAKU-V only requires about 5MB of storage for the kernel, libraries and the user-programs. One of the reasons that the Linux distribution can have been made this small is that a special C-Library, designed for embedded devices, called uCLibc is used. uClibc only requires a few megabytes of storage. Common Linux distributions use the GNU C-Library (glibc) which requires about 20 MB.

The SUZAKU-V uCLinux package comes complete with drivers for the network chip and other software that is needed to run Linux of the FPGA-board. The package also includes a cross-compilation environment for development of C-programs.

4.2.1.1 Permanent storage of the FPGA-board

The FPGA-board only has 8 MB of flash memory. The flash memory is used for permanent storage and contains the Linux operating system. The Linux kernel and configuration data occupies about 5 MB which leaves about 3 MB for user programs to be added to the permanent storage.

Since flash memory cannot be written at random, like a hard drive, it gives the effect that the file system of the device will become read-only. To add something to the file system permanently you first need to add it to the file system image. The file system image is usually located in the host PC. The image can then be written to the flash memory of the device. It is possible to temporary create and modify files in the /tmp directory of the file system of the device since this directory uses the SDRAM for storage. However, this data will be lost after a reboot.

During development there is a need to move files back and forth to SUZAKU many times. To first make a change in the file system image and then writing the image to the flash is a very time-consuming process. Using ftp to temporary transfer files to SUZAKU is an alternative. The problem is that files transferred this way will be lost after the next reboot. It was found that NFS is the easiest way for avoiding the problems of the read-only file system. By using NFS it is possible to make changes in the mounted directory and keep the changes after rebooting the embedded computer. The uCLinux-package shipped with the FPGA-board has a bug which makes NFS fail when used in a normal way. Appendix C describes how to avoid this bug.

4.3 Device Drivers

This section describes the development of device drivers for motor control and encoder measurement.

4.3.1 Background

Generally device drivers provide user-programs with an interface to access hardware. Device drivers should take care of the technical details of a device so that an application programmer can concentrate the application logic.

In Linux, device drivers can either be statically built into the kernel or linked to the kernel as modules during runtime. User programs are said to be run in user-mode with very restricted access. They rely on the operating system to provide its services. However, device drivers run in kernel-mode side by side with the operating system.

There are three different types of device drivers: character devices, block devices and network devices. Character devices are the simplest ones. User-applications communicate with a character device driver by writing or reading in a stream of bytes to a device file. The device file appears as a file in the file system. When the device file is accessed the operating system steps in and maps the data to the device driver code. The mapping is done through the use of major numbers. Each device file in the file system has a major number. The corresponding device driver code registers to the operating system by giving its major number.

4.3.2 Device drivers for the pendulum interface

The pendulum interface will be interfaced to the electrical components of the pendulum. The pendulum interface application will need to:

- Control the motor movement
- Read the encoders of the inverted pendulum

Since the pendulum interface application is run in user-space it cannot directly access hardware. Device drivers are needed to provide the application with this functionality.

Figure 9 shows how the pendulum interface application will access the device drivers which in turn will access IPCores within the FPGA.

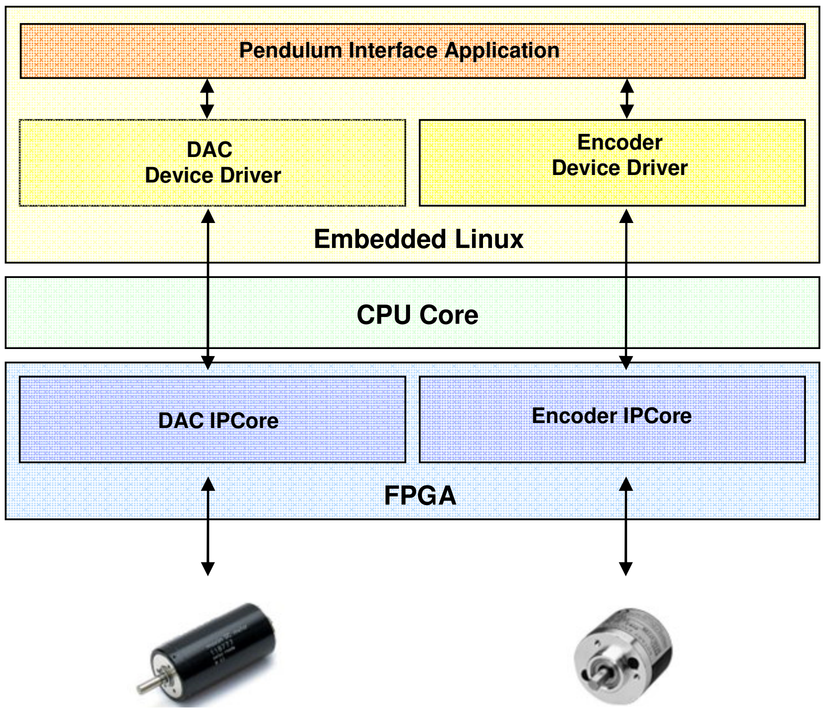


Figure 9: Device drivers

Two device drivers will be developed: the DAC (digital to analog converter) device driver and the encoder device driver. The function to the drivers are described briefly below:

- The DAC device driver will be used for motor control. The pendulum interface application will write some values to the device driver which specifies a motor movement. The DAC device driver will use the DAC IPCore, which is accessed through memory-mapped registers, to output a voltage which controls the motor.
- The encoder device driver will be used for reading the position of the rotary encoder. The application will read the positions from the encoder device driver. The encoder device driver will access the encoder IPCore to retrieve the positions. The encoder device driver also uses memory-mapped registers to access the IPCore.

The device drivers will be developed as modules that can be linked into the kernel at runtime. This is done to avoid having to re-build the kernel every time a change is made to the device driver code.

4.3.3 DAC device driver

The DAC device driver has the purpose of outputting a voltage which can be used to control the motor. The DAC IPCore is explained further in the chapter about electrical design. However, the interface of the DAC IPCore is described below.

The DAC IPCore can output a voltage between -3.3V and +3.3V. The voltage can be of both positive and negative direction to make the motor turn in both directions. For example, positive output makes the motor turn right and negative makes it turn left.

The DAC IPCore has two memory-mapped registers called DAC1 and DAC2. The registers each accept a two bytes value which defines the voltage output. For positive output a value should be written to DAC1 and zero should be written to DAC2. For negative output a value should be written to DAC2 and zero should be written to DAC1.

At another memory-mapped register a one byte a value can be written to tell the DAC IPCore to start output the specified voltage.

The interface of the DAC device driver was decided as follows. A four byte value should be written at a time to the device driver. The four bytes value has two parts. The first two bytes specify the value for DAC1. The second two bytes specify the value for DAC2. If any other number of bytes than four is written to the device driver an error will be returned.

When a user-program writes four bytes with values for DAC1 and DAC2 the device driver will copy these values to the memory-mapped registers of DAC1 and DAC2. The device driver will then write a value to the one byte control register to signal to the DAC IPCore that it should start output the new voltage.

4.3.4 Encoder device driver

The encoder device driver has the purpose of reading values from the rotary encoders. It supports two encoders.

The interface of the encoder IPCore is as follows. The encoder IPCore has two memory-mapped registers: ENC1 and ENC2. The registers are both four byte in size and hold the current position of the two rotary encoders.

The interface of the encoder device driver was specified as follows. The application should read 8 bytes at once from the device driver. The 8 bytes are divided into two parts. The first part is the four bytes with the position of encoder one. The second part is the four bytes with the position of encoder two.

The function of the device driver is very simple. When the application reads 8 bytes from the device driver the driver copies 4 bytes from each of the memory-mapped register ENC1 and ENC2 and returns it to the application.

4.4 Running Java on the FPGA-board

This section describes the work which was done to find a JVM which could run on the FPGA-board.

4.4.1 Problems with getting Java to run on the FPGA-board

There are two main problems with getting Java to run on FPGA-board. The first problem is due to the small size of the flash memory. The second problem is that most java-distributions are compiled as binaries which use glibc (while uCLinux uses uClibc). These problems are described in detail below.

Flash memory size
In the default configuration of uCLinux (which is shipped with the FPGA-board) around 3 MB are available for user programs. Such a small memory size is not enough for most

JVMs. Only the java-libraries required by a JVM require about 8-9 MB (in the case of GNU classpath).

Glibc and uClibc
Most Linux-distributions uses the GNU C library (glibc). Glibc is primarily designed for portability and performance. uCLibc, on the other hand, was developed to be used for embedded systems and thus to be as small as possible. The FPGA-board is shipped with uCLinux which uses uClibc. The problem is that many JVMs are distributed as binaries and already compiled for glibc.

4.4.2 Approaches

Three approaches were tried to deal with the small size of the flash memory:

- Used a stripped down version of the Java SE class library.
- Load the JVM and java-classes over NFS.
- Used GCJ to compile the java-program to native code (and avoid the huge java-library).

The first approach is used by Mika, which is a commercial JVM based on Wonka and targeted for embedded systems. Mika's JVM and java-classes together require only about 2.5 MB.

The second approach, to load a big JVM over NFS, may be a possible way. However, problems with uCLibc and glibc still remain and of course the performance penalty in loading files over NFS.

The third approach, to use GCJ, also worked well. However, to build a working GCJ was tricky. Appendix C describes how it was done.

4.5 Accessing Hardware from Java

This section describes a simple method for accessing a Linux device driver from Java.

Using C it's easy to access a device driver. It can be done by simply using the open system call on the device, and using pread/pwrite for reading or writing to it. In Java it's not so straightforward since the language is not supposed to interface with the underlying system.

4.5.1 Traditional approach

One common way of accessing native code from Java is to wrap C-code by using JNI (Java native interface). GCJ also have a similar feature called CNI (compiled native interface). Since GCJ was used for the project it was decided to also try using CNI.

Early in the project a sample device driver was developed for accessing the FPGA. To try this device driver out a C user-program were written. The first approach was to simply making the C functions from the user-program accessible via JNI or CNI. Figure 10 illustrates this approach.

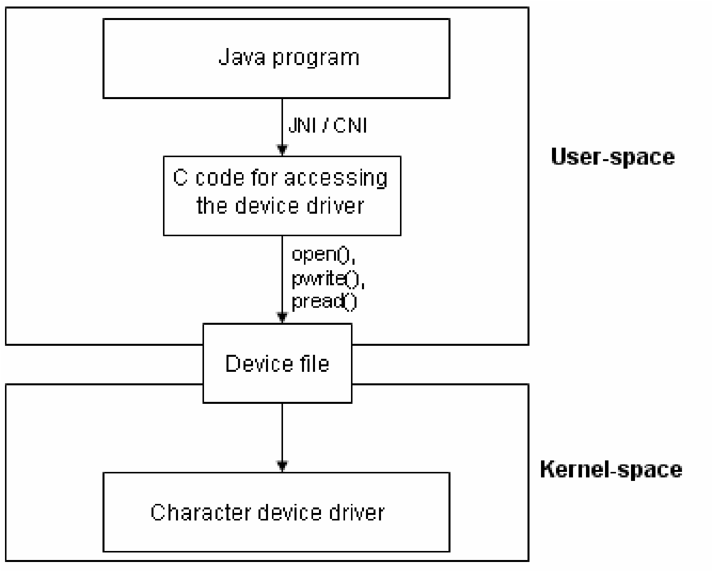


Figure 10: Traditional approach using JNI/CNI

As illustrated in the figure the device driver run in kernel-space and the Java-program run in user-space. The device driver is accessed through a device-file. Accessing a device driver in a Java-program is usually done by C-functions which are wrapped by JNI or CNI.

4.5.2 Problems with JNI and CNI

For various reasons CNI and JNI did not work together with GCJ. The functions to be accessed via JNI needed to be compiled to a shared library. However, the compiled GCJ only worked for building static Java programs. Therefore JNI could not be used. The CNI problem was different. Compilation of the C-function wrapped as CNI was possible but a problem appeared during runtime. When running the program a “memory error” occurred. Debugging the code showed that the C-function was entered correctly, however, at the open() system call the program crashed. The problem was not investigated further.

4.5.3 The solution: Accessing the character device as a file

After these problems it was considered if there were any alternatives to using JNI or CNI. The device driver was of the type character device which exposes itself as a file, which can be read and written. The idea was to simply use FileInputStream/FileOutStream for reading and writing to the virtual device file. It turned out this approach worked well and it was possible to skip an unnecessary layer of JNI. Figure 11 describes this simplified approach.

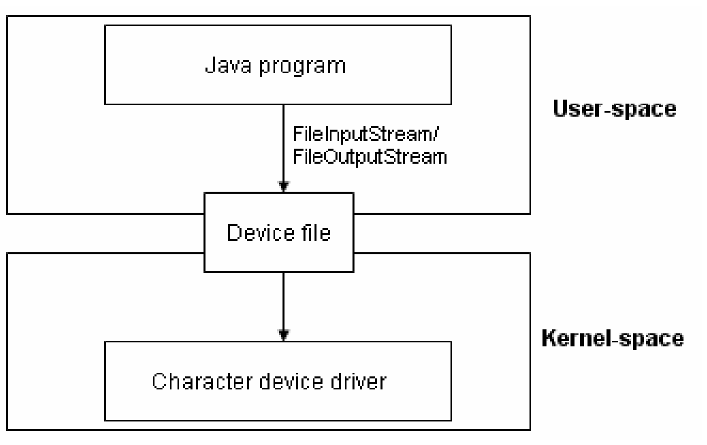


Figure 11: Simplified approach using FileInputStream/FileOutputStream

As shown in figure 11 the Java-program uses the classes FileInputStream and FileOutputStream for opening the device file as a normal file. The device driver is accessed by using the read-method of FileInputStream and write-method of FileOutputStream.

4.6 Cross-development Environment

Together with the FPGA-board comes a cross-compilation environment for C-programs. The environment consists of a version of GCC which can build native binaries for the PowerPC 405 CPU and the uCLibc-library. The environment includes coLinux which is a Linux distribution which can run inside the Windows operating system.

The reason for including coLinux in the cross-development environment is to make it easy for using already running Windows XP. The developer can install coLinux on top of the WindowsXP installation and quickly start using the GCC cross-compiler.

The cross-development had to be extended to support compilation of Java-programs. Eclipse was chosen as IDE. GCJ was used for cross-compiling Java programs to the FPGA-board. Similar to the cross-compilation version of GCC, GCJ also requires running in a Linux environment. GCJ makes use of GCC for producing native code.

coLinux has a feature which gives it it's own IP-address even though it's run inside the Windows operating system. This is useful for sharing files between the coLinux and Windows environment. Eclipse was installed in the Windows environment. A directory containing the development projects was shared in the Windows environment. This directory was mounted from the coLinux environment using SAMBA. By sharing the development projects files between the two environments it became easy to develop code in Eclipse and then compile it using the cross-compiler.

Figure 12 shows a diagram of the development environment.

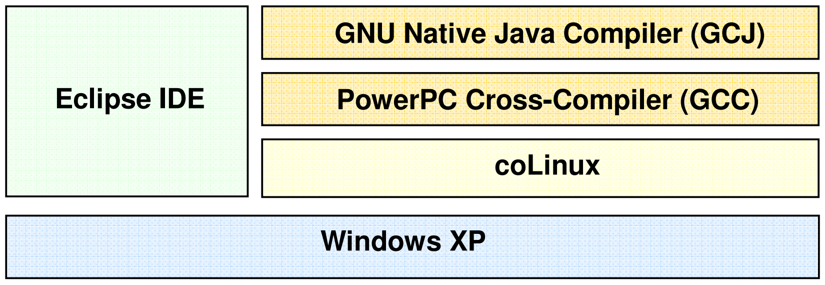


Figure 12: Java-based cross-development environment

As shown in the figure WindowsXP runs as the main operating system for the development PC. On top of WindowsXP runs the coLinux Linux distribution. The Eclipse IDE also runs in Windows. Inside the coLinux environment the GCC and GCJ compilers are installed.

4.7 Network Middleware

The platform was developed for use with two different network middleware-technologies: HORB and JavaRTM. This section describes how these were prepared for use on the FPGA-board.

4.7.1 HORB

HORB is a Java ORB which is developed by Dr. HIRANO Satoshi at AIST. To use of HORB in the platform it needed to be cross-compiled using GCJ. The latest public version of HORB, which is 2.0, was tried. Compilation was fine, but a problem occurred at runtime. The error had to do with a problem with ObjectInputStream in GCJ.

A workaround was found by using HORB 1.3 instead. This older version of HORB does not use ObjectInputStream. HORB 1.3 could be compiled and run successfully.

4.7.2 JavaRTM

JavaRTM is a component-based middleware for robotic applications. It uses CORBA for communication. At the time of writing it was under development. Therefore it had previously only been tested using Sun JDK. To be able to run JavaRTM on the FPGA-board required JavaRTM to be compiled with GCJ. Since RTM uses CORBA for communication a CORBA that could be compiled with GCJ was also needed.

Two different CORBA-implementations were tested to see if they could be compiled with GCJ: JacORB 2.0 and Orbacus 4.0.1. One issue was that the both JacORB and Orbacus included Swing GUI tools in their jar-packages. The GCJ used for compilation did not support Swing. Once the GUI classes was located and removed from the jar-files compilation was successful. During runtime another problem with JacORB appeared. The JacORB version that could compile with GCJ lacked the NamingServiceExt classes which are required by the JavaRTM implementation. However, Orbacus was found to compile and also worked well during runtime.

An issue also appeared when trying to compile JavaRTM with GCJ. Compilation of JavaRTM was fine; however, upon runtime an error caused all of the test-program to crash. The problem was traced down to the use of reflection. The crash could be avoided by converting the methods that used reflection to conventional programming. The problem was probably due to a bug how GCJ handles reflection.

5 Inverted Pendulum Construction

This chapter describes how the inverted pendulum was built. The work was divided into the following parts: electrical design, controller design and controller implementation.

5.1 The inverted Pendulum

The inverted pendulum is a common tool in control theory to test control algorithms. The goal of the inverted pendulum is to balance the pendulum arm in the upper position. Figure 13 shows a drawing of a rotary inverted pendulum (also known as the Furuta pendulum). The apparatus consists of four parts: pendulum arm, pendulum encoder, motor and motor arm.

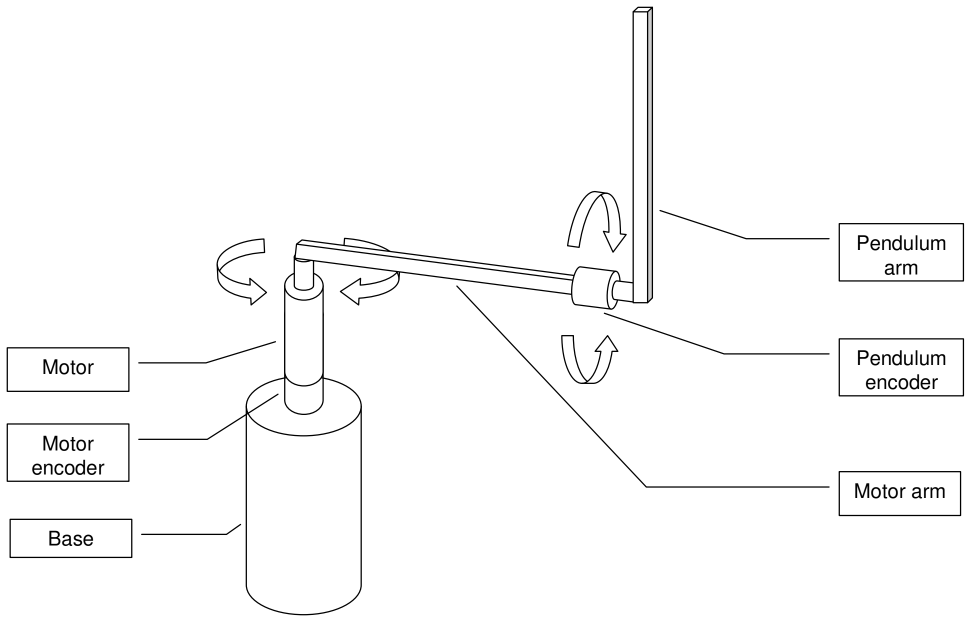


Figure 13: Inverted pendulum

As shown in figure 13 the pendulum arm is attached to the pendulum encoder and the motor arm is attached to the motor. The motor encoder is attached to the shaft of the motor. The encoders are used to sense the angles of the pendulum and motor arm. The pendulum arm is balanced into the top position with the help of a balance controller. The controller will operate by continuously reading the angle of the pendulum encoder, and then turning the motor to compensate for the displacement of the pendulum arm.

5.2 Electrical Design

The electrical design work consisted of choosing electrical parts for the inverted pendulum hardware and interfacing these with the FPGA-board.

5.2.1 Overview

Figure 14 shows a diagram which gives an overview of the parts which make up the inverted pendulum system and how they are connected.

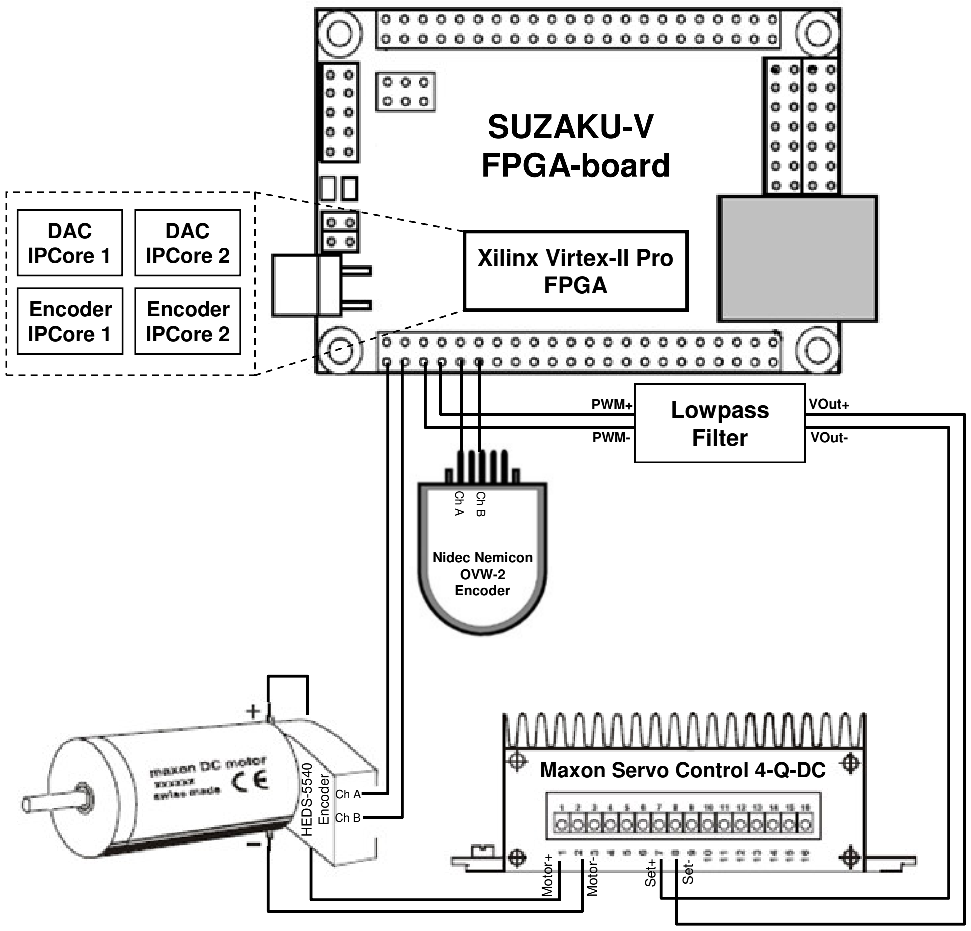


Figure 14: Electrical design block diagram

The electrical components of the inverted pendulum are: the pendulum encoder, the motor and its attached encoder and a servo amplifier. The servo amplifier is used for controlling the motor. As shown in the figure the electrical components are connected to the FPGA-board. In the FPGA-board circuits, known as IPcores, with specific functions to interface with the encoders and servo amplifier have been implemented. The function of the encoder IPCore is to read and decode the signals of an encoder. The function of the DAC IPCore is to control the motor by outputting a motor set voltage to the servo amplifier.

5.2.2 Encoders

This section describes the basic function of encoders, the encoders that are used in the system and how the encoders were interfaced with the FPGA-board.

5.2.2.1 Basics of encoders

An encoder is type of sensor which senses mechanical motion. The encoder translates a mechanical motion (such as speed, direction and shaft angle) into electrical signals. There are two types of encoders: absolute and incremental.

An absolute encoders report its position within 360 degrees. It has a fixed zero position and will output its current position as digital signals.

An incremental encoder output only the changes in movement as pulses. The output of an incremental encoder is called a quadrature-signal. A quadrate-signal use two channels which are typically called channel A and B. When there is a movement, a pulse is output in both of the channels but in a specific order. The order in which the pulses are sent in each channel is depending on the direction of the movement.

An important characteristic of an encoder is its resolution. The resolution is measured by CPR (cycles per revolution). Each cycle can provide 1, 2 or 4 counts (or pulses).

5.2.2.2 Choosing encoder

The most important critera for chosing encoder for the pendulum arm was the resolution. From studying other inverted pendulum projects it known a resolution of 1000-2000 P/R (pulses per revolution) would be needed. The encoder that was chosen for the pendulum arm was the OVW2 from Nidec-nemicon. The encoder has a resolution of 1800 P/R (pulses per revolution).

Figure 15: Rotary encoder OVW2

5.2.2.3 Encoder interface

The encoders were interfaced to the FPGA-board by using an IPCore developed by Dr. OHKAWA Takeshi. Figure 16 shows how the encoders are interfaced with the FPGA-board. The diagram shows only one of the encoders; however, both of the encoders were interfaced as described in the diagram.

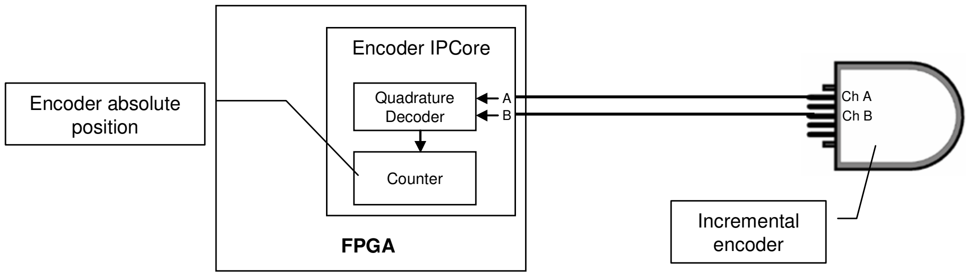


Figure 16: Encoder interface

The signals from the encoder are connected to the FPGA-board as shown in the figure. As mentioned, the encoder signal (quadrature signal) consists of two channels. The Encoder IPCore has the purpose of converting the quadrature signal of the encoder to an absolute position. This is done in two steps. The first step decodes the quadrature signal. The second step updates a counter of the movement.

5.2.3 Motor

The work with the motor involved choosing a motor, finding a suitable motor control method and interfacing the motor to the FPGA-board.

5.2.3.1 Choosing motor

The motor used for this project was a Maxon RE 25 brushed DC motor. The motor had an already attached encoder called HEDS-5540. The encoder has a resolution of 400 P/R.

5.2.3.2 Choosing motor control method

There are many different methods for controlling a DC Motor. In the beginning of the project it was not known which motor control method is suitable for the inverted pendulum.

Some common motor control methods are: voltage control, speed control, torque control and position control. The methods are briefly described below:

- Voltage control is the simplest method of controlling a DC motor. The motor is controlled only by voltage. Higher voltage will make the motor turn faster, and lower voltage slower. However, it will not be possible to tell at exactly which speed the motor is turning.

- Speed control can be achieved by attaching a rotary encoder to the shaft of the motor. By using the encoder output as feedback a control loop can set the motor to the desired speed.

- By position control the motor can be stopped in a specific position. Position control also requires an attached encoder for getting the feedback of the motor movement.

- Torque control is used to control the torque of the motor. Torque control can be achieved by controlling the motor by current instead of voltage.

Experiments were done with all of the above methods. Finally the torque control method was used for controlling the motor. The reason is that the motor arm will need to accelerate faster than the pendulum arm is falling. By controlling the motor by torque it's possible to control its acceleration, since the torque of a motor is proportional to its acceleration.

The servo amplifier Maxon Servo Control 4-Q-DC LSC 30/2 was used for the motor control. The servo amplifier can be configured for all of the motor control methods described above (except position control). The servo-amplifier has a set voltage input. The set voltage is used to specify for example the desired speed of the motor if the servo amplifier is configured in the speed mode.

5.2.3.3 Motor interface

The servo amplifier is interfaced to the FPGA-board by the use of DAC IPCore, which is also developed by Dr. OHKAWA Takeshi. The details of the interface are described below.

As mentioned the motor itself is controlled by a servo amplifier. To control the servo amplifier the FPGA-board need to output a set voltage between -3.3 and +3.3V. Analog output is achieved by using a DAC (digital-to-analog) IPCore which output a PWM (pulse-width-modulated) signal. A PWM signal is a digital signal which can have a specific ratio of zeros or ones. For example, at 100% it includes only ones, and 0% only zeros. The PWM signal is then connected through a lowpass-filter built by an RC-circuit, which results in an analog voltage. The motor interface diagram is shown in figure 17.

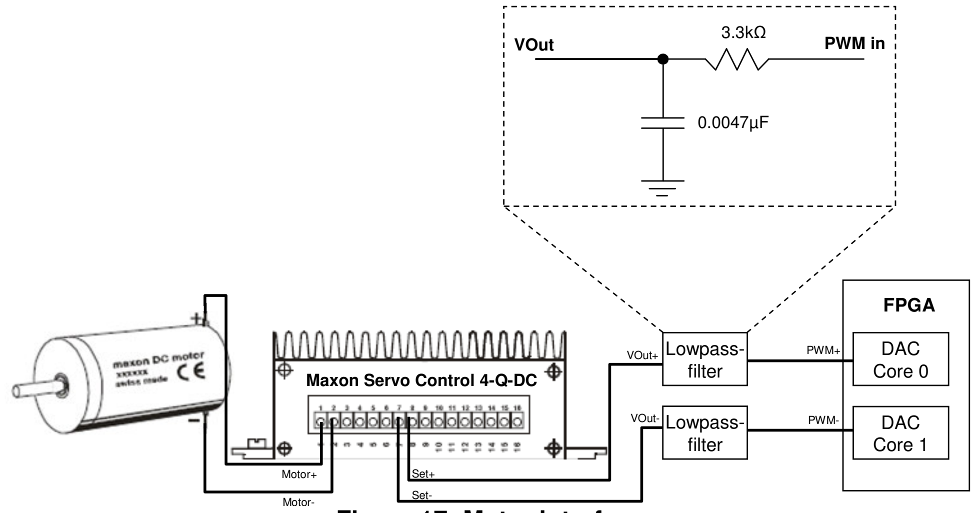


Figure 17: Motor interface

In the FPGA there are two DAC IPCores. The reason is that the analog voltage need to be between -3.3 and +3.3V. A single DAC can only achieve a positive voltage. Therefore two DAC are used and the potential becomes the voltage output to the servo amplifier.

5.3 Controller Design

This section describes the design of the controller for balancing the inverted pendulum. The controller was designed for two separate functions: balancing the pendulum and swinging up the pendulum.

5.3.1 Balance Controller

The balance controlled has the most important function in the inverted pendulum system. It balances the pendulum in straight position.

5.3.1.1 Background

The goal of the balance controller is to keep the pendulum arm in the upper position as shown in figure 18. If there was no control at all the pendulum arm would quickly fall to the bottom position. The controller is continuously reading the angle of the pendulum arm from its encoder. If the controller detects that the pendulum arm is moving it will turn the motor arm to compensate, and thus keep the pendulum arm straight.

The figures below give an example scenario of how the controller will work.

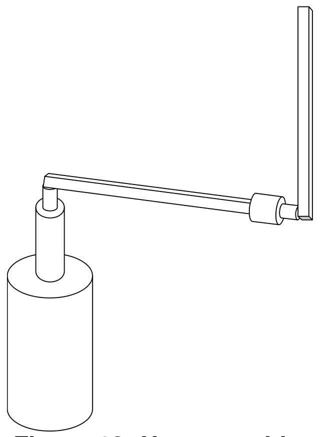


Figure 18: Upper position

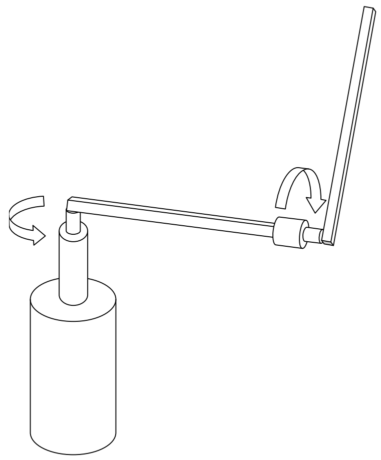


Figure 19: Falling

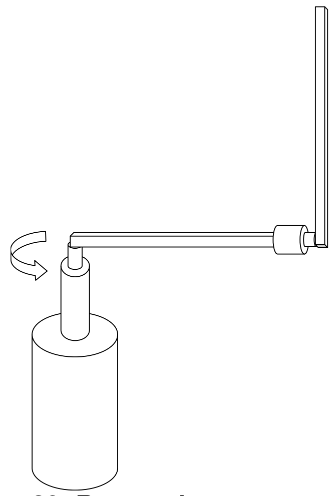


Figure 20: Returned to upper position

In figure 18 the pendulum is in the upper position and no motor movement is necessary. Figure 19 shows how the pendulum arm starts falling. The controller will turn the motor to move the motor arm in the same direction as the pendulum arm is falling (figure 19). Figure 20 shows that after moving the motor arm the pendulum arm is back in the upper position.

The balance controller operates in a continuous feedback loop which is illustrated in figure 21.

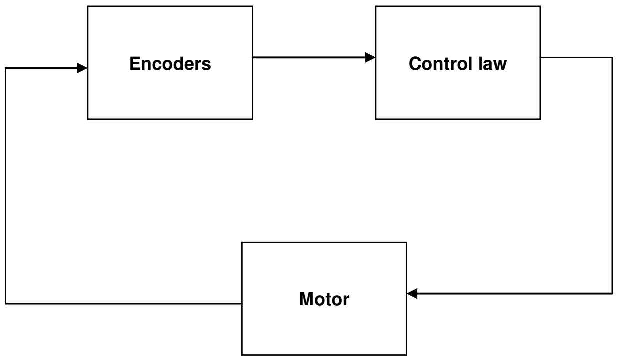


Figure 21: Control loop

As shown in the figure the encoders are the input for the balance controller. The balance controller implements a control law which transforms the encoder input to motor movement. The motor movement will affect the new encoder measurements, and so on.

5.3.1.2 Approaches

Two different approaches were tried for balancing the pendulum.

- PID regulator
- State feedback

The parameters of the control algorithm will be found by experiments.

5.3.1.3 PID regulator

The PID regulator is easy to understand and implement. For this reason it was believed to be a good first approach for trying to balance the pendulum.

PID operates on single input. The goal of the PID regulator is to keep the input at a certain value. The difference between the actual input and the desired value is called the error. The output of the PID regulator tries to compensate for the error and bring the input closer to the desired value.

This error is transformed to an output by the constants: P, I and D. The proportional (P) constant is multiplied with the error and thus gives a proportional compensation to the size of the error. The integral (I) constant is multiplied with the integral of the error. The term is used to take into account history of previous errors. The derivate (D) constant is multiplied with the derivate of the error. This term is used to take into account the speed of change of the error.

The PID regulator was implemented and tried for the inverted pendulum. The desired value was specified as the top position of the pendulum arm. Many experiments were done to find the values for the P, I and D parameters. The parameters were tuned by setting all of them to zero, and then systematically changing them until it seemed the regulator could almost balance the pendulum. Often the regulator would be too weak with the result of not catching the pendulum quickly enough. However, by using slightly higher values for the parameters the controller became too strong with the result of overshooting. Once such maximum and minimum parameters were found the parameters was increased and decreased to find new parameters which were not too weak or too strong.

The best parameters that were found could sometimes balance the pendulum for a very small change in the upper position. However, most of the times it still failed. The PID regulator was determined not to be a suitable regulator for the inverted pendulum problem. Later it was learnt that an important reason why the PID cannot balance a pendulum is that PID only take into account one input variable. The inverted pendulum has multiple inputs which all need to be taken into account.

5.3.1.4 State Feedback

For control problems with multiple input parameters a state feedback regulator is needed. A state feedback regulator can be described as a mathematical model of the physical system. The following variables are introduced for describing the pendulum arms:

x_{ref}	desired position of the pendulum arm	θ_{ref}	desired position of the motor arm
x	position of the pendulum arm	θ	position of the motor arm
$\dot{x}$	speed of the pendulum arm	$\dot{\theta}$	speed of the motor arm

The state feedback control law is:

$$u = -LX = -\begin{bmatrix} k1 & k2 & k3 & k4 \end{bmatrix} \begin{bmatrix} x_{ref} - x \\ \dot{x}_{ref} - \dot{x} \\ \theta_{ref} - \theta \\ \dot{\theta}_{ref} - \dot{\theta} \end{bmatrix}$$

The variable u is the motor output. The constants k1, k2, k3 and k4 is used to give weights to each of the input variables. The controller can be understood intuitionally by considering the input variables that are needed. The input variables that are needed for the control law are: pendulum arm position, pendulum arm speed, motor arm position and motor arm speed. Why are these input parameters important?

- **Position of pendulum arm.** The position of the pendulum arm should be straight. The controller will need to output a motor movement with a strength that is proportional to the size of the displacement of the pendulum arm from the top position.

- **Pendulum arm speed.** If the speed of pendulum arm is fast the motor arm also need to turn fast to compensate. If the speed of the pendulum arm is slow the motor arm need to move slower to not cause an overshoot.

- **Motor arm speed.** The speed of the motor arm also needs to be included in the control law. This is important for two reasons. One reason is this input variable contains feedback of the actual motor speed. The other reason is that without this variable the motor would continue turning since there would be nothing in the control law which care about the motor speed.

- **Position of motor arm.** The position of the motor arm is useful for a more advanced control of the inverted pendulum. By including this variable into the control law it is possible let the motor arm keep a certain position while balancing the pendulum.

Figure 22 shows a diagram of the balance controller.

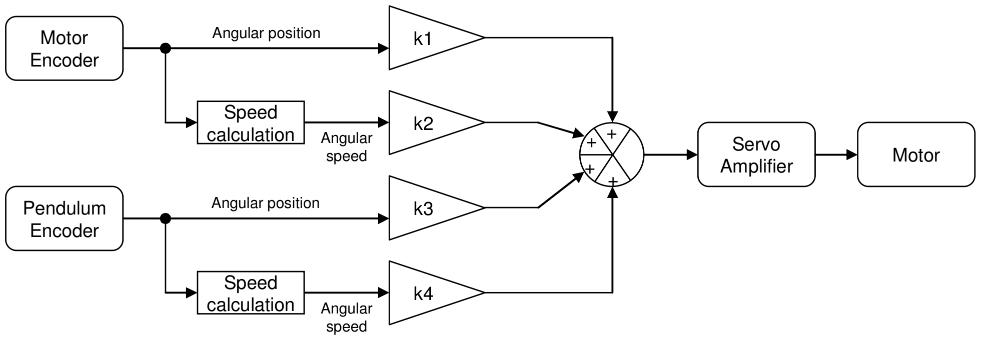


Figure 22: Controller diagram

As shown in figure 22 the constants which are multiplied with each input parameter are called k1, k2, k3, and k4. The result of each input variable is added. The output from the controller is input to a servo amplifier which controls the motor.

5.3.1.5 Understanding the constants

The constants of the control law will depend on the mechanical characteristics of the pendulum (such as length of motor arm, length of pendulum arm and their weights). Another parameter which greatly influences the constants is the sample time of the control loop.

With an understanding of how the controller operates it is possible to manually find the constants. The following is a description of how to find the sign of the constants.

- **Pendulum position (k3).** If the pendulum arm falls in some direction, the motor arm should move in the same direction to compensate. Therefore, the sign of the constant k3 should be positive.

- **Pendulum speed (k4).** The speed of the pendulum arm should be proportional to the speed of the motor arm. The sign of this constant should also positive.

- **Motor arm speed (k2).** The purpose of the motor arm speed variable is to get feedback from the actual motor arm movement. If the motor arm is always turning the inverted pendulum controller will not appear stable. Therefore the goal of motor arm speed should be zero. How can this be achieved? Consider decreasing the speed of the motor arm if the motor arm already has some speed by setting a negative constant. This will give a bad effect since if the motor arm is already turning, there is probably a good reason for it – it is trying to catch up with the falling pendulum. Therefore, the sign of this constant should be positive so that the motor arm can catch up with the pendulum arm more quickly.

- **Motor arm position (k1).** The purpose of the motor arm position constant is to let the motor arm stay in a certain position. The motor arm should slowly return to its initial position. This constant should be positive so that while the motor arm is turning to compensate for the falling pendulum arm it will always cause a small overshoot. If the motor arm is far away from its initial position the overshoot will become larger. The effect of the overshoot will be that the pendulum arm falls back in the direction of the motor arms initial position.

5.3.1.6 Procedure for tuning the constants

During the project a procedure for tuning the constants of the controller has been found. The procedure can be used to find the constants quite easily.

Initially all constants should be set to zero.

1. Start with the pendulum position constant (k3). Put the pendulum in the upper position and slowly increase the value of the constant. When giving the pendulum arm small pushes the motor arm should compensate without causing an overshoot.

2. Continue with the pendulum speed constant (k4). Slowly increase the constant. When the constant is tuned correctly it will be able to compensate for the speed of the falling pendulum. The pendulum control will appear more stable. However, the motor arm will continue turning without any stop.

3. At this time, start increasing the motor arm speed constant (k2). After increasing the constant to a certain value the pendulum arm will stop to continuously turning. Increasing the constant too much will cause overshoot but will also make the pendulum controller strong.

4. The motor arm position (k1) is the most difficult constant to tune. The reason is that it should be very small not to affect the other parameters in a negative way. Try with very small values compared to the other constants until it seems as the motor arm starts to return to its initial position.

Repeat step 1-4 until the balance controller is completely stable.

It is worth noting that the interval of the constants is quite large. There is much room for tuning before finding an optimal balance controller.

5.3.2 Swing-up Controller

The purpose of the swing-up controller is to swing-up the pendulum from the bottom position to the top position. Once the pendulum has reached the top the balance controller will catch the pendulum and resume operation.

For the balance controller to be able to catch the pendulum the speed of the pendulum arm can not be too high when reaching the top position. A common strategy for swinging up the pendulum is known as “swing-up by energy control”. Åström and Furuta describe this strategy in detail in [14]. The basic idea of this strategy is to control the pendulum by its energy. Energy is “pumped” into the system until the energy which corresponds to the top position of the pendulum is reached.

Based on this method Svensson in [15] describe how an inverted pendulum can be brought to its upper position by the following equation:

$$u = ksign(\dot{\theta}\cos(\pi - \theta)$$

In the equation θ is the position of the pendulum arm, $\dot{\theta}$ is the speed of the pendulum arm and k is a constant. The sign function is defined as 1 for a positive argument and -1 for a negative argument.

5.4 Local Controller Implementation

The control algorithm was first implemented locally in the FPGA-board. This was done to learn how to control the inverted pendulum without the additional complexity added by network communication. The local controller was implemented both in C and Java.

The diagram in figure 23 shows an outline of how the program operates.

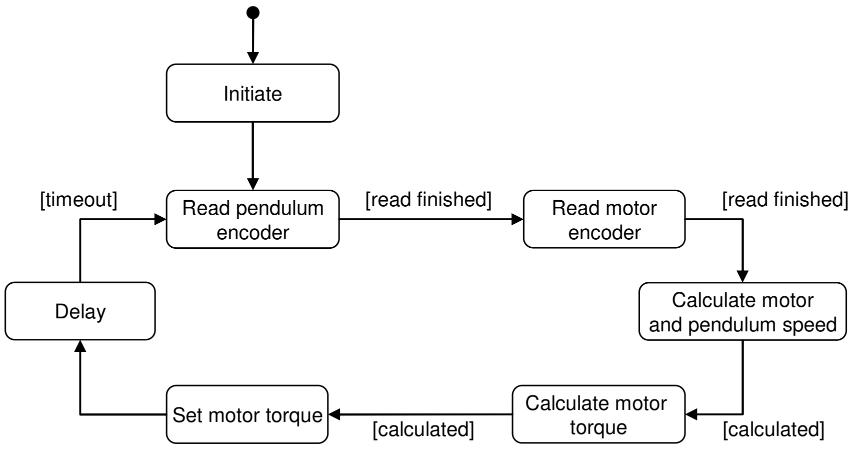


Figure 23: Local controller state diagram

As shown in the figure the program first read the positions of the encoders. It then calculates the speed of the pendulum and motor arm. By using these values the program calculates the torque that should be applied to the motor to balance the pendulum. The program then outputs the calculated torque to the motor. To give the control loop a specific sample time the program then enter a delay function.

The behavior of the program is explained in detail in the controller algorithm which is shown in algorithm 1.

```
while(true)
{
    pendulum_pos = read_pendulum_encoder();
    pendulum_radian_pos = convert_to_radians(pendulum_pos);
    motor_pos = read_motor_encoder();
    current_time = get_time();

    pendulum_changes[next] = pendulum_pos - last_pendulum_pos;
    motor_changes[next] = motor_pos - last_motor_pos;
    sample_times[next] = current_time - last_time;
    next = (next + 1) % RING_SIZE;

    motor_speed = calculate_speed(motor_changes, sample_times);
    pendulum_speed = calculate_speed(pendulum_changes, sample_times);
    radianSpeed = calculateRadianSpeed(pendulumChanges, sampleTimes);

    check_bounds(pendulum_pos, motor_pos);

    if( swingup mode )
        u = swingup_constant * sign(speed * cos(PI – pendulum_radian_pos);
    else
        u = (k1 * motor_pos + k2 * motor_speed + k3 * pend_pos + k4 * pend_speed);
    set_motor_torque(u);

    last_pendulum_pos = pendulum_pos;
    last_motor_pos = motor_pos;
22.         last_time = current_time;
23.         if(not swing-up mode)
24.             delay(SAMPLE_TIME);
25.         else
26.          {
27.              if(reached top position)
28.                  swing-up mode = false;
29.          }
30.     }
```
Algoritm 1 – Local controller algorithm

For the balance controller it is helpful if it can operate with a constant sample time. It will give a more stable control. It will also make the speed calculation easier. However, the implementation is done in a Linux-variant without real-time support so the algorithm will need to take into account that the sample time between each control loop iteration will vary. Therefore the algorithm will need to measure the actual sample time in each control loop. In the algorithm this is done by reading the current time (in a value which should have microsecond precision) which is shown at line 6. The value is then stored in the last_time variable at line 22.

Line 7-9 stores the pendulum arm movement, motor arm movement and sample time during the current control loop iteration in a ring buffer. The values inside the ring buffer are then used in the speed calculation at line 11 and 12. The speed calculation is shown in algorithm 2.

```
1.   change = 0;
2.   sample_time = 0;
3.   for(i=0; i < RING_SIZE; i++)
4.   {
5.        change += changes[i];
6.        sample_time += sample_times[i];
7.   }
8.   speed = total_changes / total_sample_time;
```
Algorithm 2 – Speed calculation

As shown at line 5 the encoder changes stored in the ring buffer added. The sample times stored in the ring buffer are also added. The speed is then calculated by dividing the changes with the sample times.

Line 18 in algorithm 1 shows how the motor feedback is calculated when the algorithm is working in balance mode. The constants k1, k2, k3 and k4 are multiplied with the positions and speeds of the pendulum and motor arm. The feedback value is then set as torque to the motor. The motor will have this torque until the next iteration of the control loop.

Line 24 is a call to a delay route which will cause the program to sleep. The purpose of the delay is to make the control loop operate with a constant sample time. The sample time is very important for
the stability of the control algorithm. The goal is to have a sample time which is both constant and as low as possible. A high sample time will make the response of the controller appear slow. However, if the sample time is too low the motor will not have time to adjust to the new torque. From studying other inverted pendulum projects it was known a sample time of about 1 millisecond would be suitable.

5.4.1 C-based implementation

The program was first implemented in POSIX/C. The get_time() function could be written using the gettimeofday() function. This function provides the time in microsecond resolution which is accurate enough for the time measurements inside the control loop.

The delay function will also require microsecond precision since the control loop should be about 1 millisecond. A common function for sleeping is sleep(), however, this function has only a precision of seconds. Three other methods were examined: usleep, nanosleep and a gettimeofday-loop.

A function similar to sleep is usleep. According to the the man page usleep can sleep with microsecond precision. However, testing this function showed that the smallest sleeping time becomes 20 milliseconds. This is due to process switching time in Linux. Nanosleep usually functions the same as usleep (the smallest sleeping time becomes 20 milliseconds). However, in kernel 2.4, it can be made to operate differently if set_scheduler() is used. This way nanosleep can sleep with microsecond precision. Another method for delay is to make a loop that continuously calls gettimeofday() until a specified time has elapsed.

Evaluation of delay-methods
As mentioned, one requirement of the delay function is that it can wake up within one millisecond. Another requirement is that it wakes up at the exact time that was specified. It should not wake up too late, and not too early.

The following delay-methods were evaluated: nanosleep, gettimeofday in loop and gettimeofday-loop wrapped in a function. The methods were evaluated in the following way. Each method was instructed to sleep exactly 1 millisecond. A timer was started before the delay call and stopped after. The time that elapsed after one millisecond was printed. The test was repeated 60 times for each sleep-method. Figure 24 shows the result.

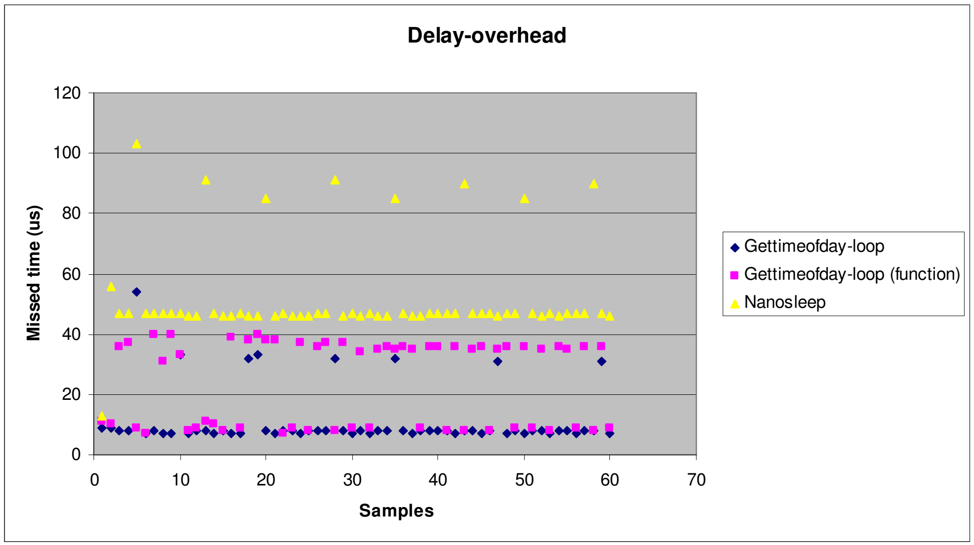


Figure 24: Evaluation of sleep-methods

As shown in the graph nanosleep produces the worst result. The average missed time for nanosleep is about 50 microseconds. However, at times it misses the specified time with 90 microseconds. Using the gettimeofday-loop wrapped in a function is better. The worst overhead time of this method is below the average overhead of the nanosleep-function. The gettimeofday-loop (without wrapping it in a function) has the smallest delay-overhead.

The method was that chosen was the gettimeofday-loop wrapped in a function. The slight overhead from using a function was acceptable since it increases the code readability.

5.4.2 Java-based implementation

The Java implementation (without remote balancing) was almost identical to the C implementation. The things that were different have to do with the fact that Java cannot easily access low-level resources of the operating system. The device driver was accessed using the solution which was described in 4.5. Java does not provide a way to access a high-resolution clock. Java only provides the System.getTime() which has a millisecond precision. Therefore CNI was used to use the gettimeofday() function which was also used in the C-program. The sleep-method of Java can accept an input of millisecond precision. However, the actual time it wakes up was similar to the usleep-function in the C-program which was about 20 milliseconds. Therefore the gettimeofday-loop in the C-program was also used with CNI.

In the platform chapter it was mentioned that CNI could not be used to accessing the device driver (using POSIX open and pwrite functions). Nonetheless, CNI worked fine for wrapping the gettimeofday()-functions.

6 Remote Balancing Implementation

The inverted pendulum was implemented for remote balancing in two different network middleware-technologies: HORB and Java-RTM. This chapter describes the implementation of each solution.

6.1 HORB-based Implementation

HORB is a Java ORB developed by the Network Middleware Group at AIST (http://horb.aist.go.jp). Some of the advantages of HORB are that it's light-weight and fast ORB which makes it suitable for implementing the remote balanced inverted pendulum.

6.1.1 Design

Figure 25 shows a class diagram which describes how the HORB-implementation was designed.

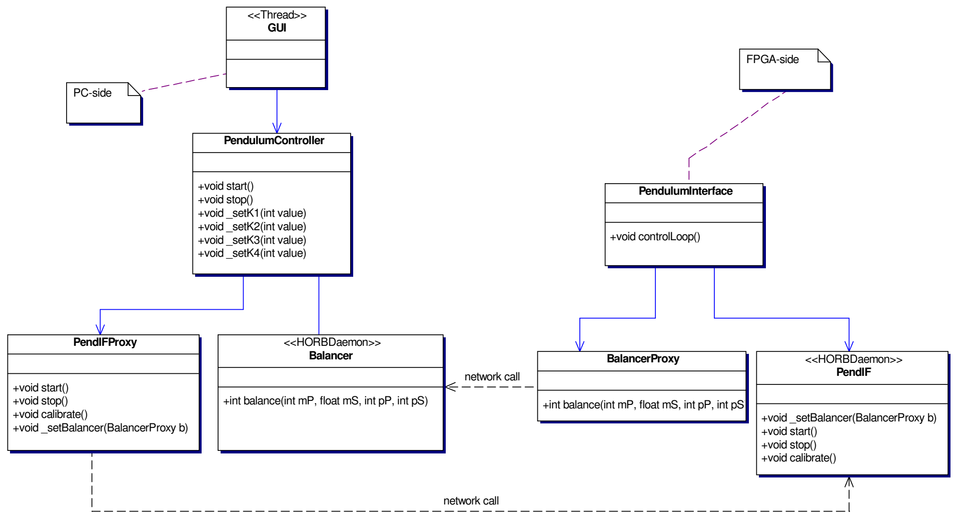


Figure 25: HORB-based solution

The left side of the diagram shows the classes which make up the pendulum interface application which is run on the PC. The classes in the right side of the diagram make up the pendulum interface application which is run on the FPGA-board.

The pendulum controller has three classes: PendulumController, PendIFProxy and Balancer. PendulumController is the controller class which is used by the GUI. PendIFProxy is used by the PendulumController to send the start and stop command to the pendulum interface. The Balancer class is a remote object which is used by the pendulum interface to balance the pendulum.

The pendulum interface also has three classes: PendulumInterface, BalancerProxy and PendIF. PendulumInterface is running the control loop. The PendulumInterface uses the BalancerProxy class to send remote balance requests. The PendIF is a remote object which is used by the pendulum controller to send commands to the pendulum interface.

The solution is designed in a way so that the pendulum interface will not need any knowledge of the pendulum controller. This is achieved by using a feature in HORB which allows transferring a proxy as a remote object. The pendulum controller will send its Balance object as a proxy to the pendulum controller. This is illustrated in figure 26.

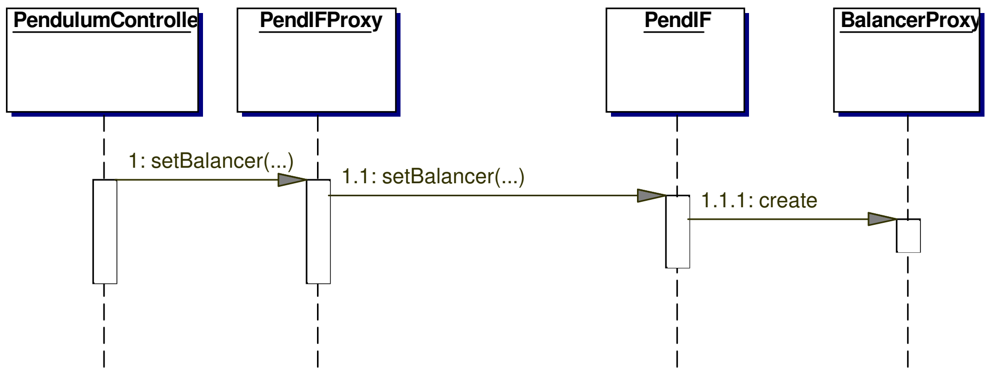


Figure 26: Transfer of the balancer proxy object

At the startup of the pendulum controller will send its Balancer object to the pendulum interface via the PendIFProxy. The pendulum interface can then use the balancer proxy for sending remote balancing requests. Without this approach the pendulum interface would be required to know the IP address of the pendulum controller for looking up the Balancer object.

Figure 27 shows how the objects in the solution interact while running.

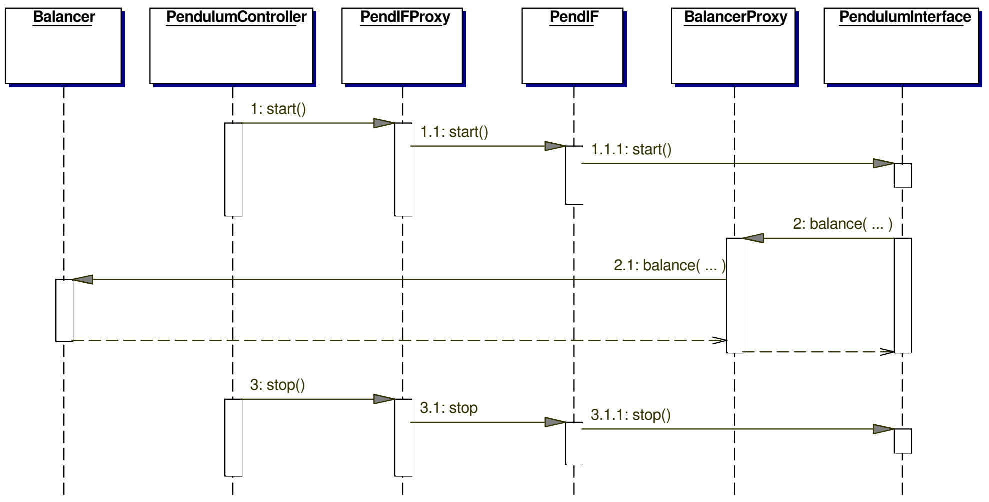


Figure 27: Sequence diagram of the HORB-based solution

At first the PendulumController send a start command. The PendulumController call the PendIFProxy which make a network call to the PendIF remote object. The PendIF object tell the PendulumInterface to start the control loop. In 2 the control loop has started. The PendulumInterface object uses the BalancerProxy to send remote balance commands. The Balancer object does the balance calculation and returns the result. The control loop will run until a stop command is sent as shown in 3. The stop command is sent in the same way as the start command.

6.2 JavaRTM-based Implementation

RTM (Robot Technology Middleware) is a specification for distributed component-based robotic applications. In RTM individual parts of a robot system are developed as separate components. Each of the components can be connected to build a system. The components use in and out ports for communication. The components can also use a service port for receiving commands. RTM uses CORBA as the underlying technology for communication between components.

RT components have a specified life-cycle. Methods in the RT component are invoked by the runtime during each step of the RT component's life. The phases of the life-cycle of the RT component includes: born, started, stopped, and running.

OpenRTM is an implementation of the RTM specification which is written in C++. JavaRTM is a version of RTM developed in Java. JavaRTM is based on the OpenRTM source code.

6.2.1 Design

The inverted pendulum system was designed to work in the component-model provided by JavaRTM. Figure 28 shows a diagram which describes the design.

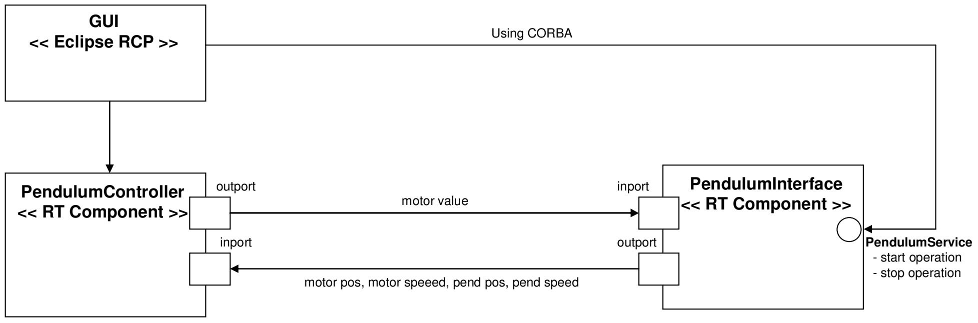


Figure 28: JavaRTM-based solution

The pendulum controller and the pendulum interface are developed as RT components as shown in the figure. The pendulum controller has an in port which is used to receive the pendulum state from the pendulum interface. The pendulum controller returns the motor value through its out port which is connected to the inport of the pendulum interface.

A GUI was developed using Eclipse RCP (Rich Client Platform). The GUI references the PendulumController component for setting the balancing constants. The PendulumInterface component has a service port with two operations: start and stop. The GUI can access the service port by looking up the RT component through the CORBA naming service. The actual calls to the operations of the service port are done via CORBA invocations.

While the RT component is running the JavaRTM runtime continuously call active_do method of the component. In this method the logic of the components has been implemented. Within the active_do method the components uses the read and write operation on their in and out port to send and receive data.

7 Evaluation of HORB and JavaRTM

This chapter describes how HORB and JavaRTM were evaluated to find out if they were suitable as network middleware for the remote balanced inverted pendulum.

7.1 Evaluation Criteria

The inverted pendulum system requires that the network middleware can provide a low response time. The reason is that the response time effectively becomes the sample time of the control loop. If the average response time is high, balancing the pendulum becomes difficult, and the balance controller will appear poor. For this reason the average response time of HORB and JavaRTM have been measured.

While a low average response time is important for overall stability of the balance controller another important aspect is the real-time characteristics of the response time. If the worst case response time is above some value the balancing will fail and the pendulum arm will drop. In that case the hard real-time deadline of the system has been missed. To illustrate this situation the real-time characteristics of the response time were also measured.

7.2 Measurement of the Average Response Time

Figure 29 shows the response time for three different versions of JavaRTM and HORB.

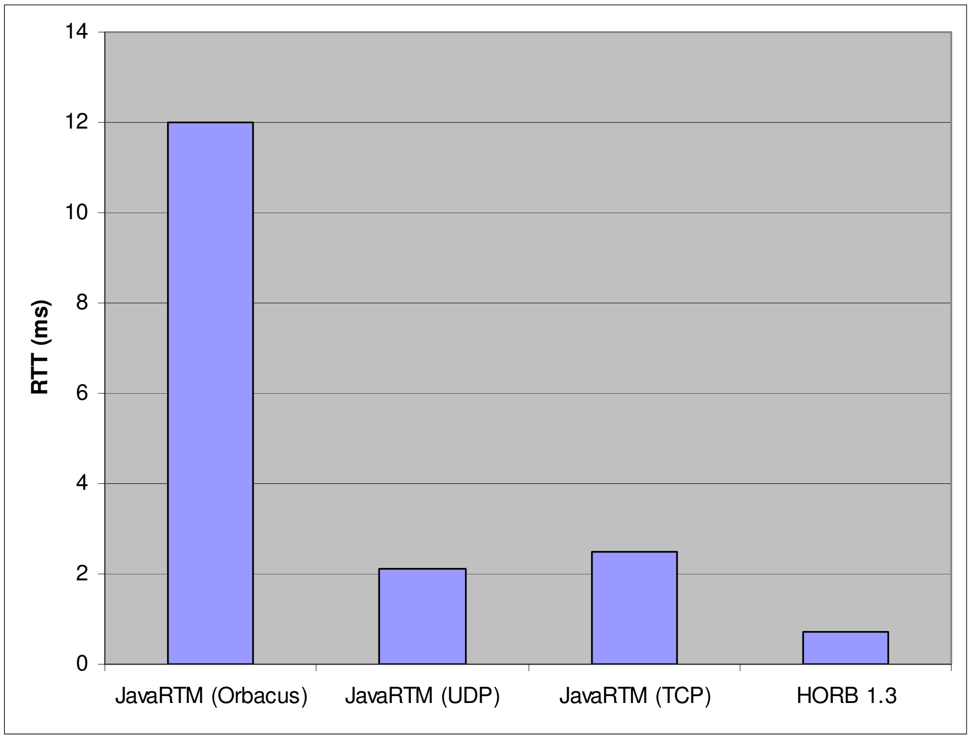


Figure 29: Average response-time

The average response time of HORB was 1.2 ms which is suitable for remote balancing the inverted pendulum. It was also possible to balance the pendulum using the UDP and TCP versions of JavaRTM, which have a response time of 2.1 and 2.6 ms respectively. However, in the CORBA version of JavaRTM, which used Orbacus, the inverted pendulum could not balance more than a few seconds.

7.3 Real-time characteristics of the response time

To show the real-time characteristics of the response time the response time was measured over a period of time and each time was plotted in a graph. By showing the response time this way it is possible to get an idea of how often the worst-case response time occurs.

7.3.1 HORB 1.3

The measurement was done with HORB 1.3. Figure 30 shows the result.

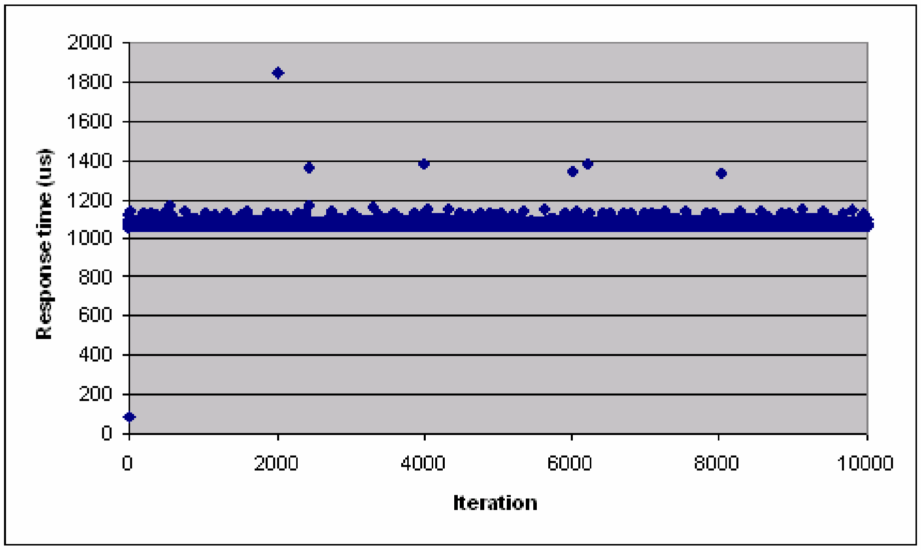


Figure 30: Real-time characteristics of HORB 1.3

As shown in the graph the average response time of HORB is about 1.2 ms. In very few cases (in 6 out of 10000 control loops) the response time becomes 200 us higher. Such small noises did not affect the inverted pendulum at all.

7.3.2 JavaRTM (Orbacus version)

The same measurement was done using the Orbacus version of JavaRTM. Figure 31 shows the result.

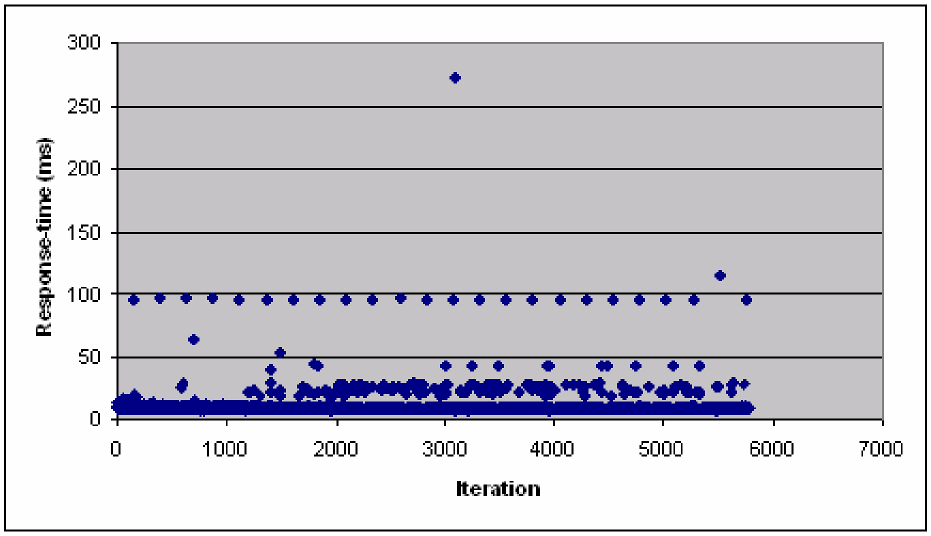


Figure 31: Real-time characteristics of JavaRTM (Orbacus version)}:

As mentioned above the pendulum failed balancing after a few seconds using this version. The average response time of the JavaRTM Orbacus version was 12 ms. However, as shown in the graph there are many noises at values such as 100 ms. These noises made the pendulum fail balancing The noises were probably caused by the many internal threads of the Orbacus CORBA implementation.

7.3.3 Summary of the results

HORB 1.3 performed very well both in average response time and real-time characteristics of the response-time. It was possible to use JavaRTM for remote balancing using the UDP and TCP versions. However, the Orbacus version had a high average response time (12 ms compared to 1.2 ms for HORB 1.3). The main difference between HORB and the Orbacus version of JavaRTM is in the real-time characteristics of the response time. HORB has a very consistent response time but the Orbacus version of JavaRTM had many noises at as high values as 100 ms.

Some reasons for these different results are discussed. HORB 1.3 is a very well-tested ORB which has been around for many years. On the contrary, the JavaRTM version which was used in this evaluation is still in the development-phase and much optimization remains. HORB 1.3 has a simple architecture which uses a thread-pool for handling requests. JavaRTM has more complex architecture. There are many internal threads and much thread-synchronization is needed to handle a single request.

8 Conclusions

The inverted pendulum system was developed and worked successfully both for local and remote balancing. The work included electrical and mechanical design of the inverted pendulum hardware, design and implementation of a control algorithm for balancing the pendulum and implementation of remote balancing using two network middleware: HORB and JavaRTM. Furthermore, a Java-based software platform was developed for a Xilinx Virtex-II Pro based FPGA-board. The platform included a solution for how to run Java within the embedded CPU of the FPGA, a solution for how to access hardware from Java and device drivers for controlling motor and reading encoders.

For building the inverted pendulum the FPGA-board was found to be very useful. The FPGA made it possible to quickly try new electronic designs. Since the FPGA-board had an embedded CPU it was also possible to easily take advantage of the power of the FPGA from software. An FPGA-board with an embedded CPU is very productive for hardware–software co-design projects.

An embedded system poses some challenges to the developer due to restrictions in memory size and processing capacity. This fact made it difficult to make a standard JVM run on the FPGA-board which only has 8 MB of permanent storage – not enough for most JVMs. However, it was found that using GCJ (GNU Compiler for Java) is a good way of running Java in an embedded system. The reason is that GCJ compiles Java directly to native code and thus a JVM is not needed.

As mentioned, for controlling the inverted pendulum a balance controller capable of swing-up was implemented successfully both in C and Java. It was found that a PID regulator was not suitable for balancing the pendulum since the PID regulator is only taking into account a single input parameter. The inverted pendulum has multiple parameters that need to be taken into account. A state feedback regulator, which takes multiple parameters into account, worked well for balancing and its constants could easily be found by tuning.

The inverted pendulum system required a 1-7 ms servo-loop for stable balancing. It was easier to tune the balancing algorithm for a servo-loop with a lower time.

Java performed well enough to balance the pendulum both locally and remotely. It was suspected that garbage collection of Java would affect the controller, but no such problem appeared in the local balancing. The local balancing program operated in a 1 ms servo-loop.

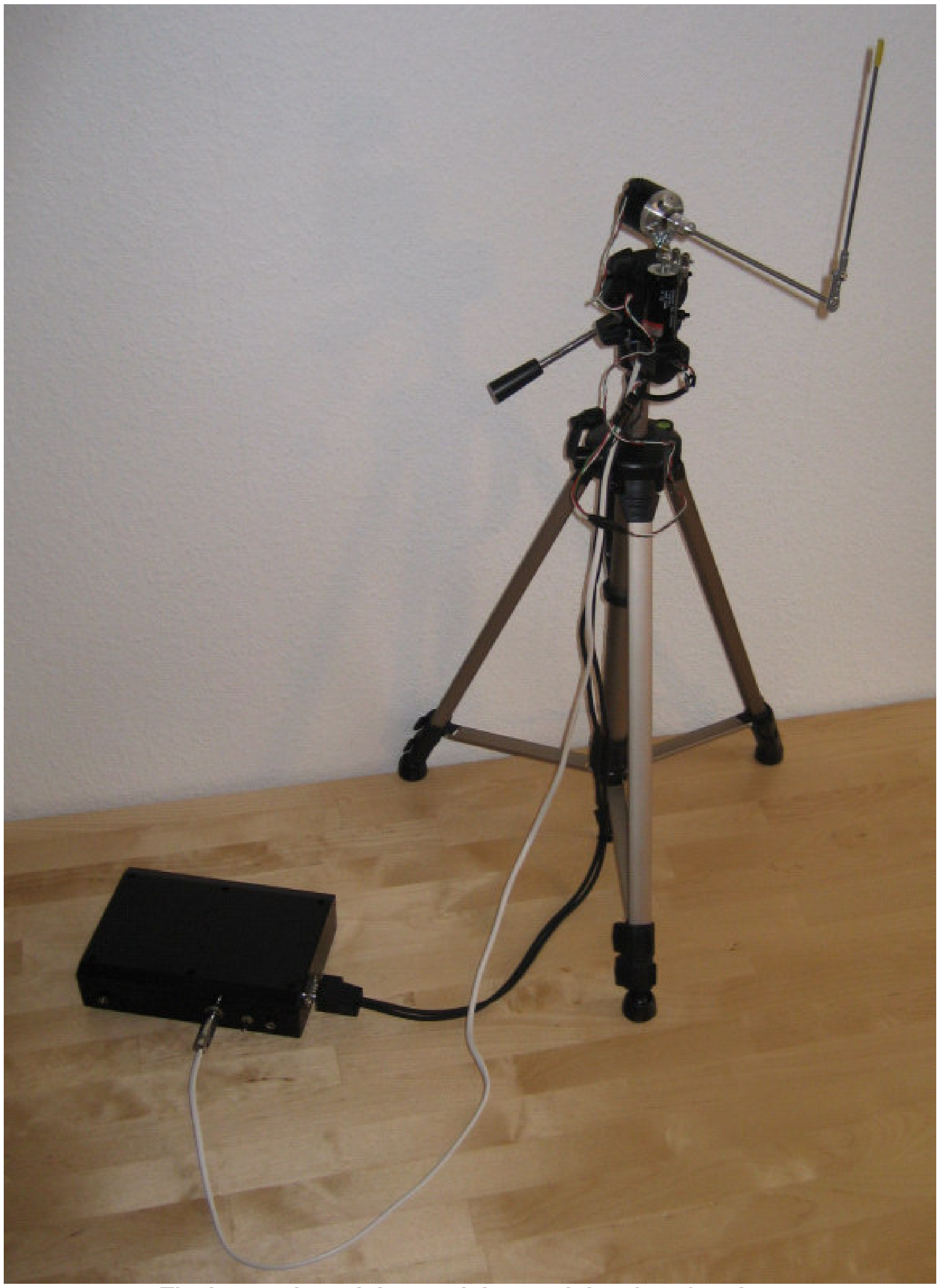
As mentioned remote balancing was implemented in HORB, which is a Java-based ORB, and JavaRTM, which is a component-based network middleware. Remote balancing was possible both using HORB and the UDP/TCP versions of JavaRTM. HORB performed very well (1.2 ms servo-loop). The UDP/TCP versions of JavaRTM used 2.1 and 2.6 ms servo-loops. The Orbacus version of JavaRTM failed balancing the pendulum due to the unpredictable response times.

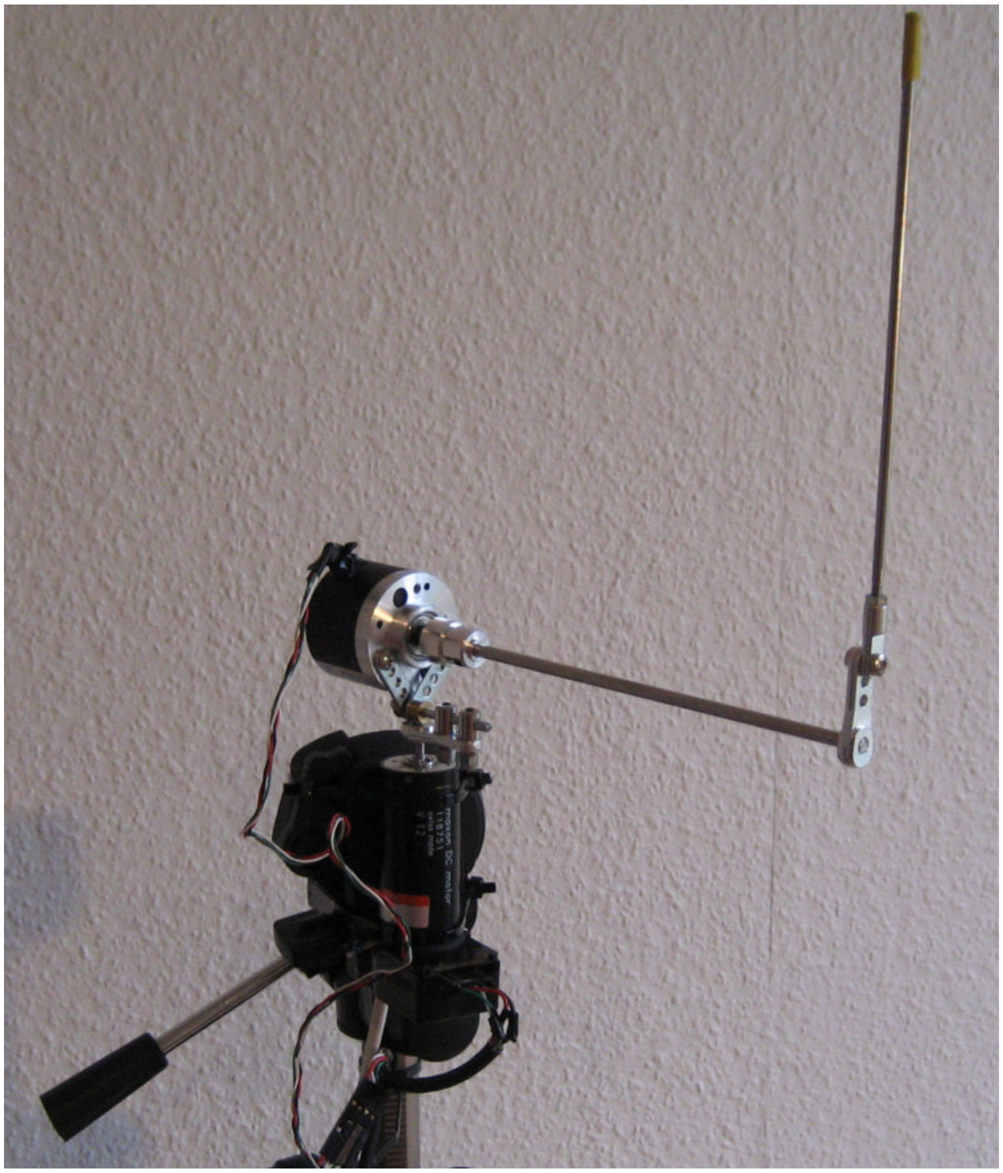
The author believes standard Java can be good enough for single-thread soft-realtime applications but RTSJ might be needed for more complex architectures, which was shown in the case of JavaRTM with CORBA.

References

[1] Alan Burns and Andy Wellings, Real-time Systems and Programming Languages, Addison Wesley, 2001.
[2] Jane W. S. Liu. Real-time Systems, Prentice Hall, 2000.
[3] Phillip A. Laplante. Real-time Systems Design and Analysis, IEEE Press, 2004.
[4] Andy Wellings. Concurrent and Real-Time Programming in Java, John Wiley & Sons, 2004.
[5] Wayne Wolf. Computers as Components: Principles of Embedded Computer System Design, Elsevier, 2005.
[6] Qing Li. Real-time concepts for Embedded Systems, 2003, CMP Books.
[7] Michael Barr. Programming Embedded Systems in C and C++, O'Reilly, 1999.
[8] Stuart R. Ball. Embedded Microprocessor Systems: Real World Design, Newnes, 2000.
[9] Joseph A. Fisher. Embedded Computing, Morgan Kaufmann, 2005.
[10] Jean-Pierre Deschamps, G. J. Bioul and Gustavo. D. Sutter, Synthesis of Arithmetic Circuits: FPGA, ASIC, and Embedded Systems, John Wiley & Sons, 2006.
[11] Bob, Zeidman, Back to the Basics: All about FPGAs, Embedded.com, 2006. (http://www.embedded.com/columns/showArticle.jhtml?articleID=183701900).
[12] http://www.opencores.org/

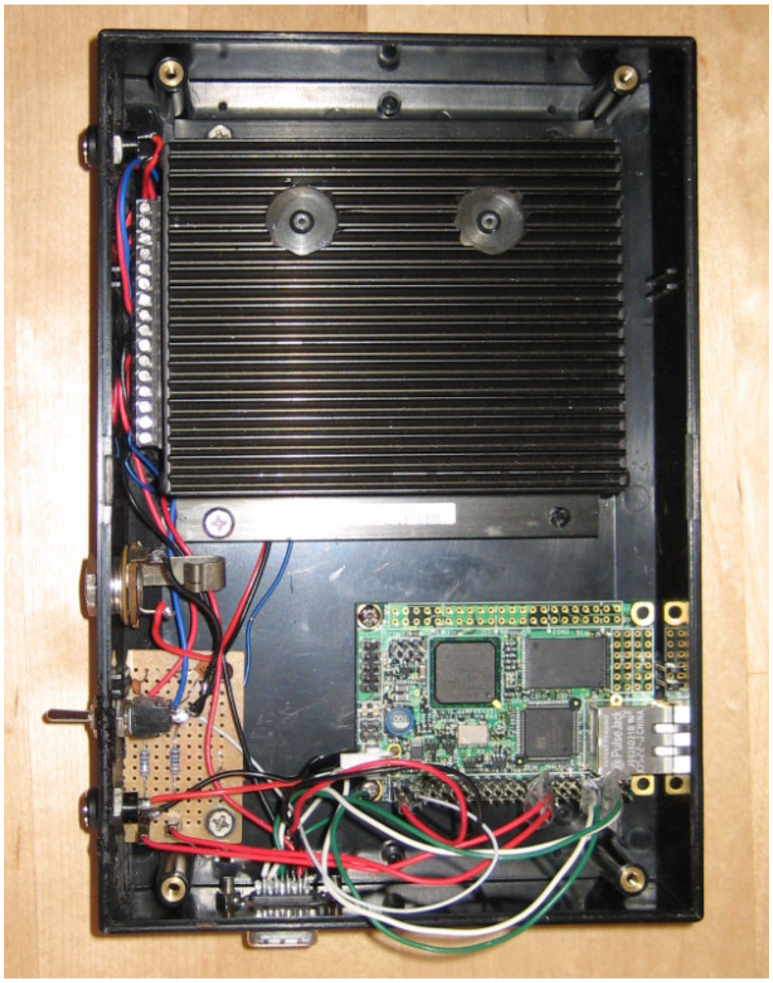
Appendix A: Pictures of the inverted pendulum


The inverted pendulum and the pendulum interface box



The inverted pendlum

The pendulum interface box


Inside of the pendulum interface box

Appendix B: How to use NFS on SUZAKU-V

Normally when mounting a directory with NFS the following command is used:

```
mount –t nfs 192.168.0.1:/remote_directory /tmp/remote
```

However, in the uCLinux-build that is provided with SUZAKU-V this won't work. There are two problems:

- You will have to wait 5 minutes when mounting the remote directory (but after this the remote directory will actually be available for use).
- It will only be possible to copy or load files under about 100 k.

The problems can be solved as follows:

- Use the nolock option to avoid the 5 minutes timeout.
- Use the rsize and wsize options to specify a certain buffer size. In my case 4096 worked well. Others have reported that 1024 worked well for them.

The final command is:

```
mount –t nfs –o nolock, rread=4096, rwrite=4096 192.168.0.1:/dir /tmp/mnt
```

Appendix C: How to compile HORB with GCJ

This article shows how to compile one of the example programs from “HORB Flyer’s guide” with GCJ and running it on SUZAKU-V.

Installing HORB 1.3

To use HORB with GCJ you will need HORB version 1.3. HORB version 2.0 does not work with GCJ (because of a problem with ObjectInputStream in GCJ). To run HORB 1.3 you will also need JDK 1.3.

- Download the file horb13b4a.zip.
- Unzip horb13b4a.zip.
- Install HORB 1.3 as described in doc/guide/install.htm.

Compiling the example program

You will compile and run the example program from section 2.2 of "HORB Flyer's guide". This example consists of two parts; a client and a server. The client can be run in SUZAKU-V and the server can be run in another computer, for example, the PC you’re working on.

Save the files Client.java and Server.java in a new directory.

As usual, first generate skeleton and proxies:

```
horbc Server.java
horbc –c Client.java
```

Compile the client with GCJ:

You will need to adjust the path to the file horball.jar. The file is located in the lib directory of HORB’s installation directory. Also make sure your classpath contains the horb installation directory.

```
powerpc-linux-gcj -O3 -c Client.java Server.java Server_Proxy.java ~/horb/lib/horball.jar
```

The files Client.o, Server.o, Server_Proxy.o and horball.o will be produced.

Link the files above to produce an executable (the Client program):

```
powerpc-linux-gcj -static --main=Client -o Client Client.o Server_Proxy.o Server.o horball.o
```

Running the example

At the server-side (for example your PC) start the HORB server process (which in turn starts the Server class):

```
horb –v
```

At the client-side (in SUZAKU-V) start the Client program:

```
./Client [ip-to-server]
```

Appendix D: How to build GCJ for PowerPC and uCLibc

Introduction

To build a GCJ you will need to build a cross-compilation toolchain. Using that toolchain you will be able to cross-compile programs for Power PC 405. One easy and common way to build such a toolchain is by using the Crosstool script. However, the current version of Crosstool seems to be mainly meant for building glibc based toolchains. Programs compiled with a compiler generated by crosstool will not work in uCLinux (since uCLinux uses uCLibc).

Buildroot is another set of scripts to generate a cross-compilation toolchain specifically for uCLinux and uClibc. By running the Buildroot script you can build a GCJ that works for SUZAKU-V.

Prerequisites

For these instructions software of the following versions were used:

- GCJ 3.4.3
- Buildroot 2006-06-17
- The uCLinux distribution from 2005-11-10.
  It can be downloaded from Atmark Techno.

Instructions for building GCJ

1. Download the version of buildroot from 2006-06-17.

It can be retrieved as follows (this requires that you have svn installed):

```
svn co -r'{2006-06-17}' svn://uclibc.org/trunk/buildroot
```

2. Configure buildroot to build the specific version of the cross-compiler you need.

The following configuration should work well:

- Linux kernel headers 2.4.27
- GCC 3.4.3
- uCLibc 0.9.28

Run the build configuration:

```
make menuconfig
```

2.1. First select Target Architecture. Choose powerpc.
2.2. Go to Toolchain Options.
2.3. Select Kernel Headers Linux. Choose version 2.4.27.
2.4. Uncheck the option to use latest snapshot of uCLibc.
2.5. Select GCC compiler Version. Choose version 3.4.3.
2.6. Check the option that says "Use software floating point by default".
2.7. Exit the menuconfig. Choose that you want to save your configuration

3. Configure buildroot to build GCJ.

This is done by adding the following lines to the file named *.config* (it's in the root directory of buildroot). These lines should be added in the section named *Gcc options*.

```
BR2_INSTALL_LIBSTDCPP=y
BR2_INSTALL_LIBGCJ=y
```

4. Start the build process.

```
make
```

4.1. You will be prompted with the question: “Target CPU has a floating point unit? [y/n]”. SUZAKU-V does not have a FPU, so answer no.

The build took 4-5 hours to complete on my 3.2GHz machine.

5. When the build has finished.

If the build finish with some error message related to building busybox, that's OK. You don’t need to compile all of busybox. The important thing here is to build a working GCJ.

However, if the build has an error before the java-library was compiled then you won't have a working GCJ. Use step 7 to verify that a working GCJ was produced.

6. Verify the build.

If the build was successful the GCJ compiler will be in: ~/buildroot/build_powerpc_nofpu/staging_dir/bin/powerpc-linux-gcj. Try to compile a simple java program and put it in your target computer.

Save the helloworld-program below in a file named Hello.java.

```
public class Hello
{
    public static void main(String args[])
    {
        System.out.println("Hello world! Using Java on SUZAKU-V.");
    }
}
```

Use GCJ to compile it as follows:

```
powerpc-linux-gcj -O3 -c Hello.java
powerpc-linux-gcj -static --main=Hello -o Hello Hello.o
powerpc-linux-strip Hello
```

Transfer the helloworld-program to SUZAKU-V using ftp:

```
ftp 192.168.0.x (ip of SUZAKU-V)
220 SUZAKU-V FTP server (GNU inetutils 1.4.1) ready.
ftp> cd /tmp
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> put Hello
local: Hello remote: Hello
```

At the SUZAKU-V console try run it:

```
# cd /tmp
# chmod +x Hello
# ./Hello
Hello world! Using Java on SUZAKU-V.
```

Appendix E: How to write device drivers for SUZAKU-V

This article describes how to write device drivers for SUZAKU-V.

Loadable modules

There are two ways of compiling a device driver: statically built into the kernel or as a module. A statically built device driver requires the kernel to be recompiled after making a change in the driver. Modules have the advantage that they can be loaded and unloaded during runtime. Therefore, during development it becomes necessary to compile the device driver as module.

In the uCLinux distribution shipped with SUZAKU-V loadable modules support is not enabled in the kernel. To turn it on, some configuration and a recompilation of the kernel is needed. The configuration is described below:

```
cd uClinux-dist-20040408-suzaku6
make menuconfig
```

- Select Kernel/Library/Defaults Selection.
- Check Customize Kernel Settings option.
- Select exit two times.
- Do you wish to save your new kernel configuration? Yes
- Select Enable loadable modules support.
- Check the Enable loadable module support option.
- Uncheck Set version information on all module symbols option.
- Check the module loader option.
- Exit and choose to save config.

Then recompile the kernel:

```
make dep; make
```

A minimal device driver

The code below shows a simple device driver which prints “Hello world” once it gets loaded.

hello.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <asm/io.h>
```

```
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");

static int fpga_init(void)
{
   printk("Hello world from Kernel space!\n");

   return 0;
}

static void fpga_exit(void)
{
   printk("driver unloaded\n");
}

module_init(fpga_init);
module_exit(fpga_exit);
```

The driver can be compiled as follows:

```
powerpc-linux-gcc -D__KERNEL__ -DMODULE -DLINUX -c hello.c
```

Load the module:

```
insmod hello.o
Hello world from Kernel space!
```

Unload it:

```
rmmod hello
```

Interfacing with hardware

Let's see how a device driver can communicate with a simple device. This example will show how to control the LED that is placed on the SUZAKU-V board.

The SUZAKU-V board has 70 external I/O pins. The FPGA can be programmed to map an I/O port to a memory address. To connect the LED to a memory address an IP core called GPIO (general purpose I/O) can be been added in the Xilinx tool. For this example the GPIO was set up to connect the LED which is located at pin A9 to memory address 0xF0FF8003.

The driver will turn the LED on when the module is loaded and turn it off when the module is unloaded.

led-driver.c:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/slab.h>
```

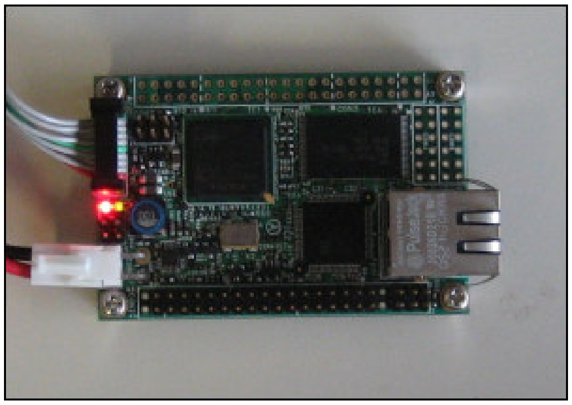
```
#include <asm/io.h>
#include <asm/uaccess.h>

#define LED 0xF0FF8003

MODULE_LICENSE("GPL");

static int fpga_init(void)
{
   printk("driver loaded\n");

   void* location = ioremap(LED, 1);
   // Turn the LED on
   writeb(0x00, location);
   iounmap(location);

   return 0;
}

static void fpga_exit(void)
{
   printk("driver unloaded\n");

   void* location = ioremap(LED, 1);
   // Turn the LED off
   writeb(0xFF, location);
   iounmap(location);
}

module_init(fpga_init);
module_exit(fpga_exit);
```

Compile the driver:

```
powerpc-linux-gcc -D__KERNEL__ -DMODULE -DLINUX -c led-driver.c
```

Load the module:

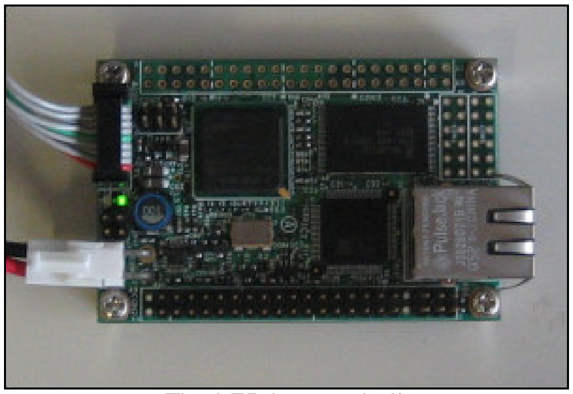
```
insmod led-driver.o
```



The LED lights up.

Unload it:

```
rmmod led-driver
```


The LED is turned off.

Exposing the device to user-space

In this section the device driver written by Ford Sleeman will be examined. It's a good example of how to make the driver accessible to user-space. Below I give short explanation of some of the important concepts introduced in the code.

There are three different types of device drivers: character devices, block devices and network devices. In this example a character device is used. A character device is exposed to user-programs as a file which can be read or written.

Device driver registration

To make the device accessible to user-programs the device driver needs to register itself.

The registration is done with the register_chrdev function:

```
register_chrdev(250, "fpga", &fpga_fops);
```

The device driver's major number maps the device file to the driver. In the function call above the character device with the major number 250 is registered. A device file with the same major number needs to be created. A user-program can then use the device file to read and write to the driver.

In most Linux systems a device file can be created by issuing the following command:

```
mknod /dev/fpga c 250 0
```

However, SUZAKU-V has a read-only file system so this command won't work.

New devices can be added for SUZAKU-V by entering them into the file /vendors/AtmarkTechno/SUZAKU-V of the uCLinux dist directory. If you open that file you will see all the devices listed. Add the new device by entering "fpga,c,250,0" after the last device and before the FLASH_DEVICES section starts.

```
DEVICES +=
mtd0,c,90,0     mtdr0,c,90,1     mtdblock0,b,31,0
mtd1,c,90,2     mtdr1,c,90,3     mtdblock1,b,31,1
mtd2,c,90,4     mtdr2,c,90,5     mtdblock2,b,31,2
```

```
mtd3,c,90,6    mtdr3,c,90,7    mtdblock3,b,31,3
mtd4,c,90,8    mtdr4,c,90,9    mtdblock4,b,31,4
mtd5,c,90,10   mtdr5,c,90,11   mtdblock5,b,31,5
mtd6,c,90,12   mtdr6,c,90,13   mtdblock6,b,31,6
mtd7,c,90,14   mtdr7,c,90,15   mtdblock7,b,31,7   fpga,c,250,0
```

Communicating with the driver from a user-program

The driver-code has read and write functions shown below:

```
ssize_t fpga_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
ssize_t fpga_write(struct file *filp, unsigned char *buf, size_t count, loff_t *f_pos)
```

These functions will be invoked when a user-program reads or writes to the device file.

The user-program can open the device as follows:

```
int handlemem = open(”/dev/fpga”, O_RDWR);
```

Data is then read or written using the functions pwrite and pread.

Appendix F: An example of how to access hardware using Java (without JNI)

This example shows how the LED on SUZAKU-V can be switched on and off depending on whether a jumper is open or closed.

First the code for the Java application is shown. The program checks if the jumper is open. If it's open the LED will be turned on.

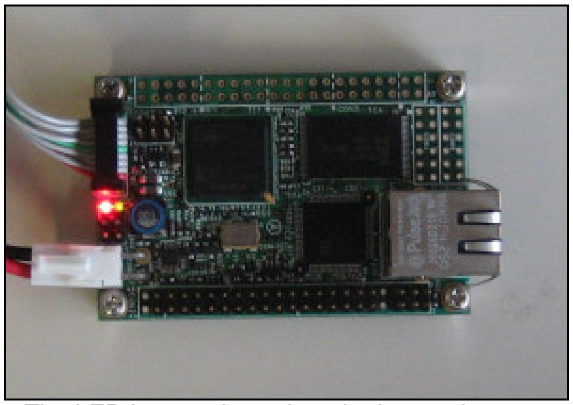
```
import java.io.*;

class Controller  {

   public static int JUMPER_OPEN = 4;

   public static int LED_ON = 0;
   public static int LED_OFF = 1;

   public static void main(String argv[]) {
       try
       {
           FileInputStream in = new FileInputStream("/dev/fpga");
           FileOutputStream out = new FileOutputStream("/dev/fpga");

           while(true)
           {
              if(in.read() == JUMPER_OPEN)
                out.write(LED_ON);
             else
                out.write(LED_OFF);
           }
        }
       catch(Exception e)
       {
           System.out.println(e);
       }
   }
}
```

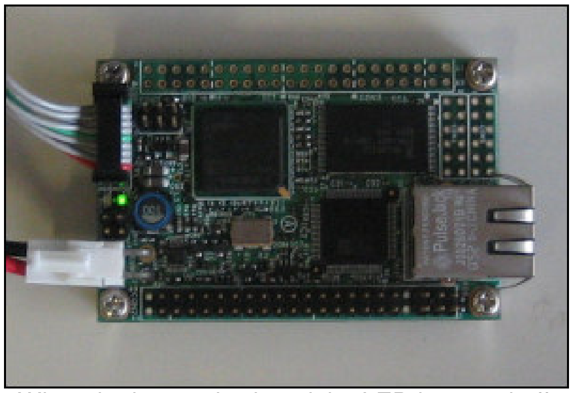
Next the source code of the device driver is shown.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <asm/io.h>
#include <asm/uaccess.h>

#define LED 0xF0FF8003
#define JUMPER 0xF0FFA003

MODULE_LICENSE("GPL");

ssize_t fpga_read(struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
       unsigned char val;
       if(count != 1) return 0;

       // Read value of the JUMPER
       val = readb((void *) JUMPER);
```

```
	copy_to_user(buf, &val, 1);
	*f_pos += 1;

	return 1;
}

ssize_t fpga_write(struct file *filp, unsigned char *buf, size_t count, loff_t *f_pos)
{
	unsigned char val;
	if(count != 1) return 0;

	copy_from_user(&val, buf, 1);
	void* location = ioremap(LED, 1);

	// Write value to the LED
	writeb(val, location);
	iounmap(location);

	return 1;
}

struct file_operations fpga_fops = {
	read: fpga_read,
	write: fpga_write,
	poll: NULL,
	open: NULL,
	release: NULL,
};

static int fpga_init(void)
{
	int result = register_chrdev(250, "fpga", &fpga_fops);

	if(result > 0)
		printk("Failed to register device\n");
	else
		printk("Device registered\n");

	return 0;
}

static void fpga_exit(void)
{
	printk("Unloading driver.\n");
	unregister_chrdev(250, "fpga");
}

module_init(fpga_init);
module_exit(fpga_exit);
```

Running the program


The LED is turned on when the jumper is open.


When the jumper is closed the LED is turned off.