REAL SIMULATION CONTROL

- Design of Communication for a Simulator Device -

M.Sc.Thesis by
Ola Hammarlund, M98
Dan Persson, M99

Supervisors:
Gustaf Olsson, IEA, LTH
Istvan Ulvros, Tetra Pak R&D

Abstract

This master thesis work presents a solution for a communication between a real PLC and a simulated machine. In the industry today there has for a long time been a desire to find a way to test and verify a PLC program before a prototype of the machine is built. If this could be done it would shorten the developing time in the primary design stage, give the engineers the possibility to analyse the behaviour of the simulated machine without any damage and verify the functionality without having to struggle with any kind of accessibility. All these advantages will certainly reduce the development costs.

We have had at our disposal a PLC program, simulation software and a simulation of a machine. What needs to be done is to choose a fieldbus and develop a device driver between the bus and the simulation software. To get a solution that is as general as possible, the conditions are that a commonly used fieldbus has to be used and the device driver to be developed shall only transfer process data between the PLC and the simulation, because it must be possible to reuse the device driver for another simulation.

Profibus DP has been the selected fieldbus, because Tetra Pak in Europe uses it. The device driver that is developed contains actually two device drivers. One on the application side is written in Java, and one on the fieldbus card side is written in C++. These two device drivers have to be tied together using the Java Native Interface technique. The software are divided into two parts, one for making the model, Lumeo Motion, and one for running and controlling the model, Lumeo Scenario.

We can conclude that it is possible to control a model in Lumeo Scenario by an external controller.

Preface

In today's industry, one is always trying to be one step ahead of everything. The subject for this thesis work is no difference. Here we try to be one step ahead of a prototype, using a simulation for developing the control system before the prototype is ready. In a few years this will probably be the common way to do it, because saving time equals saving money.

This master thesis work has been carried out at Tetra Pak R&D Packaging Process Control in Lund, between December 2002 and June 2003, and with support from Lumeo Software. We will take this opportunity to thank all people around us that has made it possible to complete this work, all included.

Ola Hammarlund
Dan Persson
Lund, June 2003

1 Introduction

In this opening chapter the background to the problem is discussed and what benefits there could be achieved if the problem at hand could be solved. Further details about the ambition with this thesis work are also described with respect to objective and method. In the last section an overview of the report is given to facilitate a perusal for the reader.

1.1 Introductory Background

The company Tetra Pak has for some years tried to find out how it would be possible to reduce costs as well as save developing time in the primary design stage. One way would be to create a computer-simulated device, lets say some kind of machine idea, and add dynamics and other constrains that would agree with a similar machine in reality. Furthermore an external programmable logic controller connected with a commonly used fieldbus should control this simulated machine inside of the computer, so that it would respond to the signals in the same way that a real machine should have done. This would give the designers the ability to implement, test and analyse the behaviour of the virtual machine, without risking any damage of valuable real equipment, and any change of the functionality could easily be made without having to consider accessibility, power shortage etc. With all this information it would be possible to decide if the project should proceed or be revoked in an early stage, and with that save a lot of unnecessary costs for the company.

Previous thesis works, Sivtoft (2002) and Norén (1999), has tried to solve this challenge by using different programs for simulation and communication, and then controlled the simulation with a soft PLC. The conclusion has been that that there were no suitable programs for this purpose on the market. A company in Finland, Lumeo Software, has developed an add-on to the program ProEngineer (ProE), a software package for Computer Aided Design (CAD). In this program, Lumeo Motion, it is possible to take an assembly made in ProE and add dynamics and other constrains that apply to the model. To see how the model acts in a simulation it is possible to test it via another software module, Lumeo Scenario. In Lumeo Scenario the movements of the model can be tested manually or with digital/analog controllers. There are possibilities to interact with the reality, but it is necessary to develop a device driver that suits the specific requests.

1.2 Objective

The goal is to verify that it is possible to control a simulated machine in the program Lumeo Scenario with an external control system. The main focus in this thesis work will be on the functionality and not on any kind of optimisation. To achieve this we have to develop and implement a device driver between the application and a common fieldbus interface. We will show that a simulation of the real machine, with the same control system can interact and response on signals from a control system in the same way as the real machine, and that it presents the same pattern of behaviour. Furthermore, the ease of use for the designers has to

be tested. This becomes even more important when the simulation consists of many axes with a high degree of complexity.

1.3 Methods

The work has started with a literature study with emphasis on fieldbuses, programming in C++ and Java. Fieldbuses are important because they will be used to communicate between the control system and the computer, and programming in C++ and Java because the fieldbus interface is written in C++ and the Lumeo Scenario interface is written in Java. This makes it also necessary to find a communication link between C++ and Java. It is also necessary to find out how to talk with the control system, what signals to receive and what signals the control system expects from the system. Furthermore we have had to understand and learn how the program from Lumeo software works and in particular how signals can be used in Lumeo Scenario to control different CAD parts in an assembly such as motors, sensors etc.

Tetra Pak have provided a test rig, a CAD model of the same test rig and a PLC program. Furthermore Lumeo software has applied some of the dynamic constrains on the CAD model and provided a small code example of a link between the Lumeo application and the fieldbus card. Our assignment has been to set up a communication between the computer and the PLC with a fieldbus, design a device driver for the simulation program and control the simulation.

1.4 Outline of the Report

After this introductory chapter, a brief presentation of the two companies that are involved in this thesis work is done. Then in chapter three the task is described, and what paraphernalia we have to work with. We have then divided the communication link in two chapters, one chapter about the communication that is outside of the computer and one for the internal communication. Both these chapters begin by given the reader some background information about fieldbuses respective Java, then follows a description of our implementation and some other alternatives are mentioned in the later parts. In the sixth chapter we explain the features in the programs from Lumeo Software that we use for the simulation of the test rig. Chapter seven is all about the real world; here we describe the test rig and its control system. All code that has been developed during this thesis work can be found in the appendixes, we have provided the code with a lot of declaring comments to make it easy to follow and understand.

2 Company Presentations

This thesis work is done commissioned by the company Tetra Pak, and the company Lumeo Software supplies the simulation software. Therefore a brief presentation of both companies follows in this chapter.

2.1 Tetra Pak

The Tetra Pak history starts back in 1929 when Dr Ruben Rausing and Erik Åkerlund founded Scandinavia's first specialized packaging factory, Åkerlund & Rausing. In 1943 they started a primary design stage at Åkerlund & Rausing which aim was to create a milk package that needed a minimum of material but provided the maximum of hygiene. Tetra Pak was founded in 1951 as a subsidiary company to Åkerlund & Rausing and one year later the first tetrahedron shaped package were sold from Lundaortens Mejeriförening.

In 1991 Tetra Pak acquired Alfa-Laval, one of the world's largest suppliers to dairy farmers and milk production. Today's organization structure can be seen in Figure 1. Tetra Laval International is responsible for financing, monitoring the overall legal structure such as tax planning, debt/equity ratios as well as the executing of all mergers and acquisitions transactions

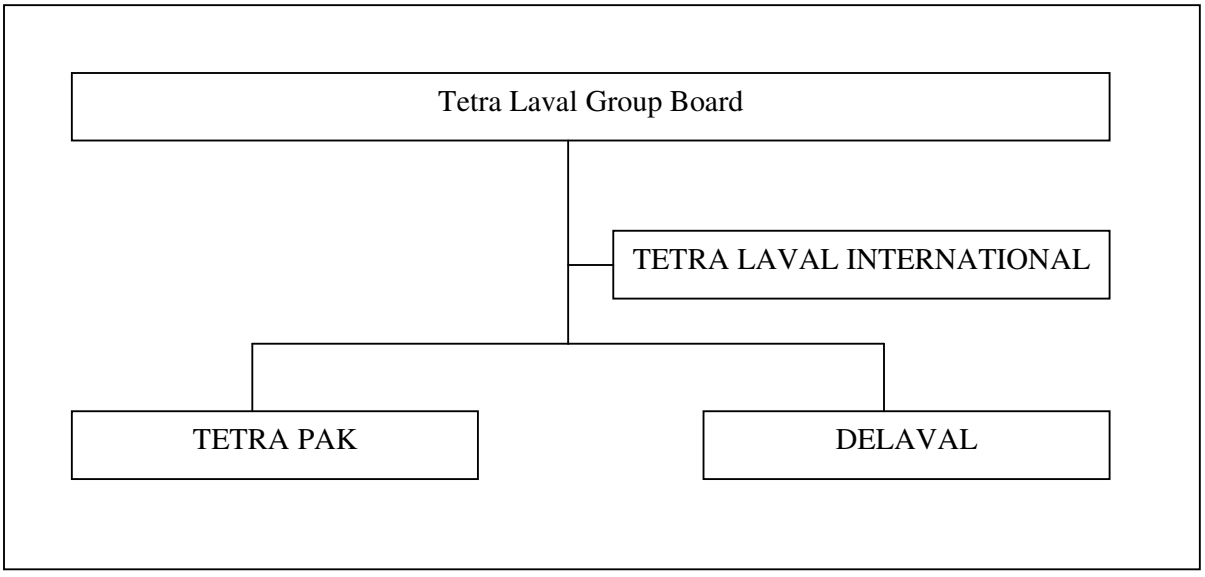


Figure 1. Tetra Laval organization structure.

Today Tetra Pak is one of the largest packaging companies with over 20 000 employees in 165 countries all around the world and with a net sales of EUR 7.6 billon. In 2001 there were produced 94 billion packages and with the new package Tetra Recart, a alternative to more traditional tin can and glassworks, there will be produced even more in the future.

2.2 Lumeo Software

Back in 1994, the Canadian company Lateral Logic Inc started to develop interactive simulation system. They started by developing several custom-built simulation programs. In April 1999 where Lateral Logic Inc acquired by the company MathEngine PLC and Lumeo Software acquired the assets and R&D work of MathEngine OY in 2000. It is this R&D work that consisted of three projects that lies as an foundation for the programs used in this thesis work, Lumeo Motion and Lumeo Scenario.

Lumeo Software is a small company having its headquarter in Esbo, Finland. There is also an affiliated company in Toronto, Canada. In Finland there are sixteen employees of which eight works with development and the remaining with administration, consulting and sales.

The company main focus is on creating easy to use applications that can work inside leading CAD programs. This strategy is based on that the number of mid-end users will increase much faster than the high-end users. The software that we use in this thesis work is a plug-in application to the CAD program ProE, and is called Lumeo motion. With this plug-in it is possible to add constraints to the CAD model. We are also using Lumeo scenario, which is a simulation program where it is possible to control the model with several of choices.

3 The Challenge

Here we look into the task that we are confronting. The actual background to the given task is discussed and the question why it is done is answered. In the later sections a presentation of the problem is formulated, and in the end an account of what equipment we have at our disposal is given.

3.1 Background to the Task

Tetra Pak wants to test if it is possible to control a computer-simulated machine with a real PLC and use the same PLC program to control the computer-simulated machine and the real one, se Figure 2. Another important aspect is that it has to be no differences in behaviour between the simulated machine and the similar real machine using the same control program. The aim is to be able to have a tested control program ready to use before the actual machine has been built. This can be achieved if the programmers can start their work as soon as a CAD-model has been made. There are even more advantages to gain by use this approach. For example with a simulator it is possible to analyse the behaviour of a machine idea without any damage, test the process for robustness and make function verifications without the constraints of real machinery like accessibility etc.

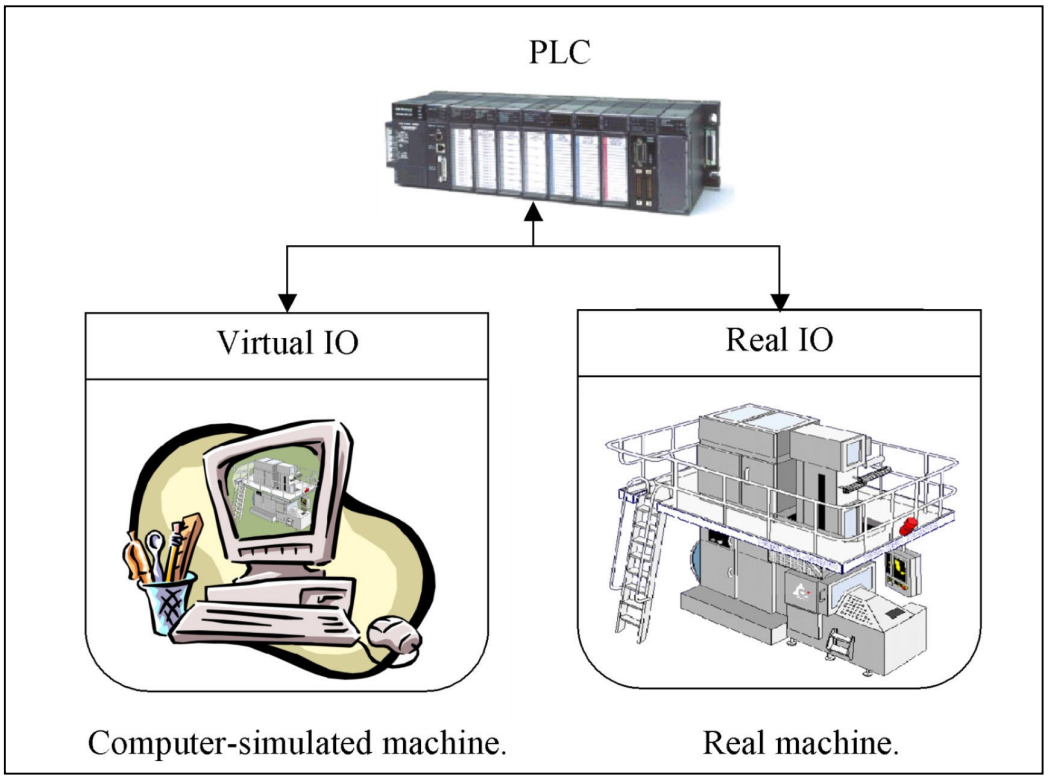


Figure 2. One control system for the both applications, virtual and real.

3.2 Specification of the Task

The task is to make a communication link between a PLC and all the way into the simulation interface in the computer, to prove that interaction with the real world is possible. The simulation interface works with Lumeo Scenario and is written in Java code. Furthermore the PLC receives and sends commands for the behaviour of the simulation. A device driver has to be designed and implemented in C++ to work between the Profibus and the simulation. To make the device driver as general as possible it will be designed to only pass process data back and forth. Inside the simulation program the process data has to be interpreted so that it make sense for the simulation and makes the simulated machine perform in the right manner. Furthermore the Profibus connection has to be configured. So, we have to configure the fieldbus to make it talk to the PLC and design a device driver to handle the communication. The outline of the communication is shown graphically in Figure 3.

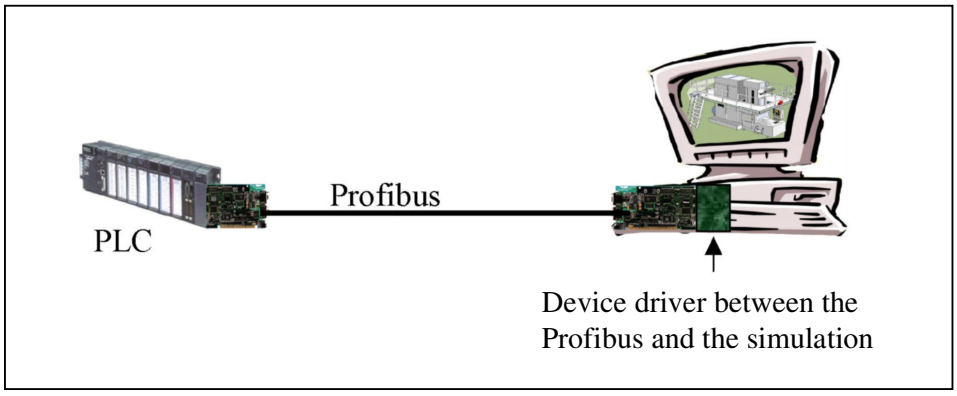


Figure 3. Outline plan for the communication.

3.3 Available Equipment and Software

The real machine that will be simulated is a prototype under development at Tetra Pak. It is designed for some treatment of packages. Since this machine is so big, Tetra Pak has built a small but quite adequate machine for testing purposes called the test rig. It consists of two linear motors and two index motors, these four motors are controlled through a module called LinMot. The test rig is built for this purpose only and is not a pre design study but works fine to demonstrate that the principles of simulation and communication are working. Tetra Pak has made a CAD model of the test rig in ProE and Lumeo Software has, after consulting with us, made the necessary modifications in the Lumeo software in order to make the model as realistic as possible.

The fieldbus that we use is a Profibus, which is a commonly used fieldbus in the industry today in Europe. Worth mentioning is that the choice of fieldbus is not of any greater importance. If it works with Profibus it most likely will work with other fieldbuses like CAN-open, netbus etc. In the computer we have installed a Hilscher Profibus slave pc card, CIF30 DPS. To make all the necessary configurations we use the program SyCon that came with the Profibus card. For the development and implementation of the device driver to the Profibus card we use the documentation that came with the Profibus slave pc card.

From Lumeo we have received the Lumeo Motion software that is an add-on to ProE and is the software module where all the constraints and dynamics are added to the model. To show the model in motion we use Lumeo Scenario. This software is used for the movement control signals.

4 Fieldbus Connection

To understand the basics of bus communications, a general survey is done in the following chapter. As a background to the first section about general aspects on fieldbus, the reference Olsson (2002) and Nordbladh (2002) has been used. We continue with a more detailed description of the fieldbus Profibus, which will be used in the further development. At the end, a few alternative field buses are described.

4.1 General Aspects on Fieldbus

In order to establish a communication between the PLC and the computer we need to have some kind of data-bus. Since the application is going to be used in the industry with different kind of disturbances a fieldbus communication has been specified. A fieldbus is a generic-term that describes a digital communications network that will presumably be used in industry to replace the existing 4 - 20mA analog signal. The network is a digital-, bi-directional-, multidrop-, serial bus communications network used to link isolated field devices, such as controllers, transducers, actuators and sensors. An obvious advantage of digital versus analog technique is that you can replace a large number of 4-20 mA conductors with a single digital loop.

Each field device will be able to execute simple functions on its own such as diagnostic, control, and maintenance functions as well as providing bi-directional communication capabilities. With these devices the engineer will not only be able to access the field devices, but also to communicate with other field devices.

Today there is a couple of standards and specifications for different kinds of open fieldbuses. An open fieldbus must be able to satisfy the following demands:

- The specification of the fieldbus must be published and available at a reasonable price.
- The specific electrical components and chip to the fieldbus must be available at a reasonable price.
- A well-defined validity process should be open for every user.

These simple rules are formulated so that any manufacturer can be able to manufacture peripheral equipment and devices that can be connected and communicate on the open fieldbus.

There is a great work going on today with the attempt to standardize the fieldbus technology. In the present position there are a couple of open fieldbuses that are part of different national and international standards. Still there isn't a common standard that different supporters/manufacturers are willing to accept. The attempt to gain a common standard was started too late. Already several big open fieldbuses were supported by large companies (Profibus – Siemens, DeviceNet – Allen Bradley etc.).

The OSI-model (Open System Interconnection) describes communication between the stations of a communication system. It was the International Organization for Standardization (ISO) who developed the OSI reference model in 1983. It is based on a layered architecture consisting of seven layers, each of them having their own specific task and function in data transmission and receiving, Figure 4. This gives the benefit that an implementation on one level only has to consider the implementation of the layer above and below.

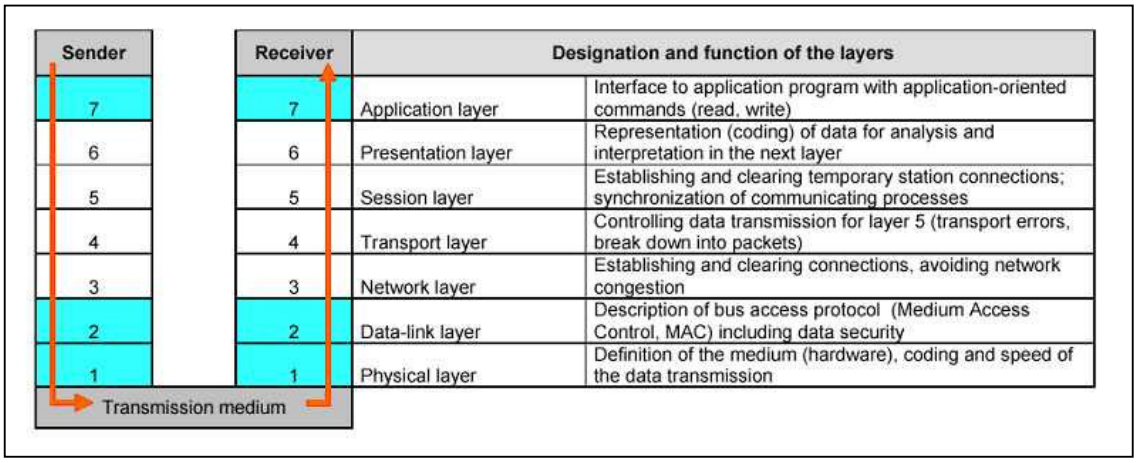


Figure 4. The structure of the OSI layers.

Each layer has to fulfill specified functions within the communication process. If a communication system does not require some of those specific functions, the corresponding layers have no purpose and are bypassed. The OSI-model is applied also in fieldbus system. Their data communication is, however, not very complicated so that only some of the layers of the OSI-model are commonly needed.

Tetra Pak wanted us to use Profibus-DP for the fieldbus connection because they have a long experience of this bus system, so here follows a general description on Profibus.

4.2 Profibus

Profibus, short for Process Field Bus, is a fieldbus and device network technology used primarily in Europe, but gaining worldwide acceptance. The Profibus family follows an open network standard (EN 50170) with hundreds of vendors supporting a common, interchangeable interface and protocol. Several companies and Federal institutes in Germany initially developed Profibus in the late 1980s.

Profibus uses layers 1, 2 and 7 in the OSI-model. Up to 127 stations, divided in active and passive ones, can be connected to the bus depending on the bus cycle time, if there is a repeater (a kind of amplifier) installed etc. Profibus is a multimaster bus that uses a token to administrate who has the right to use the bus, see Figure 5. It is only the master node having the token that is allowed to send messages to other devices.

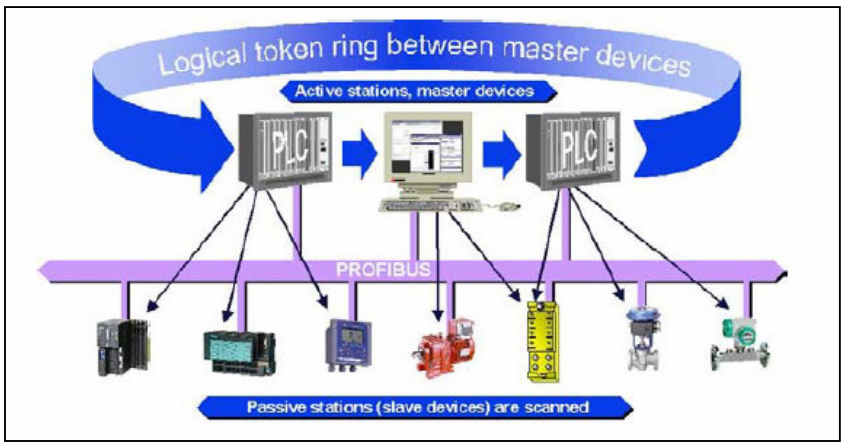


Figure 5. Illustrates a multimaster system.

Since the first specification several other variants has developed and have been adjusted to different types of environments and parts of automation networks. The first one was Profibus FMS (Fieldbus Message Specification) that is a protocol for communication between masters or devices that have to exchange a large amount, but not time critic, information. Then there are Profibus PA that is adjusting to the process industry. PA can be used in an own secured environment and can supply devices over the bus. ProfiSafe is used for secured critic devices, for example at emergency stops. The Profibus DP has been used in our application.

The Profibus DP (Decentralized Periphery) Communication Profile is designed for efficient data exchange at the field level. The central automation devices, such as PLC/PC or process control systems, communicate through a fast serial connection with distributed field devices, which can be I/O, drives and valves, as well as measuring transducers. Data exchange with the distributed devices is mainly cyclic. The master controller cyclically reads the input information from the slaves and cyclically writes the output information to the slaves. The bus cycle time should be shorter than the program cycle time of the central automation system, which for many applications is approximately 10 msec. DP requires about 1 msec at 12 Mbit/sec for the transmission of 512 bits of input data and 512 bits of output data distributed over 32 stations. The chart below shows the typical time, depending on number of stations and transmission speed.

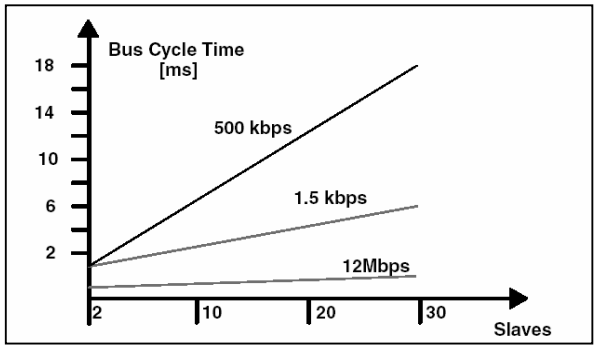


Figure 6. The relation between bus cycle time and the number of slaves in a system is at a rough estimate linear.

Every Profibus slave device come with a GSD-file (General Slave Data). A GSD is a readable ASCII text file and contains both general device-specific specifications for communication that is created by the device manufacturer. Each of the entries describes a feature that is supported by a device. To read more about Profibus visit the Profibus Trade Oragnization web site, www.profibus.com.

4.3 Implementation of the Profibus

For the purpose of this thesis work we need a Profibus slave pc card installed in our computer, since the PLC should work as a master. To start with we had a Profibus master card available that, from a Hilscher source, should be able to work as a slave card. But after trying and then consultation with another Hilscher source we found out that this wasn't possible. So therefore we bought a Hilscher Cif30 DPS card, a real slave card. This vendor was chosen because Tetra Pak has used them before and they are well known by the company. We also needed a Profibus cable, a screened twisted pair cable according to the RS-485 specification, to connect the PLC to our computer. On the PLC-side we have a GE-Fanuc 90-30 series rack. The rack has a base plate with 5 slots for different devices. The first slot is the power supply. In slot 2 we have the racks CPU where the PLC-program is downloaded too. Slot 3 and 4 are I/O modules and slot 5 is the Profibus-DP master module.

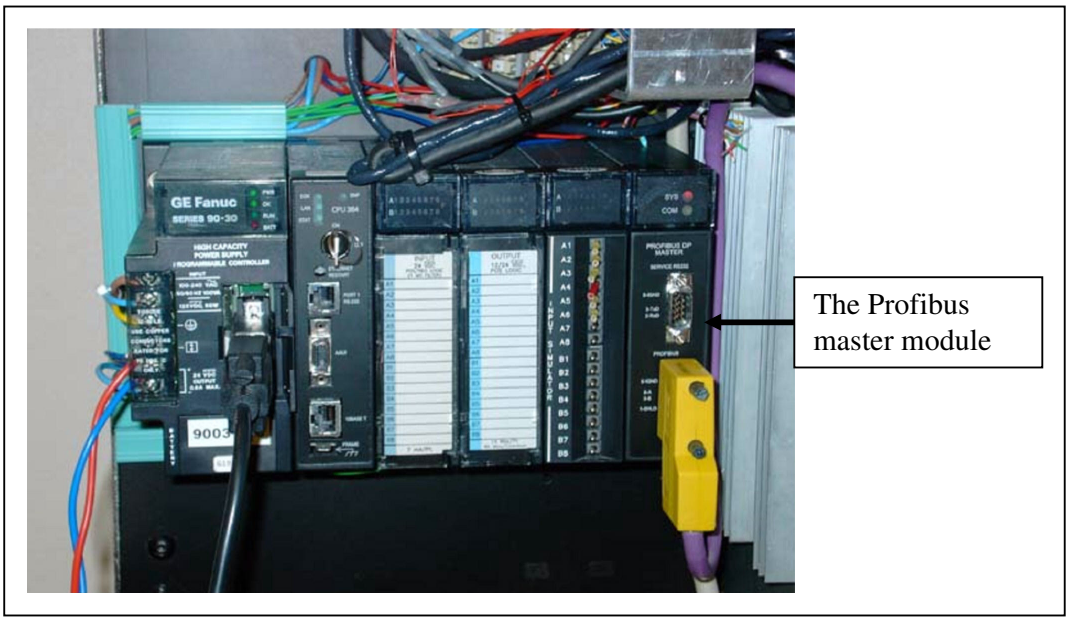


Figure 7. The PLC rack with the Profibus master device.

The PLC and the Profibus master module are programmed and configured by a PC with the program Cimplicity. In Cimplicity you configure what rack the output and input signals should be in. You also import the corresponding GSD-file for the slave, so the master module knows how it can communicate with the slave. On the virtual simulator side we have to install the Hilscher Cif30 DPS pc card. This is done with the drivers that follow with the card. There are a lot of settings to be made, but we choose to set most of them to standard. Our settings are as follows:

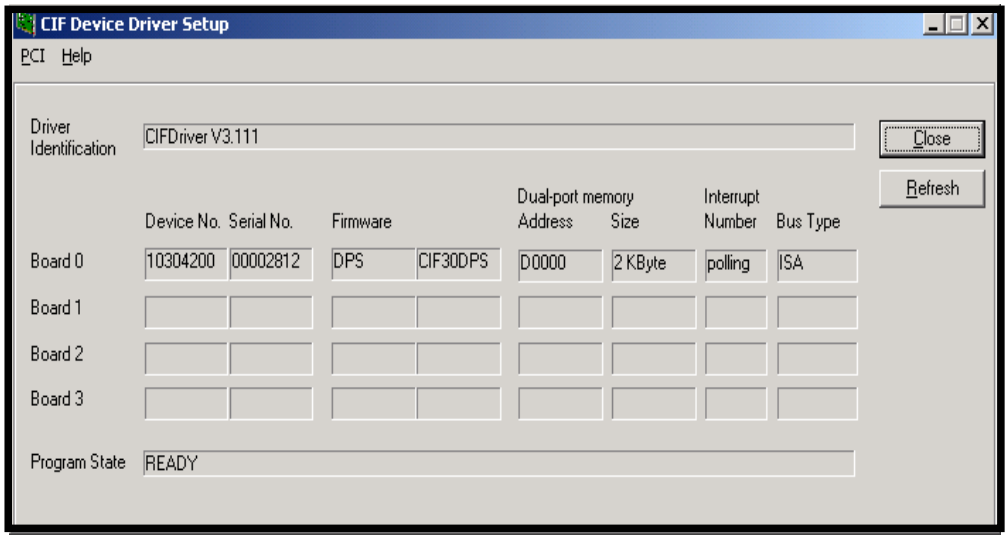


Figure 8. The Profibus slave pc card settings.

The Dual-Port memory on the card is 2Kbyte and the address D0000 is the start address for the CIF-card. The start address is configured manually by setting jumpers on the card. There has to be a free 2Kbyte memory area on the address that is chosen. This setting is necessary to do because it is an ISA-card and it doesn't have Plug-n-Play. We have also tried with a PCI-card that supports Plug-n-Play but we had some conflict on our computer so that didn't work.

The interrupt number can be run in interrupt or polling mode. The difference between them is only the handling of application requests timeout situations. In polling mode the program cyclically ask all inputs if they have new data. In interrupt mode it's the other way around, the card sends an interrupt signal when it have new data.

Since we are working with two different slaves, the LinMot module and the Hilscher Profibus slave pc card, we have two different GSD files. After consulting with Hilscher GmbH we could conclude that it would not be possible to use the LinMot commands with the Hilscher slave pc card, because the GSD files are dependent of the hardware design of the slave it supports. Due to this we hade to make an own protocol that has to act as the LinMot commands and something that makes sense for the application. We decided to use one word in and one word out, that acts as data carrier. For the configuration of the slave pc card we used the configuration program SyCon that followed with the Hilscher slave pc card, where one simply gives the slave an address and decide what kind of data it are supposed to be exchanged, in our case one word in and one word out.

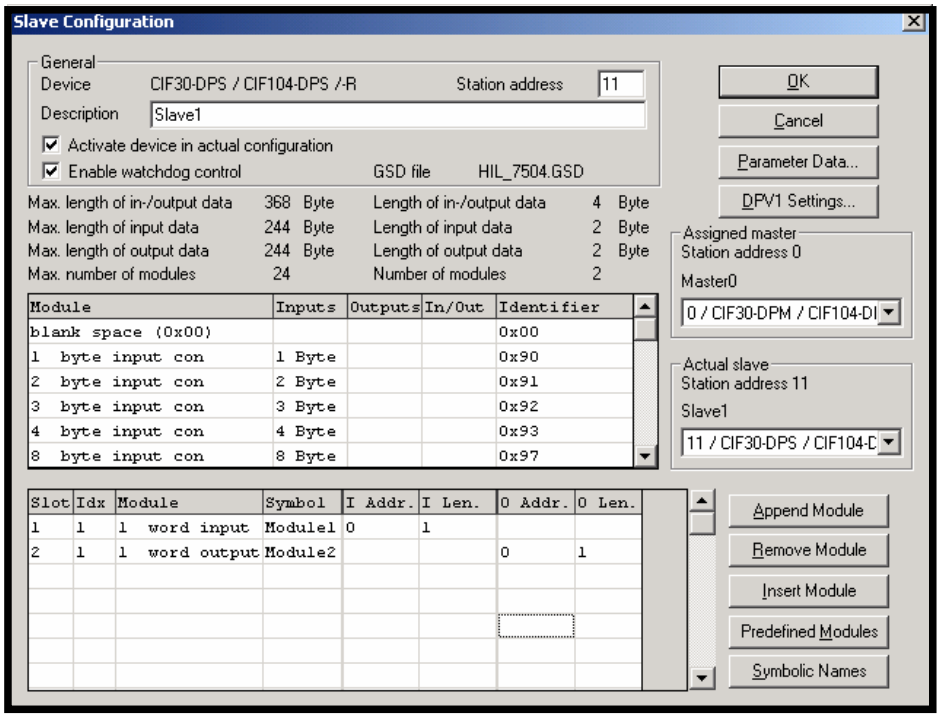


Figure 9. The slave configuration.

4.4 Other Fieldbuses

There are several other fieldbuses on the market but most of them behave in a similar way. So if this project will work with Profibus it would probably work with small modifications together with the other fieldbuses as well. Here follows a short description of some other fieldbuses, the information is mainly from Nordbladh (2002):

Fieldbus Foundation (FF) was created when the two groups Interoperable System Project and WorldFIP North America started collaboration in 1994. FF is today an independent organization with over 100 device vendors. Software and hardware specifications have been written by design and marketing teams, and many products are becoming available that conform to this standard across many different vendors. Since FF is a relatively new fieldbus it hasn't the same market penetration and amount of installed nodes as more established fieldbuses. But in North America it has gained great acceptance. The Foundation protocol uses OSI layers 1, 2 and 7. Layers 2 and 7 are considered bundled together in a communication stack

CAN (Controller Area Network) is originally developed to simplify the wiring in automobiles. The cars security systems, such as ABS (Anti-lock Brake System), demand high accessibility and robustness of the fieldbus. The many advantages of the bus have made the CAN bus popular in several other areas on automation with some modifications. The use of CAN in cars has lead to several manufactures and that a very large amount of nodes has been installed. CAN is meant to be used in small systems or subsystem with a communication speed up to 1 Mbit/s. The CAN bus can be up to 40 meters and have 32 shielded or 64

unshielded nodes at the highest speed. The data exchange is done like this: The node is trying to send its message as quickly as possible, it doesn't wait for "permission" or its turn. All messages are therefore sent by need basis. If several nodes try to send at the same time, the message with the highest priority is sent and the others have to wait until the bus is available again. The priority is settled when the bus is configured and it is not possible to change it dynamically.

DeviceNet is an open network based on the Can-Protocol. DeviceNet is like a software-layer with the CAN-protocol as a base. DeviceNet was developed by Allen-Bradley 1994 but is now an open fieldbus that is supervised by ODVA (Open DeviceNet Vendor Association). ODVA are handling the specification, support, marketing and testes of products to DeviceNet. A DeviceNet bus can be up to 500 meters with 64 shielded nodes. Every message the bus sends can be up to 8 bytes of data. DeviceNet can with fragmented messages send messages that are larger then 8 byte. The original message are divided into 8 byte parts and will then be sent one by one to be put together by the receiving node.

Interbus was one of the first fieldbuses to achieve widespread popularity. The development started 1987 by Phoenix Contact. They later on made it an open fieldbus. Interbus has a ring topology, where the connection goes from the master to the all slaves and then back to the master. Each slave nodes has two connectors, one that receive data and one that passes data onto the next slave and also repeat/amplify the data so the distance between two nodes can be up to 400 meters. The ring can be maximum 12.8 km with up to 256 nodes. Interbus doesn't send a lot of small messages instead the master send a large message where all the slaves have their own address area.

5 Bus - Simulation Communication

Now we are looking closer to the development of the device driver. We start by describing the general structure of the device driver and some physicals properties of the Profibus slave pc card. The linking technique being used is Sun Microsystems JNI. Therefore we give a quick background to the Java programming language and the fundamentals of the JNI, the references used are Gordon (1998), Holm (2001), Liang (1999) and Sun (2003-01-14). In the third section, the implementations of the device drivers are described. We end this chapter with a brief description of some alternatives to the JNI.

5.1 General Structure of the Device Driver

We are now at the inside of the computer and have to pass incoming and outgoing process data between the Profibus slave pc card and the simulation program. The Profibus slave pc card that we have chosen has an Application Programming Interface (API) implemented in the programming language C++, and since the device driver for the simulation program must be implemented in the Java programming language, we have to establish some kind of communication link between these two programming language. This is a common situation because legacy native code such as C, C++ and assembly are widely spread in the industry today, and will coexist together with Java code for many years to come. The situation for our device driver can be described with the picture in Figure 10.

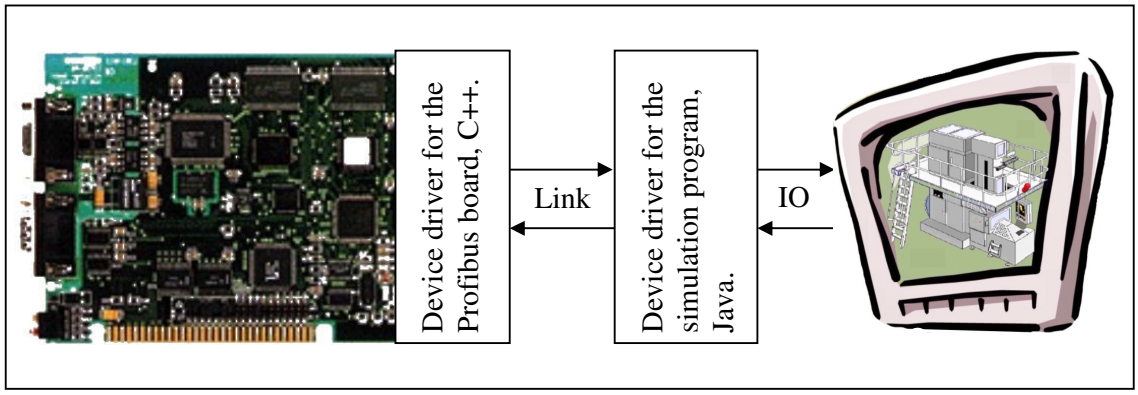


Figure 10. The structure of the device driver.

The process data transfers via the Profibus cable and are stored on the dual-port memory of the card, this is the physical interface to the communication board. To move the process data further into an application, one has to implement a device driver for the board, which is done in C++. Now we have all the process data accessible in our device driver for the board and to access the data from our device driver for our simulation program written in Java we have to use a programming technique called Java Native Interface (JNI). All these steps will be thoroughly explained in the following sections.

In our system, the PLC is considered as the master and the simulation as the slave. We used the Device Driver Manual to develop and implement the device driver. This manual came with the card from Hilscher. This manual describes the application programming interface for the communication board. The API supports different platforms DOS, Windows 3.xx/9x/NT and 2000. In our computer we have Windows 2000 installed so we will communicate between the application and the driver by a Dynamic Link Library (DLL) file.

There are two types of data transfer on the card, message oriented data transfer and process images data exchange. In our device driver we will use the process image data exchange because it is an I/O based protocol, which suits our communication well. The user interface communicates with the CIF via the dual port memory, which has two parts, a protocol dependent and a protocol independent part. The protocol independent part is the main part of the data between host and device. The protocol dependent parts are the parameters for initialising the protocol and the message structure for exchanging jobs between the host and the device, se Figure 11.

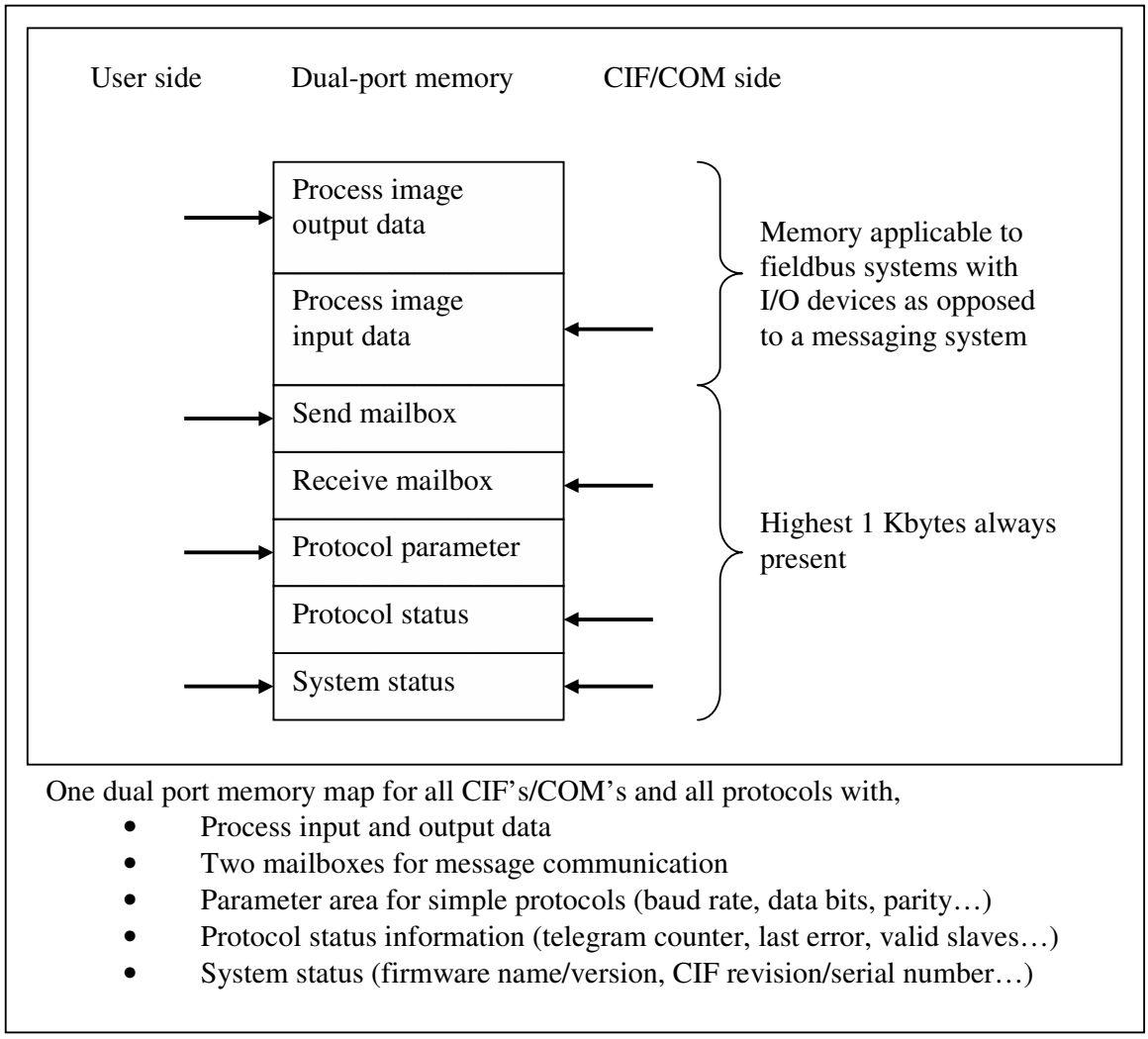


Figure 11. The Profibus slave pc card's dual-port memory.

5.2 Java Native Interface

When the company Sun Microsystems back in 1996 launched the first version of the Java programming language they wanted something that was safer to use than C++ and more portable between different platforms. So they came up with the Java platform that offers a set of features that an application can rely on, independent of the underlying host environments such as operating system, set of libraries and CPU instruction set. The Java platform is the programming environment that consists of the Java Virtual Machine (JVM) and the Java application programming interface, and it is commonly deployed on top of a host environment.

When Java is compiled, it is compiled into byte code and not into host-specific binary code as native code is. When Java byte code is executed it needs an interpreter between the byte code and the host specific binary code. Native code does not need this because it is already compiled into executable host specific binary code. This interpreting is what JVM does and this is why Java is portable between different computers, se Figure 12. In this thesis work we use Sun Microsystems Java Developing Kit (JDK) that consists of compiler, debugger, JVM etc.

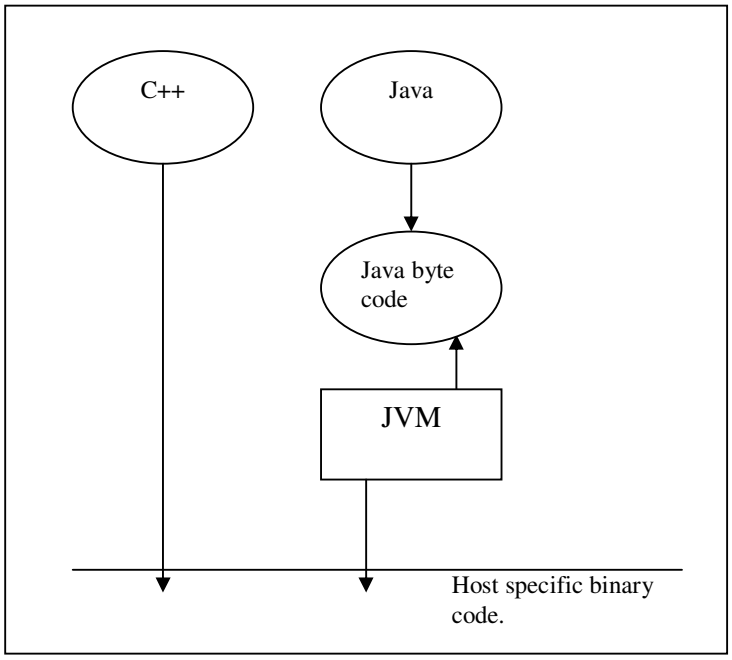


Figure 12. Compilation of the different source codes.

When Sun Microsystem developed the Java programming language they did not want to standardize the implementation of the JVM, in order to allow vendors to make their own implementation. As a result, even if the JVM implementation is different the layer they use to access the native methods is always the same. This is no limitation of the platform independencies because every implementation of a JVM must still comply with certain standards, such as the structure of a .class file. A class can be executed on any implementation. Any implementation of the Java platform is guaranteed to support the Java

programming language. Different JVM interpret in the same way, which is why it is platform independent.

When Sun Microsystems developed the Java programming language they realized that even if the Java language would be a better choice to work with, they also knew that management would not allow programmers to rewrite legacy native code that they knew were well tested and well working, just in order to have all code in one language. So some mechanism must be integrated in the JDK in order for the Java applications to interoperate with legacy native code. This mechanism is the Java Native Interface (JNI), and by writing the code using JNI one can ensure that the code is completely portable across all platforms. The only restriction is that it has to be recompile it when the platform is changed. JNI is simply used as an intermediate layer between Java class and native library or native applications, and because of this it is possible to address interoperability issues, and expect their solution to work with all implementation of the Java platform.

We will in this thesis work use native library, where we can use JNI to write native methods that allows Java application to call methods implemented in a native library. These calls for native methods in Java application will be performed in the same way as for Java methods. But behind the scenes however, native methods are implemented in a native language and reside in a native library. The structure is shown in Figure 13. JNI also supports an invocation interface that allows you to embed a JVM implementation into a native application. This allows you to perform existing applications Java enable without having to link with source code. However we will not use native application in this thesis work, therefore we will not go any deeper into the native application solution.

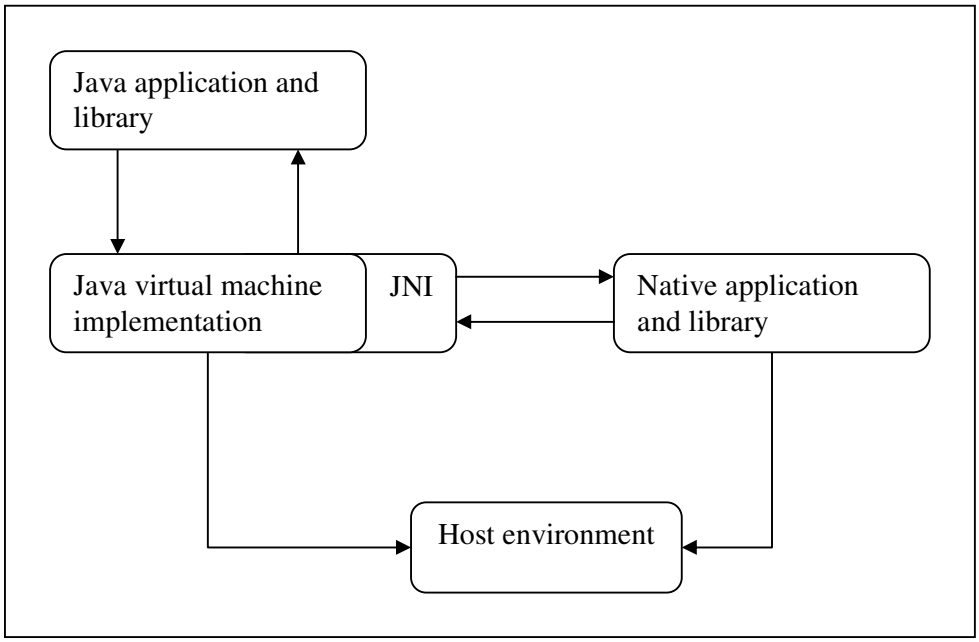


Figure 13. The structure of JNI.

One issue to beware of is that once an application uses JNI it risks losing two benefits of the Java platform. The first thing is that you must recompile your code if you change the host environment, so it will be less portable. The second thing is that you lose the security because

a misbehaved native code can crash the entire application. As a general rule you should architect the application so that native methods are defined in as few classes as possible. This entails a cleaner isolation between native code and the rest of the application.

JNI handles those situations when the whole application cannot be written entirely in Java programming language. JNI allows Java application to invoke native code and the other way around, because it is a two-way interface. The JNI framework lets your native method utilize Java objects in the same way that Java code uses these objects. Java application call native methods in the same way they call methods implemented in the Java programming language. However these methods are implemented in another language and reside in a native library.

When using JNI the native methods can create own objects including arrays and strings and then pass them to the Java application. The native method can also inspect and update objects, arrays and strings that were created by the Java application. Thus both side can create, update and access Java objects and then share these objects between them.

When a native method is implemented it accepts two standard parameters. The first is the JNIEnv interface pointer that points to another pointer, which in turn points to an array of pointers to JNI functions. The second standard parameter is the jobject, which is a parameter that is a reference to the object itself, se Figure 14.

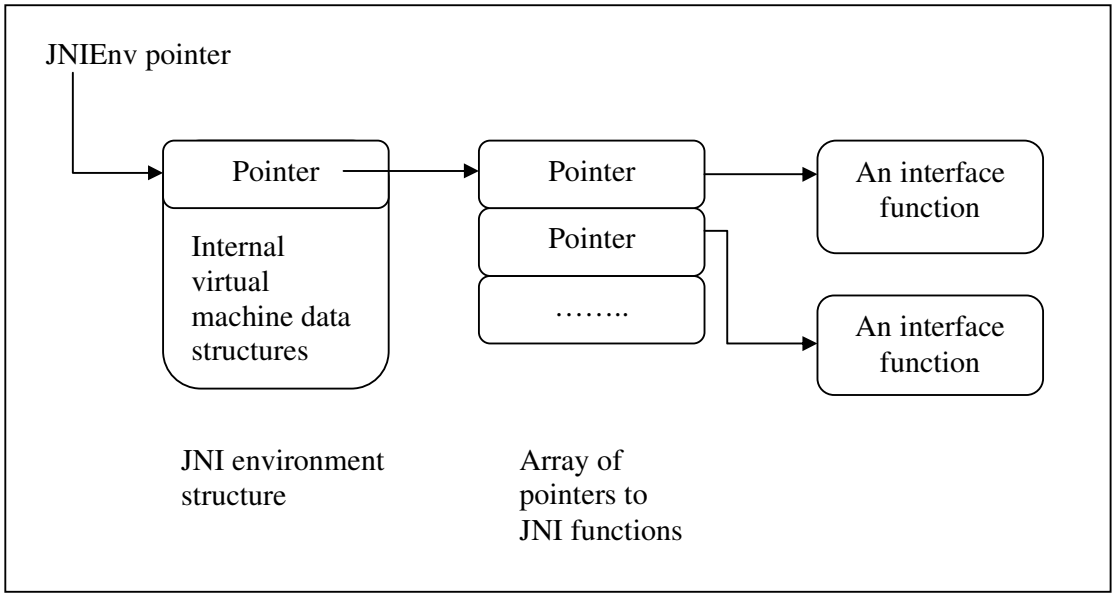
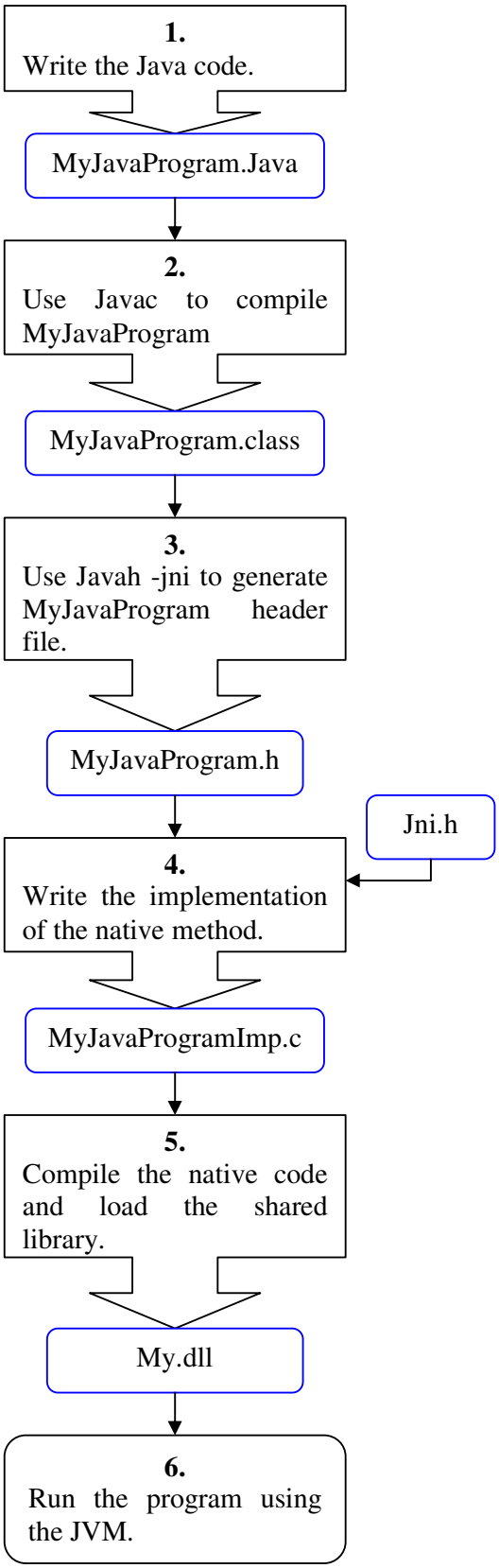


Figure 14. Shows the relationship among the JNI pointers.

To show how JNI is implemented we will demonstrate this by taking you through the necessary steps to integrate native code with programs written in Java, Sun (2003-01-14). Here we implement the MyJavaProgram which have one Java class called MyJavaProgram. It declares a native method and it implements the main method for the overall program. The implementation for the native method is provided in C, which is of no interest here. The file jni.h is a header file that provides information that the native language code requires to interact with the Java runtime system, must always be included.



Step 1. Create a Java class named MyJavaProgram that declares a native method. This class also includes a main method that creates a MyJavaProgram object and calls the native method.

Step 2. To compile the Java code that you wrote in step 1, use the command Javac.

Step 3. To create a JNI-style header file from the MyJavaProgram class, use the command Javah. The header file provides a function signature for the implementation of the native method.

Step 4. Write the implementation for the native method in a native language source file. The implementation will be a regular function that's integrated with your Java class.

Step 5. Use the c compiler to compile the .h file and the .c file that you created in step 3 and 4 into a shared library. In windows a shared library is called a dynamically loaded library (DLL).

Step 6. Finally use the JVM to run the program.

5.3 Implementation of the Device Drivers

We have now reached a point where we have to decide what the device driver has to do. One goal is that the device driver shall be designed so that other simulations can reuse it. Therefore the only action performed by the device driver will be to pass process data between the Profibus board and the simulation program. We started with installing the two programs, Microsoft Visual C++ 6.0 and Sun ONE Studio 4 CE, in which all the implementation were made.

On the Java side the device driver must be placed in the DeviceDriver folder in the path, Program\Lumeo Software\Multiphysics\Bin\DeviceDriver. Together with the device driver that you implement you must have a .cfg file that provides the simulation program with a unique identification number for the device driver. The device driver has to be a part of the package DeviceDrivers and is a subclass of ControllInterfaceDeviceDriver. On the C++ side we have to ensure that the program can find all the included files, we do this by setting the environment variables in the operating system.

When the simulation program is executed it loads all the device drivers in the DeviceDriver folder, remember that these are implemented in Java. When our device driver starts it first executes the constructor where it tries to load the path to the ProfibusDPDriver.dll, which is the entry point for the C++ device driver. Next it calls the native method initializeNative(). This method is then executed in the C++ driver where the Profibus board is opened, initialised and reset, se Figure 15. After this we have an open communication link ready to use.

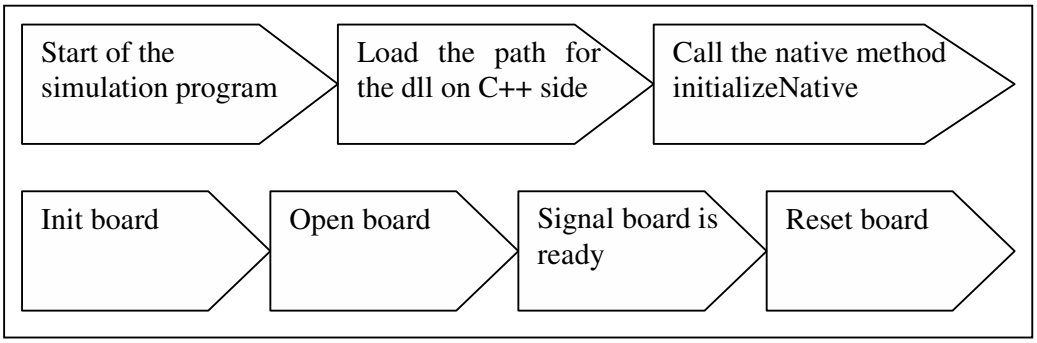


Figure 15. Initialisation of the board.

The updating of the simulation process data is controlled by the simulation tick variable, which is set in the simulation program. At every tick the simulation program calls the updateComponentValues() method in the Java device driver where it delivers all the outgoing data in the vector called value. After that it calls the native method xChangeData() method, which in one call both send and receive process data. This C++ function communicates with the CIFuser.dll file that handles the communication with the board. The last thing the updateComponentValues method does is calling the deliverHardwareInputs() method, this method delivers the received data to the simulation program, se Figure 16.

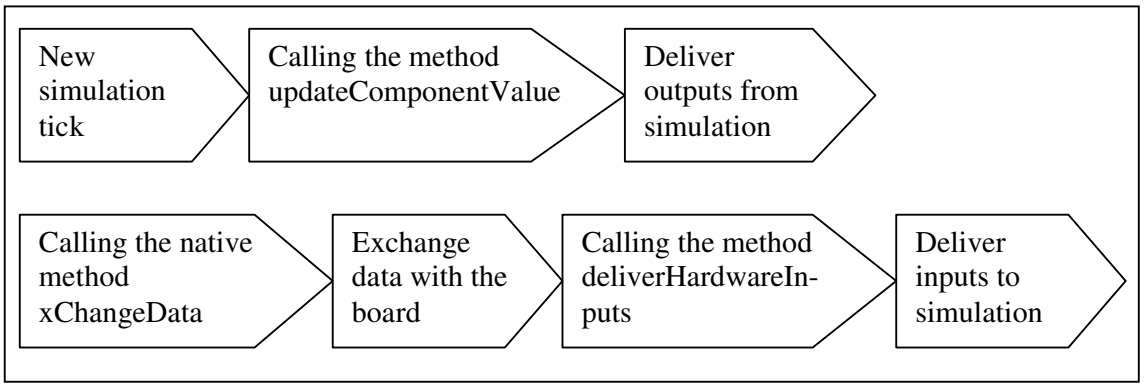


Figure 16. Exchange of process data.

Finally, when the simulation program is terminated the method exit() is called. This method calls the native method finalizeNative(), which close the communication with the board and the driver, se Figure 17.

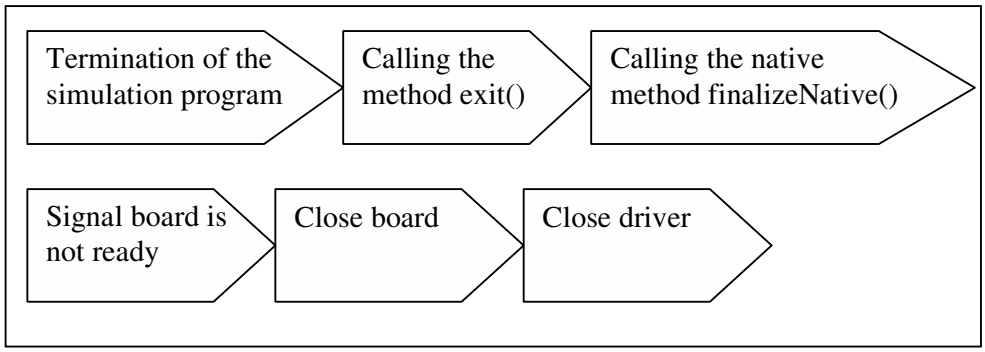


Figure 17. Methods called when closing simulation program.

5.4 Other Communication Links

An investigation reveals that there are some alternative ways to use, and we found following alternatives,

- Java IDE API, a distributed object technology
- JDBC API, a way of connect to a legacy database
- TCI/IP or other inter process communication mechanisms

The common denominator for the above mentioned alternatives are that they are used when the Java application and native code reside in different processes. Process separation offers an important benefit, the address space protection supported by processes enables a high degree of fault isolation, and a crashed application does not immediately terminate the Java application with which it communicates over TCP/IP. But since our Java applications communicate with native code that resides in the same process we will not use these alternatives.

Furthermore there are two more vendors that supply a native interface to Java, Microsoft's raw native interface (RNI) and J/DIRECT, and Netscape's Java Runtime Interface (JRI), Netscape (2003-03-31). If you look into JRI you will se that JNI has been evolved from JRI. It runs only under Netscape's own JVM.

RNI is a complex interface to native code but it allows you to write efficient code with a high degree of integration with the JVM internal operations. Apart from being less readable, RNI runs only under Microsoft's own JVM. The functions for controlling the JVM operations are directly available in RNI while in JNI the parameter JNIEnv is used. One big difference is how the garbage collector acts, in JNI it follows the same rules whether it executes native or Java code, while in RNI it is the programmer that are responsible for starting and stopping the garbage collector. You cannot call arbitrary DLL's, they have to be written especially for RNI. Even if RNI is fast and allows you to get full access to all of the fields in every object it is to complex for this thesis work. For more information about RNI se reference Jupitermedia (2003-02-10) and Microsoft (2003-03-31), b.

J/DIRECT is a complementary technology to RNI that is less complex and less powerful than RNI. J/DIRECT is much easier to use because it makes direct calls to unmodified DLL's without requiring an intermediate wrapper as RNI, this makes it easy to call the functions directly. It also makes manual type conversion unnecessary because it does it automatically. In comparison to JNI and RNI, J/DIRECT reduces the extra code and the number of DLL wrappers that need to be written. As for RNI J/Direct runs only under Microsoft's own JVM. The trade-off is that DLL functions cannot access fields and method of arbitrary Java objects. For further information about J/DIRECT se reference Microsoft (2003-03-31) a and b.

6 The Simulation Program

This chapter describes the programs that we use to solve the simulation task. The first part is about general features of the programs, what kinds of constraints you can add. In the later part, the simulation of the test rig is described and how the model can be controlled in the simulation mode.

6.1 General Features

The software from Lumeo Software is divided into two parts, Motion and Scenario. Lumeo Motion is an add-on to ProE that is used to add dynamics and constraints to an existing model that includes geometry, material, mass properties and initial forces. To add a feature or function to the model there is an additional menu, Simulation manager, in ProE with several options. The options are showed below in Figure 18.

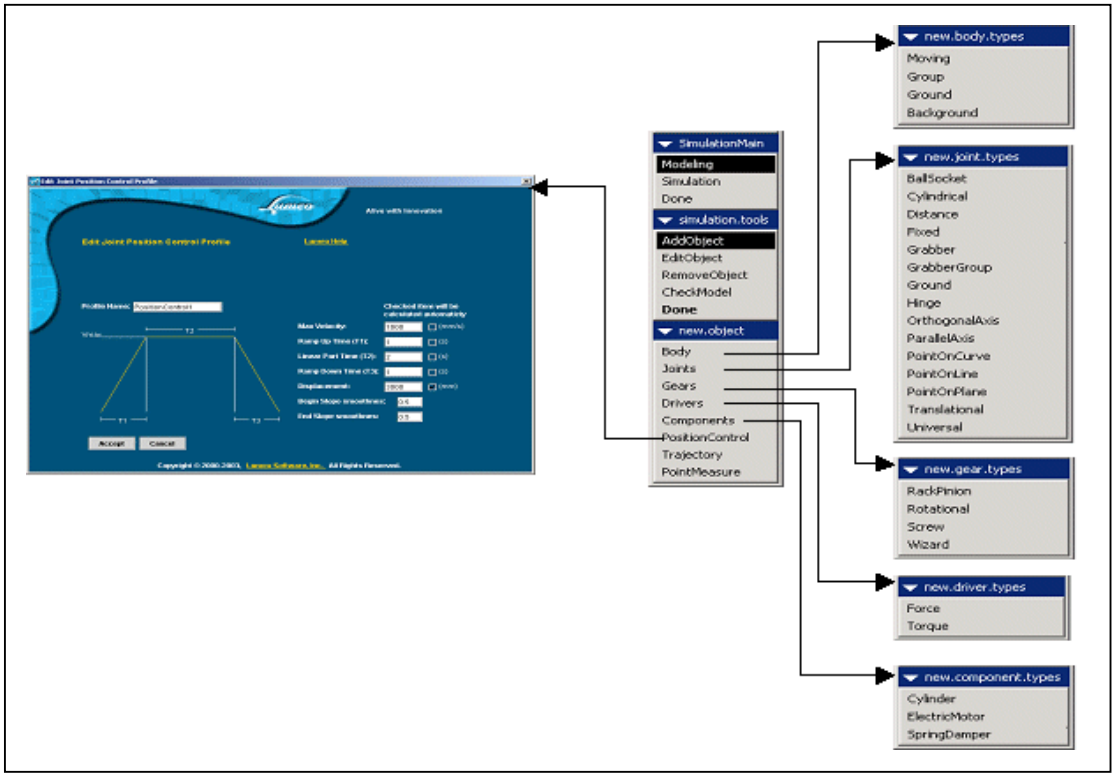


Figure 18. Lumeo Motions menu manager.

In the submenu Body you can decide which part or group of parts that shall act as ground or be able to perform some kind of movement. When you want to decide what kind of mutual relation two or more parts shall have, then there are some tools to use in the submenu Joints. The Gear submenu includes different kinds of gears to add to the model. Forces and torques

are added to the model from the Driver submenu, these drivers are normally proportional to the mass of the affected body. To create a simple ramp, use PositionControl, here you can modify the properties of the ramp that can be used for a translation. For the ramp you have to set a some values; maximum velocity, distance for the translation, the time for accelerate up to maximum velocity, the time to brake from maximum velocity and the time the part will be at maximum velocity. To indicate a path of movement for a body in three dimensions, you can use the Trajectory on the menu. When you want to supervise how a distance changes during simulation, you can use the PointMeasure on the menu.

Lumeo Scenario is a simulation player where you open the existing model created by Lumeo Motion. In the program it is possible to control the model manually with analog/digital controllers or with an external controller. In Figure 19 the main window of Lumeo Scenario is shown.

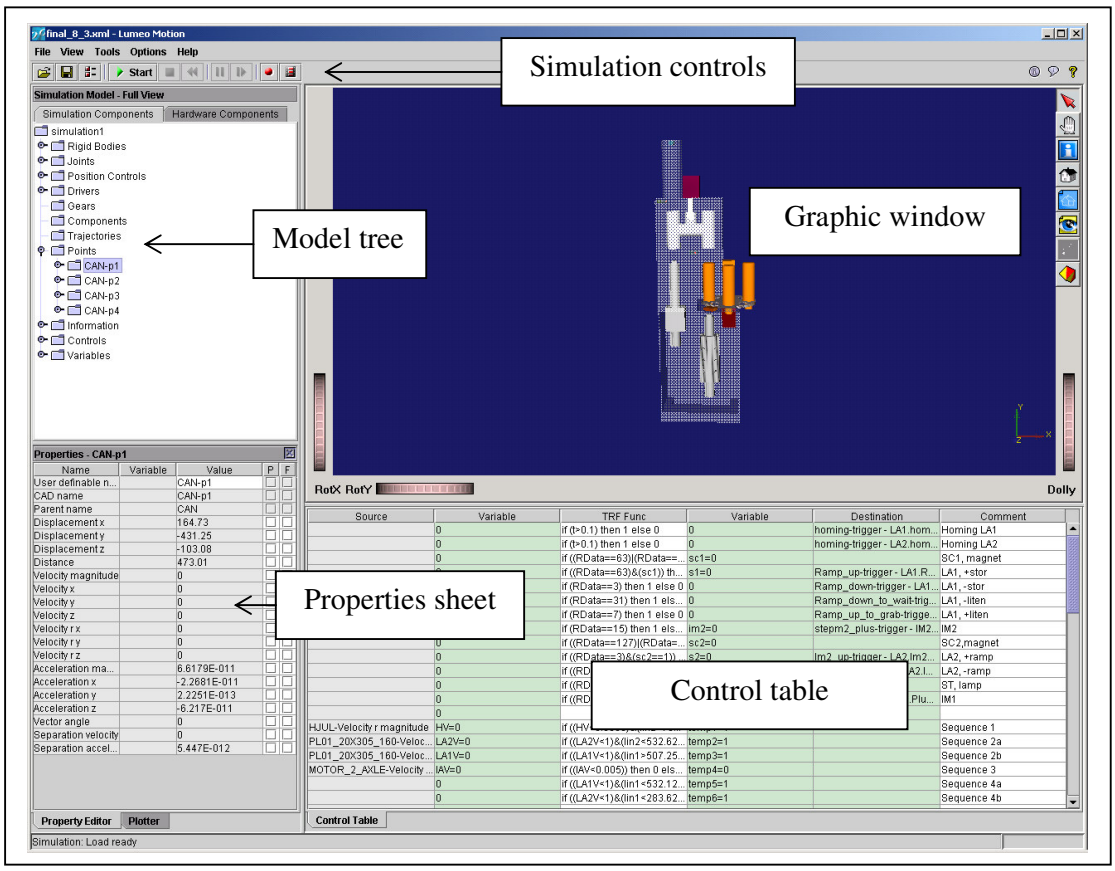


Figure 19. Lumeo Scenario main window.

In the upper right corner of the main window is the graphic window, where the model can be supervised during the simulation. To the left of the graphic window under the simulation components flap the Model tree can be seen which includes all the components that have been added to the model in Lumeo Motion. It is also possible to edit some of the values here, if a part is picked in the Model tree its properties such as velocity, acceleration or position are displayed below in the Properties sheet where values can be supervised or changed. If we change the flap to Hardware Components we get a list of available external device drivers that are placed in the folder DeviceDriver which can be found in the path Program\Lumeo

Software\Multiphysics\Bin, and we can pick anyone of our choice, such as our for the Profibus card.

To control the model the lower right window is used, the Control table. Here it is possible to insert a controller, by drag and drop from the Model tree or the Hardware component, as a source or destination. Starting from the left column we have the source for example the input signal from the Profibus card, then there is the variable column that can be used in the next column, transfer function (TRF Func), to modify or make special statements on the incoming value from the source. The second variable gets the value after the TRF Func, this variable can also be used in another TRF Func. You can use and create as many variables as you like in the Model tree or directly in the Control table. In the Destination field you put the component that you what to control, for example to start a motor or to set an output signal to a Profibus card.

In Scenario there are a couple of settings that have to be made for the simulation to work properly. First we have the simulation step size that is the duration of each simulation step. If we have an external controller that work as continues loop with a specific step time, the step size in Scenario have to be smaller so it doesn't miss a loop. It's also possible to choose Simulation tick mode instead. There we have two options, Best effort or Real-time. In Best effort the calculations are completed as fast as possible and in Real-time the calculations is done first and the updates of the graphics are made only if there are time left.

To speed up the simulation there are a setting that decide how often the graphical view should be updated. If the program is installed on a slow computer it could be good to set down the update rate and instead use the video capture function to look at the graphical view after the simulation is done.

6.2 The simulation of The Test Rig

We have a 3D-Cad model of the test rig in ProE that we use for the simulation task. This model was then modified in Lumeo Motion so that all the components that we needed for the simulation, for example grounds, motors and so on were added. To control the motors translations we needed to add some ramps that were equal to the real machines movements. For the packages to be able to make all the movements there had to be added a grabbergroup for each package so several different parts could grab it. The grabbergroup has one limitation; one part can only grab one position.

Then it's just to open the saved simulation file from Motion in Scenario. In Scenario we now have a new Hardware Component, since we made the device drive for the profibus card. From this we have one input and one output signal that we were going to use to control the model. These two components are dragged from the Hardware Component flap to the Control Table and are stored in the variables Rdata and SendData. These variables are used to control the simulation as condition for the different sequences. Since the LinMot Command Module has some special built-in features, we had to make the simulation program act as it. This was done with the Control Table. Here we added the input word from the profibus card as a source and with a complex transfer function we decoded the signal so it could be sent to the right destination.

The packages have to perform sex different movements in the test rig so we can say that there are sex sequences for the PLC-program to control. And after every sequence we need to send back a ready signal. Since there were no sensors or ramp done in Lumeo Scenario we had to imitate this with some workarounds that actually imitates the LinMot Command Module well.

For us to don't miss any messages we have to set the step size correct. Every device in this simulation has their own step time, showed bellow in Figure 20.

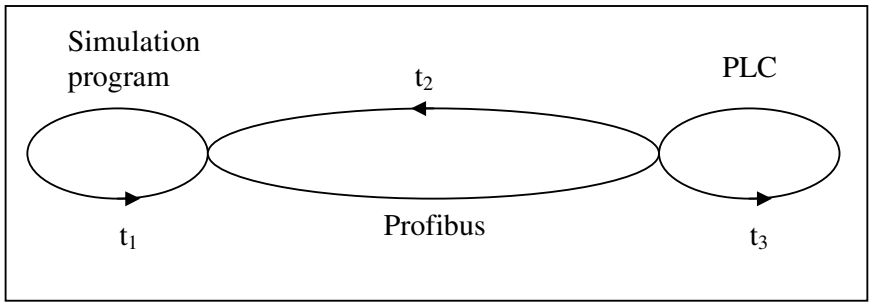


Figure 20. The time step size relations.

The profibus link is only going to transfer data as fast as possible so there we have the lowest step size. The PLC is going to send time critic data to the simulation program that it is not allowed to miss. This requires us to set the step size in Scenario to less then the PLC's step size. In other words $t_1 < t_3$ and $t_2 << t_1, t_3$.

7 Testing The Full Scale Implementation

In this chapter we describe the real world, starting with the machine, how it works and what it consists of. We then move on to the PLC program, how it is implemented and the protocol for the communication. The last part is a comparison between the simulation and the real machine.

7.1 The Test Rig

To have something to start the thesis work from, a simple test rig has been built. The test rig is a simplification of a machine that is under progress. Even though the test rig is simple, nevertheless it has been a good starting point for this thesis work because it is easy to grasp and has a faire degree of complexity. The test rig's assignment constitutes in moving packages in a continuous manner. The easiest way to describe how The test rig works and all of his components will probably be to follow one bottle through a whole sequence, the sequence diagram can be seen in Figure 21.

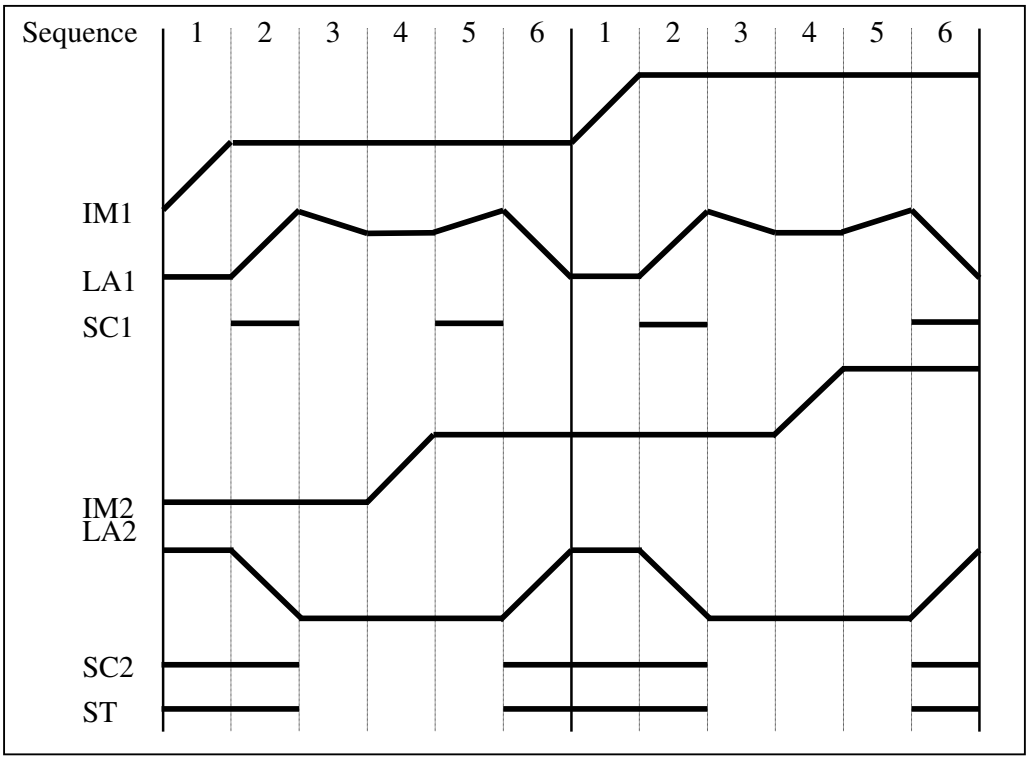


Figure 21. Sequences diagram for the test rig and simulation.

We start by assuming that the bottle we follow is in the start position, see Figure 23. The indexer motor wheel (IM1) turns one position forward, 90°, so that the linear actuator for the first lift (LA1) can grab the bottle with its suction cup (SC1), and at the same time start its upward movement. When the bottle has been properly placed in the chamber indexer sleeve,

the SC1 turns off and the LA1 makes a little downward movement and waits. Now when the bottle is in position, the indexer motor (IM2) makes a 180° turn with the bottle. The second suction cup (SC2) can now grab the bottle, and the liner actuator for the second lift (LA2) starts its upward movement towards the lamp. When the LA2 reaches its turn point, it waits for the next sequence and then starts a downward movement until it reaches the point of departure. The SC2 turns off, and the IM2 can make yet another 180° turn with the bottle, and the bottle is now in position for the LA1 that goes up and grabs the bottle with the SC1. The bottle moves finally all the way down back into the IM1, and is placed one position behind the previous one. To make the machine more efficient, there are always two bottles in motion, one on the way up and the other one on the way down. There are also two sensors for the homing sequence and two sensors that shall supervise the packages from getting jammed while they are in motion. The lamp (ST) at the top of the machine is indicating that the LA2 is in motion.

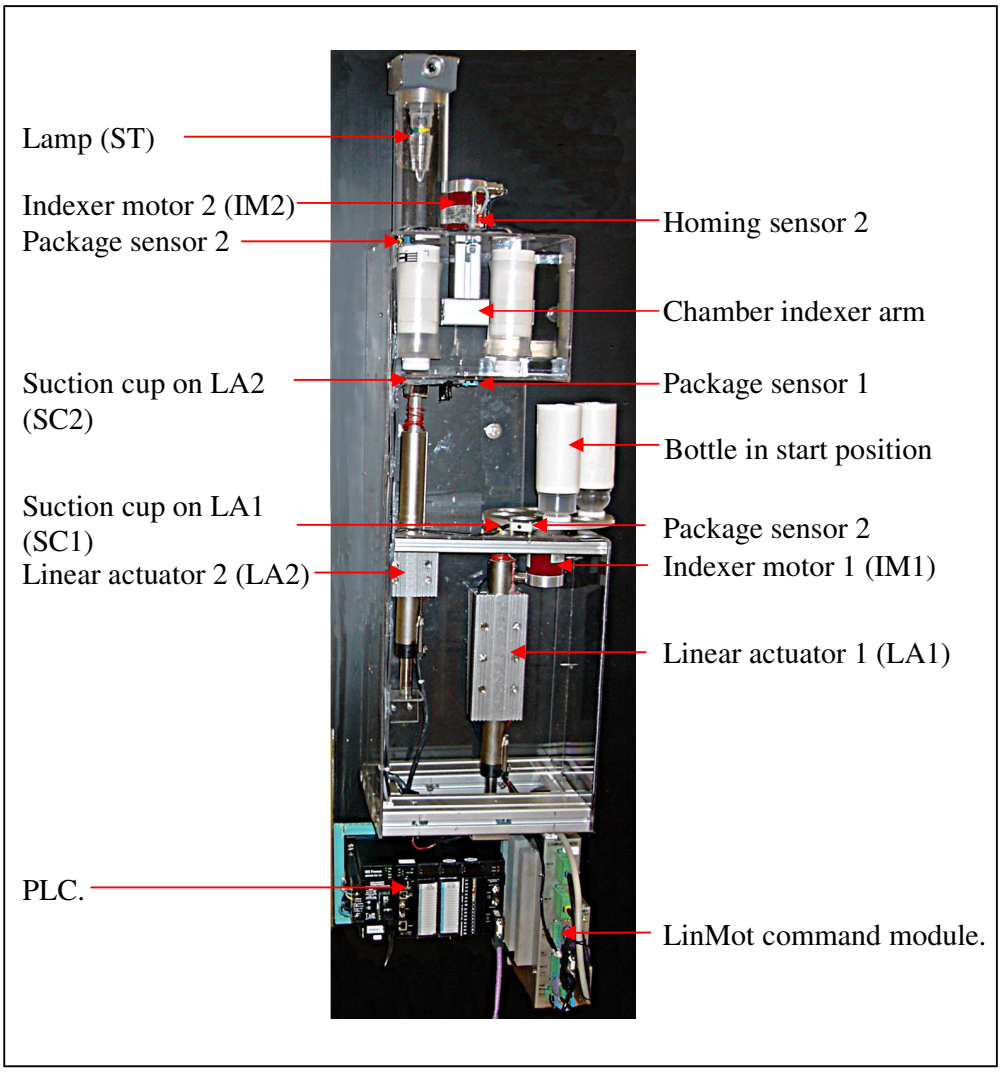


Figure 22. The test rig's components.

7.2 The PLC Program

The PLC program that we are using is not a complete program. It lacks an initialisation phase as well as a finalisation phase. What it handles is the sequences when all things are running normally. Two function blocks are used, Drum Sequencer and Move Word.

When the Move function receives power flow it copies data from one location in the PLC memory to another. In our case we copy C_S_in (IN), which are the signals from the simulation, to the Step1C (Q) word. The Step1C word is use to tell the PLC that a sequence is done, the lowest bit represents sequence 1, bit number two represents sequence number two and so on.

The second function is the Drum Sequencer, a program instruction that steps through a set of predefined output bit patterns stored in the word variable PatInput (PTN). The output bit pattern is selected based on the inputs to the function block, and the value is copied to a group of 16 discrete output references. When the Drum function receives a power flow on ResToStep1 (R) or NextStep (S) it copies the selected reference to the C_S_out (Q) reference. ResToStep1 is used to select a specific step in the PatInput, and the NextStep moves one step forward in the sequence when it makes an OFF to ON transition. Rungs 2-8 are used to generate these OFF to ON transition. Rungs 2-7 consist of two normally open contacts and one positive transition coil. The first contact is a reference to the actual step we are in at the moment, and are the same as the Follow (FF) operand in the Drum Sequencer. The second becomes true when the simulation sends a done sequence signal. When both contact are true the positive transition coil goes ON for one logic scan, and make rung 8 go OFF. This makes the Drum Sequencer move one step forward and copies the next bit patter to the C_S_out reference.

The bit patterns in the PatInput reference correspond to the protocol that has been decided as;

Word coming from the simulation.

Bit nr	Description
00	Step 1 complete
01	Step 2 complete
02	Step 3 complete
03	Step 4 complete
04	Step 5 complete
05	Step 6 complete
06	
07	
08	
09	
10	
11	Package sensor 1
12	Package sensor 2
13	Homing sensor 1
14	Homing sensor 2
15	!OK

Word going to the simulation.

Bit nr	Description
00	IM1, +90°
01	Jog IM1
02	LA1 run
03	LA1: 00 = + big, 01 = -small,
04	10 = + small, 11 = - big
05	SC1, on
06	
07	IM2, 180°
08	Jog IM2
09	
10	LA2, run
11	LA2: 1 = + ramp, 0 = - ramp
12	SC2, on
13	ST, on
14	
15	Ack event

7.3 A Comparison between Simulation and Reality

It is possible to simulate physical properties of the real Machine like gravity, mass, movements etc. In some sense one could even say that a simulation is too ideal, because there are no vibrations or any wear in the simulation. However, in our simulation of The test rig there is no major difference in the behaviour on the assumption that we only want to perform one whole sequence. This restriction due to that the grabber function can only grab one point per part. On the other hand there are some of the functions in the real Machine that the simulation lacks. For example, in the real Machine there is a possibility to use the package sensors to abort the program if a bottle is jammed during motion. This is not possible to do in Lumeo Scenario yet, the sensors do not work as in the real world and there is no complete collision function yet.

Unfortunately there is not a completed PLC program for The test rig. But the basic of the program will be the same; it will only be the mapping of the signals that differs. Because of this there has not been any possibility to do any comparisons between The test rig and the simulation. When there is a completed PLC program it would be easy to trim the ramps in the simulation so it moves as the real process.

During simulation the computer has to perform a large amount of calculations, therefore the real-time aspects are not relevant. However, Lumeo Scenario is able to record a video movie when it simulates, and this video movie can then be showed in real-time.

The homing sequence in the simulation are done only to get the simulation in the right position for the PLC program that we have, and not to obtain the right start position because the PLC program does not have an initialisation phase.

8 Conclutions

Here follows a short discussion and retrospect about the work that has been done so far and what goals that has been achieved up to now. We look into the future and tries to forecast what can be done to complete and improve this work.

8.1 Results

The most important result to point out in this thesis work, is that we have been able to verify that a machine simulated with Lumeos software can be controlled with an extern controller through a commonly used fieldbus. Furthermore, it was fairly easy to implement the LinMot command modules characteristics in the Lumeo Scenario. The drawback is that we have not been able to do a comparison between the simulation and The test rig due to the absence of a PLC program for The test rig, so no tests has been performed to see if the simulation can do any real-time performance. To get an apprehension of the real-time performance we have recorded a video of the simulation in Lumeo Scenario, this was done because of all the calculation made in Lumeo Scenario made the graphic updates were slow.

At the beginning Lumeo Software had a connection between a I/O block and the simulation via a TCP/IP link, Tetra Pak was not satisfied with this solution and wanted a more industrial solution. The answer to this is to use a field bus, and to connect the field bus device driver with the application JNI was used, an easy to use programming technique to combine an industrial field bus device driver with any Java application. With the JNI a simple design of our device driver could be accomplished. Although the final result was a success, there were a few struggles along the way. To start with we got contradictory information about if it is possible to use a Master card as a Slave card or not, as it turned out we could not get it to work, so a Slave card had to be purchased. Then there where some difficulties when the card should be installed such as address conflict, what interrupt mode to use and that all alarms had to be turned off for some inexplicable reason.

Initially, the goal was to use one and same program for The test rig and the simulation, but because of the differences in the GSD files this was not possible to achieve due to the differences in hardware design of the two cards. So two PLC programs had to be developed, and a protocol was established for the communication with the simulation. Since the simulation can be controlled in the running phase, there are no reasons for that it should not be able to be controlled in an initialisation- and finalisation phase, when this has been completed.

As the work proceeded we discovered that we needed more functions in Lumeo Scenario in order to complete the simulation of The test rig. For example the lamp, sensors, the grabbergroup that allows a part to be grabbed by more than one other part and the positioncontrol that is for creating simple ramps. This lead to a few updates of the software from Lumeo.

8.2 Future Developments and Improvements

To finish this project a few things has to be completed. The most obvious is perhaps that a faster computer is needed that can do faster calculations so that the updates of the graphics can be made with out delaying the movements in real-time. The computer we use today has a Pentium 2 processor with 300 MHz clock fervency.

A complete PLC program has to be develop with initialisation-, running- and finalisation-phases for both the test rig and the simulation. After that this has been accomplished a comparison test can be performed and evaluated.

To get a more complete simulation program some functionality ought to be developed such as a signal when a ramp is done, complete the collision functionality, more realistic sensors, make the grabber groups more easy to handle and perhaps introduce some kind of disturbances. This features will make the implementation more easy to grasp, another suggestion is that a more block oriented layout could be developed so one can work with the different phases separately.

To further examine how easy the Lumeo software is to use, additional features and function could be include to the simulation.

Instead of pas a whole word, perhaps let the device driver do a shift operation on the incoming data so that the simulation could test if a bit was set or not.

References

Gordon, Rob (1998). *Essential JNI: Java Native Interface*. Prentice Hall PTR

Holm, Per (2001). *Objektorienterad Programmering och Java.* Studentlitteratur.

Jupitermedia Corporation (2003-02-10). *Raw Native Interface (RNI).* http://www.codeguru.com/java/tij/tij0194.shtml.

Liang, Sheng (1999). *The Java Native Interface, Programmer's Guide and Specification*. Addison-Wesley.

Microsoft Corporation (2003-03-31), a. *Going Native with J/Direct*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnguion/html/msdn_drguinat.asp

Microsoft Corporation (2003-03-31), b. *Comparing J/Direct to Raw Native Interface*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vjcore98/html/vjconcomparingjdirecttorawnativeinterface.asp

Netscape, (2003-03-31). *The Java Runtime Interface*. http://wp.netscape.com/eng/jri

Nordbladh, Fredrik (2002). *Översikt av fältbussar och deras kommunikation med kringutrustning.* ISSN nr 0282-3772. Sycon Energikonsult AB.

Norén, A. (1999). *Simulating Logical Functions in a Filling Machine*. M. Sc. Thesis, TEIE-5140, IEA, Lund University.

Olsson, Gustaf (2002). *Industrial Automation, A Systems Approach.* IEA, Lund University.

Olsson, Gustaf and Rosen, Christian (2003). *Industrial automation-applications, structures and systems*. IEA, Lund University.

Sivtoft, J-O. (2002). *The Design of a Simulator for the Automation Industry*. M. Sc. Thesis, TEIE-5160, IEA, Lund University.

Sun Microsystems, inc. (2003-01-14). *Trail: Java Native Interface*. http://java.sun.com/docs/books/tutorial/native1.1

Profibus Trade Organization (2002-11-04). *Profibus Technology and Application*. http://www.profibus.com/imperia/md/content/pisc/technicaldescription/4002_vOctober2002-English.pdf

Abbreviations

API	Application Programming Interface
CAD	Computer Aided Design
CIF	Communication InterFace
CPU	Central Processing Unit
DLL	Dynamic Link Library
IDE	Integrated Device Electronics
J/DIRECT	Java Direct
JDK	Java Development Kit
JNI	Java Native Interface
JRI	Java Runtime Interface
JVM	Java Virtual Machine
PLC	Programmable Logic Controller
ProE	ProEngineer, a computer aided program
R&D	Research and Development
RNI	Raw Native Interface
TCI/IP	Communication protocol