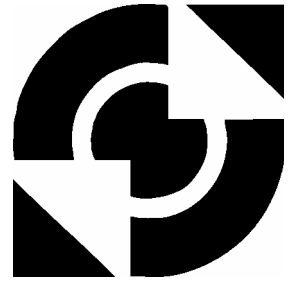


University of Twente

EEMCS / Electrical Engineering
Control Engineering



PC104 stack mechatronic control platform

Erik Buit

M.Sc. Thesis

Supervisors

prof.dr.ir. J. van Amerongen

dr.ir. J.F. Broenink

ir. P.M. Visser

March 2005

Report nr. 009CE2005

Control Engineering

EE-Math-CS

University of Twente

P.O. Box 217

7500 AE Enschede

The Netherlands

Summary

A part of the research at the control engineering laboratory of the University of Twente is on the design trajectory of embedded controllers. As a part of the research, tools are being designed for the design of heterogeneous distributed controllers. Once the controller is designed in a design tool like 20sim, it has to be tested and implemented. The testing and implementation phase is verified by simulation. For the verification by simulation phase a Hardware-In-the-Loop setup has been build in co-operation with the Boderc (Beyond the Ordinary: Design of Embedded Real-time Control) project at the Embedded Systems Institute. The setup contains four embedded PCs and two desktop PCs. The embedded PCs can run the controller(s) and the two desktop PCs can run a simulated version of the to be controlled plant. To get the designed controller running on the setup, as it was built in the previous project, knowledge is needed of the systems and of software engineering.

The aim of this project is to allow mechatronic engineers to use the setup without any knowledge of the software engineering process. To accomplish this, a method has been designed and developed that performs the software engineering part. The method is implemented in a set of tools. The tools allow easy connection of the hardware to the model, executable task generation, deployment of the generated task onto the setup, viewing and modifying values in any task online and offline, logging of values and automatic retrieval of the logged values.

For connecting the hardware to the model a hardware configuration file template and hardware connection tool have been designed and developed. The template file contains hardware specific calls to access the hardware and has to be filled once by a software engineer for every hardware device. After the template has been filled the hardware connection tool can use the template to connect model signals to the defined hardware.

On the embedded PCs and simulation PCs an, for this project designed and developed, application controls all the running tasks. A protocol to communicate to the application has also been designed and developed. The application has full control over the tasks started by the tools that are running on the stack. To start the controllers on multiple stations at the same time a synchronized start that uses the CAN bus has been implemented. Tasks are automatically generated and in case of a failure the user is notified.

The method is working and allows easy hardware connection. Task can be started and stopped and values can be retrieved and modified by a graphical user interface. Logging has been implemented into command prompt applications.

The hardware connection files are hard to write and are error prone. A tool that can assist in writing the files and do syntax checking is necessary to allow easier generation. The tools work on the same mechanism as the code generator of 20sim and could be implemented into 20sim to allow easy hardware connection from within 20sim. When using multiple controllers a synchronization mechanism between the controllers must be performed to prevent drift between the different controllers. The application running on the embedded and simulation PC has been written to run under Linux and use Ethernet to communicate. This application could be ported to other operating systems and the communication could be expanded with e.g. USB communication to allow the tools to run on multiple hardware architectures.

Samenvatting

Een deel van het onderzoek dat gedaan wordt bij de afdeling control engineering van de Universiteit Twente is naar het ontwikkel traject van ingebedde regelaars. Een deel van onderzoek bestaat uit het ontwikkelen van gereedschappen om heterogene gedistribueerde regelaars te kunnen ontwerpen. Als de regelaar is ontworpen in bijvoorbeeld 20sim, moet hij worden getest en geïmplementeerd. De test en implementatie fase wordt geverifieerd door middel van simulatie. Voor het verifiëren door middel van simulatie is een Hardware-In-de-Loop opstelling gebouwd in samenwerking met het Boderc (Beyond the Ordinary: Design of Embedded Real-time Control) project van het Embedded Systems Institute. De opstelling bestaat uit vier embedded PC's en twee simulatie PC's. de embedded PC's worden gebruikt voor het uitvoeren van de regelaar teken en de simulatie PC's worden gebruikt om het te besturen proces na te bootsen. Om een regelaar op de opstelling, zoals deze is gebouwd in een vorig project, te laten draaien, is er kennis nodig van software ontwikkeling.

Het doel van dit project is de mechatronische ontwerper in staat te stellen de opstelling te gebruiken zonder kennis van software ontwikkelingsprocessen. Om dit te bereiken is een methode ontworpen en ontwikkeld, die het software ontwikkeling proces uitvoert. De methode is geïmplementeerd in een set gereedschappen. De gereedschappen maken het mogelijk hardware aan een model te koppelen, een uitvoerbare taak te genereren, een taak op de opstelling uit te laten voeren, waarden te bekijken en aan te passen in elke taak verbonden en niet verbonden, opslaan van waarden en automatisch extraheren van de opgeslagen waarden.

Om hardware aan een model te koppelen is een hardware configuratie bestand en een hardware koppel gereedschap ontworpen en ontwikkeld. Het configuratie bestand bevat hardware specifieke functies die nodig zijn om de hardware aan te sturen. Voor elke hardware moet één maal een configuratie bestand worden gevuld door een software ontwikkelaar. Nadat het configuratie bestand is aangemaakt, kan het hardware koppel gereedschap de configuratie gebruiken en elk signaal van een model koppelen aan de, in het hardware configuratie bestand, beschreven hardware.

Op de embedded en simulatie PC's draait een, voor dit project ontworpen en ontwikkeld, programma dat alle draaiende taken beheert. Een protocol om communicatie mogelijk te maken tussen met het programma is ook ontworpen en ontwikkeld. Het programma heeft volledige controle over de taken die door de gereedschappen zijn opgestart en draaien. Om regelaars op meerdere stations op hetzelfde moment te starten is een gesynchroniseerde start ontworpen die gebruik maakt van de CAN verbinding van de opstelling. Taken kunnen automatisch worden gegenereerd en op het moment dat een fout optreedt wordt de gebruiker geïnformeerd.

De methode werkt en maakt het mogelijk om eenvoudig hardware te koppelen. Taken kunnen worden gestart en gestopt en waarden kunnen opgehaald en geschreven worden door middel van een grafische interface. Het opslaan van waarden is geïmplementeerd in opdracht regel programma's.

De hardware configuratie bestanden zijn lastig te schrijven en erg fout gevoelig. Een gereedschap dat kan assisteren en de syntaxis kan controleren is nodig om dit te vereenvoudigen. De gereedschappen werken op dezelfde manier waarmee 20sim code genereert en kan daardoor in 20sim geïntegreerd worden om hardware koppelingen vanuit 20sim mogelijk te maken. Wanneer er meerdere regelaars worden gebruikt moet er een synchronisatie methode worden ontwikkeld om drift tussen de regelaars tegen te gaan. Het programma dat op de embedded PC's draait is gemaakt voor Linux en gebruikt Ethernet om te communiceren. Het programma kan omgezet worden naar andere architecturen en de communicatie zou uitgebreid kunnen worden met bijvoorbeeld USB communicatie om op meerdere architecturen te kunnen draaien.

Preface

After a one and a half year career of fork truck driving I understood that loading trucks was not the thing I wanted to do for the rest of my life. I went back to school and finished the MTS and HTS. After the HTS I knew how to get things done by pushing the right buttons but not why it worked. At the University I wanted to learn more about the whys behind the hows. With this report, I finish my Master of Science study in the field of Electrical Engineering at the University of Twente and understand a lot more of the whys behind the hows.

I would like to thank Marcel Groothuis, Peter van den Bosch and Olaf van Zandwijk for testing the tools and giving valuable feedback.

Most of all, I would like to thank my girlfriend Alie Poelstra for standing by me all those years.

Erik Buit

Hasselt, March 2005

Contents

1	Introduction.....	1
1.1	Goal of research	1
1.2	Design trajectory of an embedded controller	1
1.3	Implementation of an embedded controller.....	2
1.4	Hardware in the loop simulation	3
1.4.1	Purpose of HIL.....	3
1.4.2	Hardware-in-the-loop Simulation used.....	3
1.5	Implementation difficulties	3
1.6	Test setup	4
1.6.1	Limitations of the setup	6
1.6.2	Setup after modification.....	6
1.7	Used Tools	7
1.8	Outline of the report.....	7
2	MSC tools overview.....	9
2.1	Purpose.....	9
2.2	Tools.....	10
2.3	Extensible Markup Language (XML).....	10
2.4	Architecture of the tool chain.....	11
2.5	Generalization of hardware	13
2.5.1	Problem definition	13
2.5.2	Current approach.....	13
2.5.3	General hardware procedure	14
2.5.4	Code generation process and generated application	15
2.5.5	Possible solutions.....	15
2.6	Conclusion.....	16
3	MSc tools implementation	17
3.1	Hardware connector	17
3.1.1	Graphical user interface	17
3.1.2	Implementation of the hardware connector	18
3.2	Compiler assistant	19
3.3	Deployment manager	19
3.4	Conclusions	21
4	Command and Control Environment	23
4.1	Purpose.....	23
4.2	Interfaces.....	23
4.3	Synchronized start.....	23
4.3.1	Investigated options	24
4.3.2	Detailed descriptions of synchronized start options	25
4.3.3	Conclusion	27
4.4	Command line programs	27
4.4.1	Sending a sync pulse (<i>sendsync</i>).....	28
4.4.2	Getting Values (<i>getval</i>).....	28
4.4.3	Manipulating values (<i>setval</i>).....	29
4.4.4	Automatic logging of values (<i>logval</i>)	29
4.4.5	Manual logging of values (<i>startlog, getlog</i>)	29
5	Embedded stack.....	31
5.1	Linux device driver standards in the Anything IO driver	31
5.2	Anything IO device driver.....	31
5.2.1	Init and cleanup.....	31
5.2.2	The /proc and /dev file systems	31

5.2.3	Read and write operations	32
5.2.4	Input and output control operations.....	32
5.3	LXRT extension of the driver	32
5.4	Stack daemon.....	33
5.4.1	Protocol.....	33
5.4.2	Implementation.....	35
6	Demo setups.....	39
6.1	Hardware.....	39
6.2	Software	39
6.3	Direct link	39
6.3.1	Purpose	39
6.3.2	Implementation.....	40
6.4	Results.....	41
6.4.1	Simulation versus real-time HIL	42
6.4.2	Simulation versus real plant	42
7	Conclusions and recommendations.....	45
7.1	Conclusions.....	45
7.2	Recommendations.....	45
	Appendix I Compiling Linux based sources on MS Windows	47
	Appendix II Building the root file system from scratch (the hard way)	48
	Appendix III Hardware configuration files	54
	Appendix IV Command and control DLL.....	57
	Appendix V Scite, Doxygen, wxDev-CPP	61
	Appendix VI Common pitfalls.....	64
	References	66

1 Introduction

At the Control Engineering chair of the department Electrical Engineering of University of Twente the main focus is the design of embedded controllers. For the design and simulation of controller- and plant models a software package called 20sim has been developed (CLP, 2002).

When the model of the controller has been designed and simulated in 20sim the next step in the design trajectory is to test the controller in the real world. In order to test the controller, a platform is needed that can steer the modelled plant. One of the platforms used for this step is a DSP board.

To get the controller from 20sim to the platform several successive steps are needed. First 20sim generates C code in order to export the functionality into a general format. The C code is then manually loaded into the DSP development environment. The code is compiled and deployed on the DSP. The DSP now operates as the controller. If modifications need to be made to the model, or one of the variables needs to be changed, the complete loop of modelling, simulation, code generation, compiling and deployment has to be repeated.

1.1 Goal of research

The goal of this research is to present a general method for deploying tasks on an embedded device like an embedded PC or DSP. The proposed method must be portable to other architectures and/or operating systems to be used for future setups. Furthermore, the method must simplify and automate the current process in such a manner that it is accessible for controller engineers who don't have knowledge about compiling and deploying.

A second goal is to abstract the model from the hardware implementation by designing a method where the hardware is connected to the model without any constraints to the model or hardware.

As a result of this research it must be possible to perform a task generated by this method for another, existing setup. The results of this task, i.e. a demo, will be compared to the results of a previous project (Groothuis, 2004) to validate the method.

1.2 Design trajectory of an embedded controller

Present-day requirements for reliable and efficiently extendable/updateable software for embedded systems, stress the availability of proper design software, assisting the complete design stretch. Especially, when *embedded control systems* are concerned, having the behaviour of the complete system available as dynamic model in the design tool is crucial for effective design work.

For an ECS, computational latency must be small compared to the time constants of the appliance. Examples are robots, production machines like wafer steppers, motor management and traction control of automobiles.

The embedded computer system is considered heterogeneous and distributed, because modern systems are often composed of existing subsystems, having their own control software and processors. Furthermore, systems must be easily *scalable* and *adaptable*, to support ever changing functional specifications and evolution of computer hardware.

With the developed method, the user (mechatronic engineer) is released from needing skills on programming the target hardware and interface devices (e.g. real-time behaviour and priority scheduling), and yet get results equivalent to those obtained by an experienced software design engineer. Furthermore, the transformation process to convert the algorithms for the control, i.e. mathematical formulae, to efficient computer code and possibly the concurrent implementation of the algorithms, is guided via a stepwise refinement procedure. After each refinement step, the results are verified by simulation.

The *main* objective of the method is to support the user (mechatronic engineer) at implementation of control systems, to eliminate design and coding errors and to diminish development costs.

This is accomplished by (Wijbrans, 1993):

- Verification and validation.
- Mechatronic design approach.

- Separate development of reusable parts.

As is quite common for contemporary Computer–Aided Control System Design software methods, *abstraction*, *partitioning* and *hierarchy* can reduce the complexity of a complete system. The system is partitioned into a hierarchical set of modules. During the design process, the level of detail will change: during control law design, the A/D converter is assumed to be ideal, while during the implementation phases, extra detail is added, to more precisely describe the behaviour.

The complete design trajectory of controllers comprises the following four parts (see Figure 1)(Broenink *et al.*, 1998)

- Physical Systems Modelling.
- Control law Design.
- Embedded System Implementation.
- Realization.

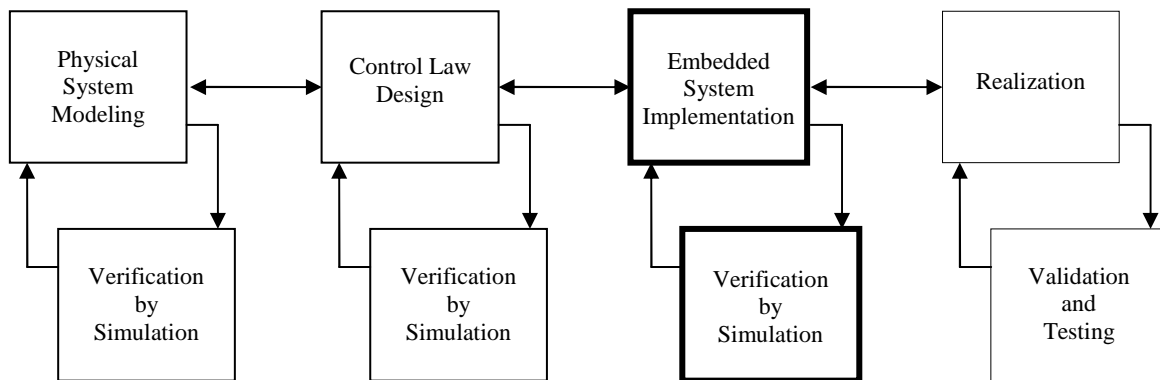


Figure 1: Design trajectory

After the control law(s) have been designed, they need to be implemented on the embedded computer. The starting point of this phase is that the control laws have been verified by simulation using the detailed model, assuming ideal devices for implementation: sensors, actuators and algorithms do *not* have any effects on the performance of the ECS.

1.3 Implementation of an embedded controller

To structure this implementation process the following procedure must be followed(Wijbrans *et al.*, 1993):

1. Integrate control laws and user requirements
2. Add technology–independent functionality
3. Add technology–dependent functionality
4. Add timing characteristics

This procedure does not prescribe the order in which the development must take place, but instead the designer has the freedom to tackle the individual sub problems in any order.

The implementation of the embedded system is a process of *stepwise refinement* from control laws to control algorithms, a specification from which computer code for the target processors (i.e. embedded computers) can be generated automatically and straightforwardly.

For the Boderc project a simulation method called *Hardware-in-the-Loop Simulation* is used to verify the embedded system implementation. This setup has been build and the next section will give an overview.

1.4 Hardware in the loop simulation

For the design, implementation and validation of control systems Hardware-In-the-Loop (HIL) simulation is increasingly being required, where some of the control-loop components are real hardware, and some are simulated. Usually, a process is simulated because it is not available (concurrent engineering), or because experiments with the real process are too costly or require too much time. The real-time requirements for such simulations depend on the time-scale of the process and the simulated components involved.

1.4.1 Purpose of HIL

HIL simulators allow to test and validate the *real* embedded control system (ECS) under different workloads and conditions. Other simulation methods do not allow testing the real embedded control system as a complete system. Often the controller part, which is about only 20-30% of the ECS software (Pasetti and Pree 2001), or the software components are tested independently. HILS makes it possible to test the complete ECS system.

1.4.2 Hardware-in-the-loop Simulation used

The Hardware-In-the-Loop simulation (HILS) where the method of this project is developed for, involves connecting the actual ECS to a computing unit with a real-time simulation model of the plant. This is depicted in the middle situation of Figure 2.

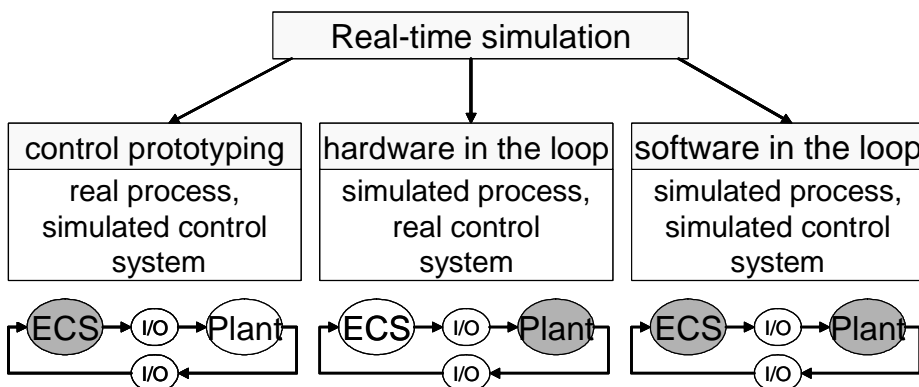


Figure 2: Kinds of real time simulation

The main advantages of HIL Simulation are:

- Plant models used during off-line design and simulation for the controller development can now be used for the ECS testing. This implies that, at software testing, the stubs representing the plant now can be proper models instead of simple signal generators. This results in better quality of the ECS tests, allowing for a less complicated integration phase.
- Software design and testing can be moved to an earlier design phase, i.e. before a first prototype is available, allowing concurrent engineering between the different design disciplines. This results in a shorter time to market.
- The ECS software changes can straightforwardly be checked for consistency with the design. Test software and test data written in the control design stage can be reused easily.

An additional benefit is that plant models used for off-line design and simulation during the control development can be reused for the ECS testing.

1.5 Implementation difficulties

The *Physical Systems Modelling* and *Control law Design* are fully supported by 20sim. The step toward *Embedded System Implementation* is still a large one.

Embedded system implementation starts by creating 20sim-templates. The templates contain a platform specific task framework with predefined tokens that are replaced by 20sim with model specific functions. The current version of the 20sim code generator has an option to generate function calls defined as DLL function calls in the simulation into the code. A disadvantage is that the DLL function call in the model has to be changed if other hardware is used. This makes the model hardware dependent which disagrees with the idea that a model must be hardware independent.

After the code is generated from templates, it has to be converted to a task that can be executed on the target device. This conversion process can be automated by scripts that are called from 20sim. The next step that has to be performed is the compiling and linking of the sources. Because every target has its own specific compiler and compiler options there is no straightforward way in generating executable tasks.

After compiling the task it needs to be deployed on the target. This could be performed by scripts, but if more devices are used or a different setup is used, the scripts must be altered to that specific setup.

In order to validate the results of a task, the deployed task needs a means to report the results. This could be implemented into the model, but again the model is depending on the hardware.

The methods proposed in this thesis pose a solution to overcome the given implementation difficulties. Furthermore, a tool chain is implemented as proof of concept.

1.6 Test setup

The HIL setup that is used was build in a previous project (Groothuis, 2004) and looks like Figure 3.



Figure 3: Setup

The HIL setup contains two types of PCs, simulation PCs and embedded PCs. The development PC running the modelling software is not a part of the setup and can be located anywhere where an internet connection to the stack can be realised. For this project a set of libraries have been build that run on the development PC that make command and control of the setup possible from the development PC.

The PCs are connected by three types of media as shown in Figure 4:

- I/O for simulation signal transfer between the setup PCs
- CAN for synchronization
- Ethernet for control command

The difference between the types of PCs and the FPGA will be explained next.

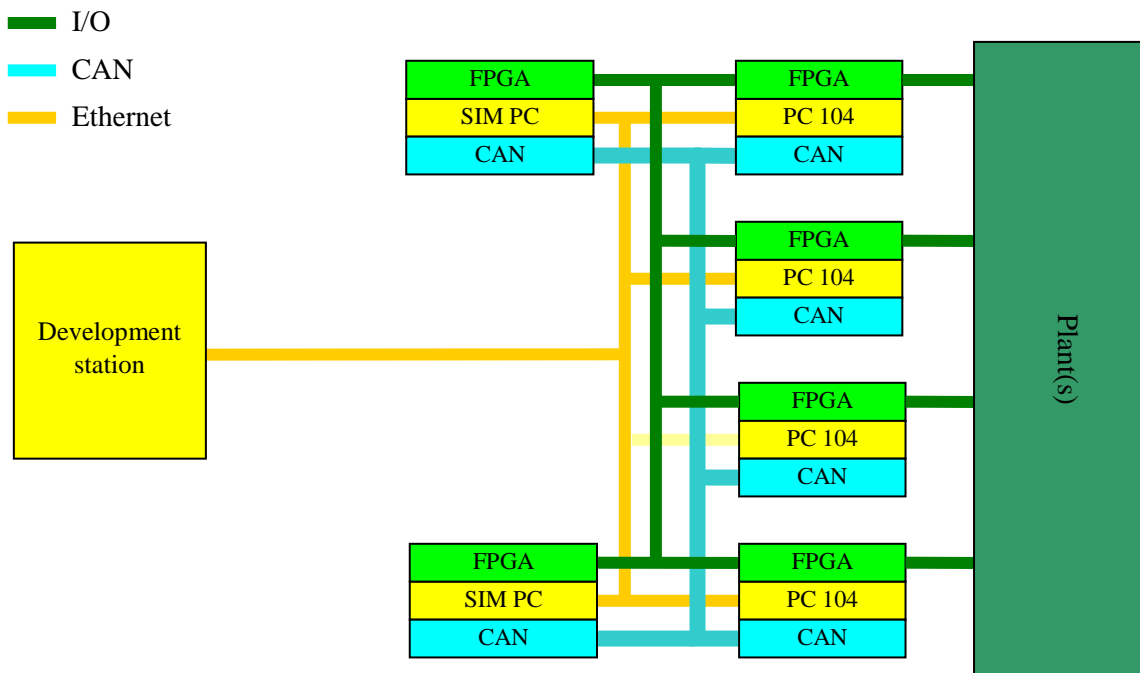


Figure 4: overview of the complete setup

Development station:

The *development station* is used to design and simulate the model in development phase. The development station can be any kind of PC running windows because the current version of libraries are made for Windows and 20sim can currently only run on Windows. The station needs to have an Ethernet connection that is connected to the PC104 boards (Mechatronic stack) and the SIM PC (Simulation PC) as shown in Figure 4. The development station is used to control and monitor all the other PCs.

Mechatronic Stack:

A *mechatronic stack* is a set of boards that together can interact with a mechatronic plant or simulation PC. In other literature it is sometimes defined as *embedded control system* (ECS) (Sanvido, 2002) (Jovanovic *et al.*, 2001). An ECS is a system that is included in the bigger system and can not be seen from the outside world as being such. ECS is, for this project, a too general description and therefore not used. A mechatronic stack has only one goal and that goal is to control a plant by sending signals to the plant and reading signals back from that plant.

The mechatronic stack used consists of three boards:

- CPU board
- CAN controller board
- Programmable IO board

All boards of the mechatronic stack are in the PC104 form factor and therefore called PC104. The CPU board has a VIA Eden fanless CPU that is designed to be used in embedded applications.

Simulation PC:

A *simulation PC* is a common of the shelf (COTS) desktop PC extended with project specific hardware to be used in the experiments. The PC is extended with the following hardware:

- CAN controller board
- Programmable IO board

The simulation PCs run the same version of embedded operating system as the embedded stacks. An advantage of this approach is that from the development station view there is no difference between

the stacks and the simulation PCs. The simulation PCs are used to simulate a real plant that was modelled in 20sim.

FPGA:

The programmable IO boards (Mesa Electronics, 2004) on the stacks and simulation PCs contain a field programmable gate array (FPGA). An FPGA can be programmed by loading a data file into the FPGA [Xilinx]. Both the simulation PCs and the embedded stacks contain an FPGA board. Signal exchange between simulation PCs and embedded stations will go through these FPGA boards in real-time. Because the FPGA can be programmed, the communication between the simulation PCs and the embedded stack can be in any digital format. This allows the user to simulate not only the plant and the controllers but also the type of signals, like pulse width modulation.

1.6.1 Limitations of the setup

The implementation that was used in the previous project (Groothuis, 2004) consisted of a number of scripts that were executed in sequence. These scripts compiled the C code generated by 20sim and uploaded the executables to the stacks. For execution of the tasks a command on the stack itself had to be given to start the experiment. At the end of the experiment a file with values was stored on the embedded stack. This file could be retrieved from the stack and compared to the simulated values.

This approach is sufficient if the all parameters are known in advance and the modeller has knowledge of the internals of the stacks. If another plant was to be tested, the sources of the templates had to be modified to connect the hardware to the software.

There are some other drawbacks to this approach. If, for example, the compile stage of the process generates an error, the sources have to be compiled by hand in order to find any faults because the scripts stop without warning if a fault occurs. The user is not notified of what internally happened and generated the faulty instruction. If the templates compile without faults, the new model can be generated and tested. This process is very labour intensive and error prone.

A more general approach of the process is the research of this project. The fault sensitive or labour intensive parts of the previous approach are:

- Connecting hardware to the software
- Compiling the sources
- Uploading the executables
- Starting the experiments (synchronously)
- Downloading the results

The commands needed for rapid prototyping that are impossible with the approach are:

- Connection hardware to software independent of hardware and model
- Changing variables online
- Using the same approach on different targets

Chapter 2 gives new insights on how to get rid of the limitations and add more functionality to the HIL setup. The next section is an overview of what is accomplished with the methods defined in chapter 2.

1.6.2 Setup after modification

The methods designed for this project allows full control over the complete setup. Graphical user interfaces have been developed for the following part of the deployment process:

- Model independent hardware selection
- Connection of the hardware to the software
- Compiling the source code
- Uploading hardware specific configurations
- Uploading embedded tasks
- Starting the embedded tasks
- Deleting embedded tasks
- Viewing and modifying variables

The following command and control actions have been implemented into the protocol and can be accessed by command line applications:

- Starting the experiments embedded tasks synchronously (to be used in distributed controllers)
- Start and stop logging of variables
- Retrieving logged results

1.7 Used Tools

A variety of tools is used to develop the tool chain. For this project a few tools have been tested on usability. A small overview of the tools is given here a more extensive overview is given in Appendix V Scite, Doxygen, wxDev-CPP.

wxDevCPP

DevCPP (Bloodshed, 2004) is an open source development environment that can be used cross platform. wxDevCPP (Kathiresan, 2005) is the windows version of DevCPP extended with options to easily create graphical user interfaces. A big advantage is that when building applications for Windows and only wxWidges libraries are used, the applications can be easily without modification ported to other platforms like Linux. The development environment looks a lot like Visual C (by Microsoft) which makes it easy accessible. The tool is still under development, but crashes are not really fatal.

Scite

Scite (Scintilla, 2005) is a text editor that can be configured completely. Examples of functions that are standard supported are:

- syntax highlighting
- code hints
- code folding
- abbreviations

A programming language called LUA makes it possible to create every possible command. There are several configuration files that can be edited with the editor itself. There are three kinds of options: Global, User defined and language specific. Global options are the same for every user. User options can be used to configure the editor for a user. Information like username can be stored here. A script could be build that for every user fills a header with the user information.

Doxygen

Doxygen (Heesch, 2005) is a source documentation application. Documenting sources is most of the times really annoying. If the sources meet certain constrains and comments are written with certain tags, Doxygen can find these tags and generate HTML, Latex, Man pages, RTF and XML version of the documentation.

1.8 Outline of the report

The second chapter gives an overview of the designed tool chain. The relation between the tools as well as the data path is explained. The chapter concludes with the problem of connection hardware to a model without changing the model itself, as is the current situation. The solution to overcome this problem is given.

The third chapter explains the tool chain in more detail and how the tool chain can be used to generate, deploy and control an embedded task.

The fourth chapter elaborates on the command and control environment. The chapter starts with the synchronized start option. The synchronized start of controllers in a distributed controller is crucial because all controllers depend on each other. A number of options have been investigated on how to propagate the start signal through the system and are explained in this chapter. An unwanted, but expected behaviour was noticed and is discussed at the end of the chapter.

The fifth chapter discusses the software designed and build to run on the embedded stacks. It will start with a discussion how Linux kernel drivers are written and what constraints they should obey. Following a general overview, the new version of the driver of the anything IO board used in HILS setup will be explained. After that a paragraph is dedicated on how the internals of LXRT work and why it is necessary for real-time communication. The last section of this chapter explains the application running on the stack necessary to make communication to the development station possible. The protocol and internal structure of that application are explained.

In the seventh chapter the demonstrations of the method will be explained. The first paragraph contains information about a, for this project developed, FPGA configuration, which allows easy data transfer between two PCs. The following paragraph will show the results.

The last chapter contains conclusion and states recommendations for further research.

2 MSC tools overview

The mechatronic stack connection (MSC) tools are a set of tools that allows rapid prototyping on any embedded controller that complies with the, in this chapter defined, requirements. In this chapter an overview of the tools is given.

2.1 Purpose

For verification and testing models of physical plants or controllers, a platform that can exchange physical signals is needed. In a previous MSc project (Groothuis, 2004) the hardware to do these tasks has been chosen and built. The hardware consists of an embedded PC104 CPU board, an Anything IO board and a CAN controller. The CPU board contains a VIA EDEN 667 MHz processor and standard PC connections like keyboard, network and VGA connections (SECO, 2005).

The assembly of the cards is called *mechatronic stack*. The mechatronic stacks are running an embedded version of Linux. The complete setup consists of 4 mechatronic stacks, two simulation PCs and a development station that are connected as given in Figure 5. The simulation PCs are used for simulation of the plant. The development station is a windows machine running the tool chain developed in this project.

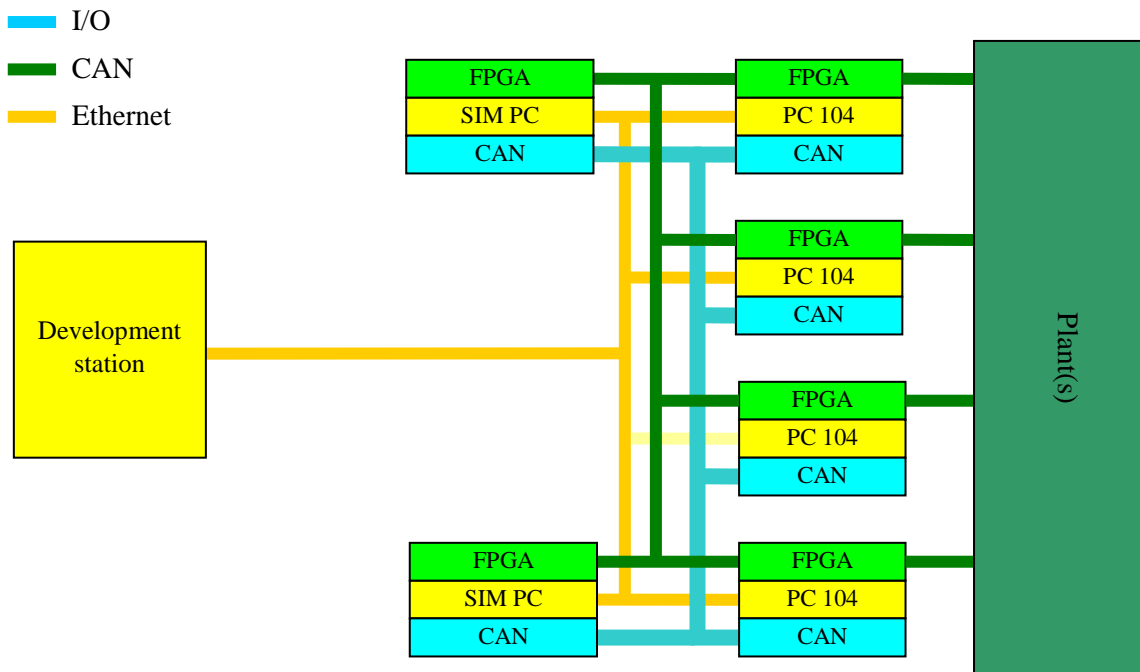


Figure 5: A complete overview of the setup.

Without the tool chain knowledge is needed of Linux to get applications to run on the stack. For example, how to build an application that can run on the stacks and how to setup a connection that can transfer data from a development PC to a stack or simulation PC and visa versa. Many modellers and control engineers are not familiar with Linux or software engineering. Many software engineers are not familiar with modelling and simulation. To close the gap a tool chain has been build to do the software engineering part so that modellers and control engineers can benefit from the embedded platforms that are used in the Boderc project.

2.2 Tools

To automatically generate executable control software from a model, in e.g. 20sim, the steps shown in Figure 6 should be performed. The designed and developed tool chain assists in this process.

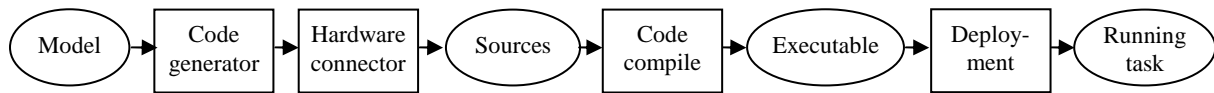


Figure 6: Transformation process

After the code has been generated by the modelling software an application called the *hardware connector* is started. This application allows the user to connect hardware to the model. In paragraph 2.5 the process of connecting hardware is explained. In paragraph 3.1 the hardware connector itself is explained.

After the hardware connector has finished, an application called the *compiler assistant* is started. This tool compiles the code with the appropriate compiler and the right flags. The application is explained in paragraph 3.2.

When the code has been compiled into an executable a tool called *deployment manager* is started. This application can connect to the embedded stack to start, stop tasks. The tool can also manipulate variables on the stack and retrieve information from the stack. The usage and implementation of the tool are explained in paragraph 3.3.

On the stack an application called *stack daemon* is running to allow the development PC to connect to the stack. The stack daemon and the designed protocol are explained in detail in section 5.4.

2.3 Extensible Markup Language (XML)

For this project several small applications were build instead of one large application to do the transformation process from model to a task running on the mechatronic stack. This method is called *piped filter* and allows intermediate starts and progress verification of intermediate results between the separate applications. This approach is chosen so that latter projects can benefit from one or more of these stand alone applications.

The problem when using multiple small applications is that data must be transferred between all these applications. Data could be passed from one application to the next by memory, but this approach is not preferable because if another project wants to use an application a way of passing the input parameters to the application must be specified. If memory transfers are used the developer of the new project must know the internals of the existing application, which is not preferable.

The use of a configuration file is preferable, because no knowledge of the application is needed and only knowledge of the structure of the configuration file is needed. If files are used to do the data transfer between the applications, a small parser that can read data from and write data to the file must be written. An advantage of an application specific parser is that the data that has to be stored to pass through to the next application can in general be smaller because there has to be no overhead because of generalization. A disadvantage is that the structure of the parser must be known to the developer that wants to use the application due to the removed redundancy. The learning curve of an application specific syntax will in general be bigger than the learning curve of a standard syntax because knowledge can already be present. Another thing is that errors in the configuration must be detected and a structure of error messages must be developed in order to find faults.

XML (W3C, 2004) is a document syntax that allows storage of any kind of data. The goal of XML is to make a document human readable and have a syntax check to easily find errors in documents.

The XML standard defines only the syntax of the document. The open structure is the strength of XML, because almost anything can be described in XML. This feature is also a weakness, because if the structure is not well defined, XML has the tendency to get unreadable, despite the fact XML is used.

An example of a piece of XML syntax that is used in this project is explained in Listing 1.

```
<CCE>
  <Executable Type="20 Sim" Name="example" Directory="c:\temp\example" ParamNr="2" VarNr="9">
    <Constants>
      <pi ID="0" />
    </Constants>
  </Executable >
</CCE>
```

Listing 1: XML example

Tags form the main structure of the XML document. A tag is a word surrounded by ‘<’ and ‘>’. The value between the brackets may be of any value as long as it does not start with a number and does not contain punctuation marks. A tag is closed by the same value, but starting with a ‘/’.

An XML document always has one root, in the example it is CCE. The root has in the example only one *child*, Executable. The Executable node has several *attributes*, Type, Name, Directory, ParamNr and VarNr. Nodes may have children, if a node has no children the tag ends with ‘/>’, the tag is closed immediately which means this is the only node and there are no further children. Nodes on the same level are called siblings.

The advantage of XML used as configuration file is that there are a lot of parsers available, and a lot of application can read and write XML. Data can be retrieved from XML documents by searching the parent child relations and the knowledge by the application of the document structure.

2.4 Architecture of the tool chain

The applications in the tool depend on each other. In order to pass the information that is added by each tool a configuration file is used. The configuration file, in the XML format, is “CCE options” in Figure 7 where the complete conversion process is shown. All the applications and the intermediate formats, except 20sim and gCSP, have been designed and developed for this project. 20sim or gCSP can be used to start the chain. Both 20sim and gCSP is also possible, but not yet implemented.

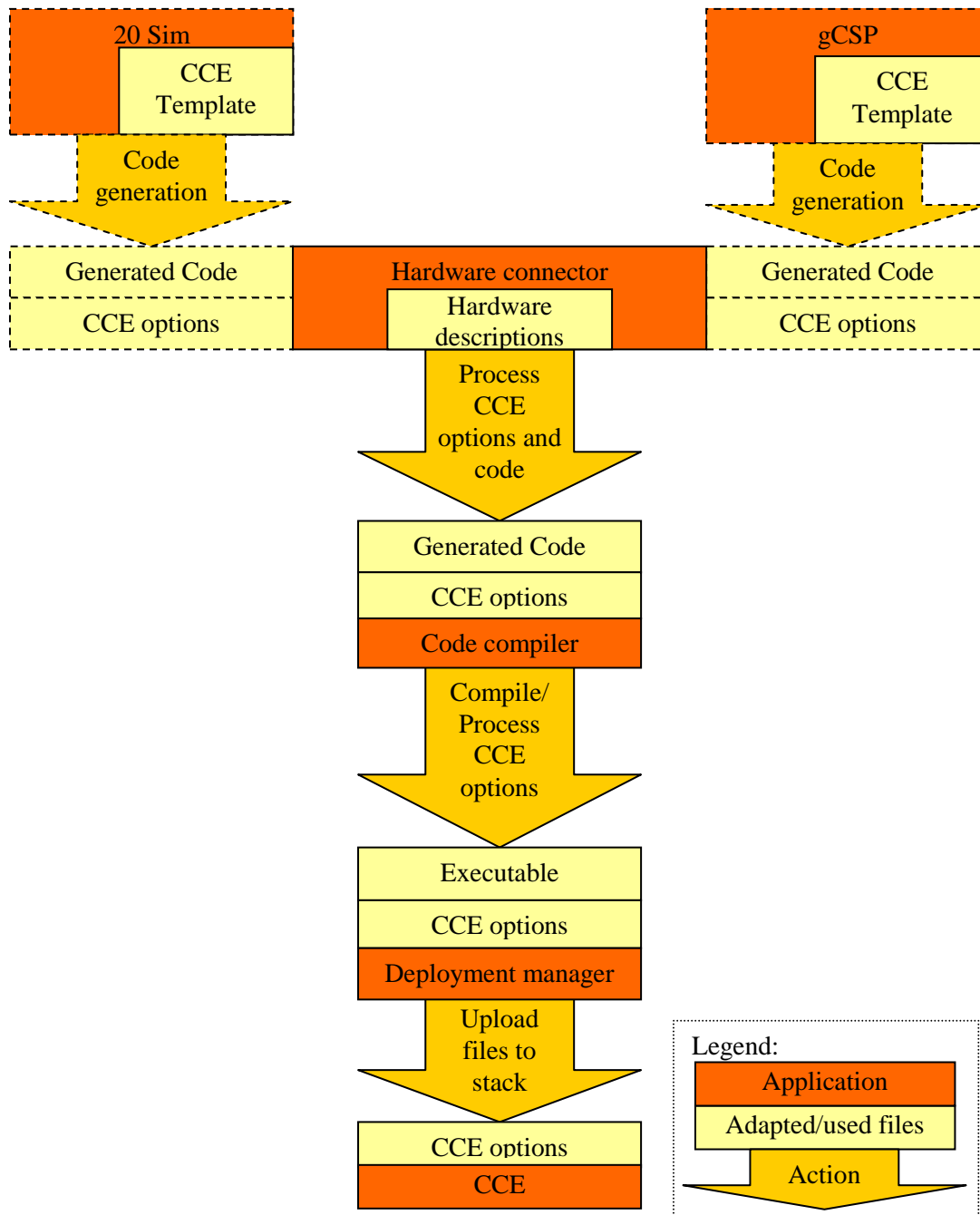


Figure 7: Conversion process from model to executable

When 20sim is used, a template of the configuration file is filled with project information by 20sim. Each tool in the tool chain will add information to the CCE options file and extract the information it needs added by a previous application. The information that is added to the file will be explained in the section about the tool.

For this project three libraries are built and implemented as DLL's:

- XXp.dll to parse 20sim template files
- tXML.dll which is a XML parser based on the tiny XML project
- UTMSc.dll for communication with the stack daemon

The tools depend on the libraries as shown in Figure 8. The library structure has been chosen to have an open structure that can be easily modified to future insights and usability for other application. Another advantage of a library is that implementations of functions can be changed without having to re-compile the applications, as long as the interfaces and number of interfaces remain the same.

The libraries are now in the form of DLL's. If the tools are to be used with another operating system, like Linux, the DLL can be rebuilt as libraries for that operating system. Most of the function implementations in the library are kept platform independent and should be easy portable.

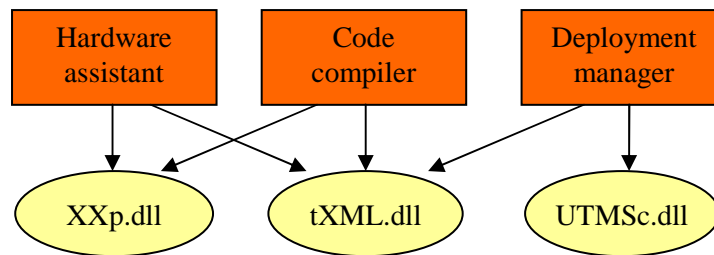


Figure 8: Tool dependencies

2.5 Generalization of hardware

This paragraph describes the difficulties and gives possible solutions when a general hardware independent model that can be converted to C code, must be converted to an application that drives real physical hardware. In these chapters 20sim is taken as an example to explain the processes. Any other tool that can generate C code from templates can be used in the same way.

2.5.1 Problem definition

When designing models with tools like 20sim the user wants to abstract the model from the real implementation. The model must represent the plant, but must not be dependent of any hardware implementations that do not belong to the plant. A plant that takes an analogue signal as input, for example, is not depending on the way this signal is generated.

The user must be able to specify the hardware in one of the last stages of designing. At the verification stage to be able to have a model that is completely independent of the hardware. Any hardware can now be used to generate the analogue signal from the example, as long as the timing and accuracy demands of the model are met.

A new problem is introduced by these constraints. If the hardware implementation is not known at the design stage, hardware must be described in such a way that every kind of hardware can be connected to the model without changing the model itself.

In the proceeding paragraphs solutions are given to overcome the problem of generalizing hardware by looking at what has to be defined to access the hardware.

2.5.2 Current approach

20sim has a function to use DLL's for special calculations in models. When C code is generated, the DLL function call will end up in the code with the connected signals as parameter vectors. What was done is that function calls with the same name as the DLL where added to the source by means of libraries. The linker connected the function calls to the final executable and made hardware access possible.

A disadvantage of this approach is that in 20sim an empty DLL must be made to allow simulation. The DLL is not a real calculation but a 'trick' to use hardware after the code is generated. Another disadvantage is that the naming of the function calls gets blurred. The templates needed to cope with the DLL function calls, and initialization had to be performed in the template. For every piece of hardware a different template is needed. If more than one device is used the number of templates will grow exponentially to allow the user to use any kind of hardware configuration. Not taken into account is the fact that hardware may need to be initialized differently in some situations. For example an analogue input or output card may have a range of 0..10V or 10..20mA. If these options must be selectable by the user even more templates need to be made. In the next section hardware will be

generalized in order to overcome the problem of multiple templates and get rid of the DLL ‘trick’ in 20sim.

2.5.3 General hardware procedure

When looking to other tools that can be used to transfer data to the environment like dSpace, Simulink and Comedi there are similarities in the way the actual data transfer is managed. dSpace and Simulink have a set of devices that can be used and a standard in how data must be transferred. These devices or devices that can communicate in the standard way can be used in their code. A drawback is that user defined code is hard to implement and only predefined hardware can be used. Hardware that is not in their database or does not communicate in their standard can not be invoked.

Comedi is an open framework where specific pieces of code have to be added. This approach is good if the developed software must be able to switch fast between hardware. A drawback is that there is a large overhead for simple devices.

Combining the options from the three mentioned applications a general structure for hardware can be extracted and will be given in the next sections.

General overview

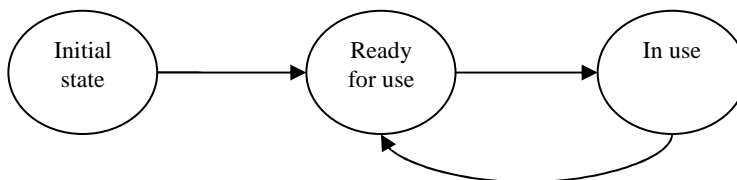


Figure 9: Common hardware sequence.

If looking at hardware from a different perspective, just what has to be done to get data from the CPU to the environment and visa versa, all hardware behaves in the same way. After starting up it is in an initial state as shown in Figure 9. After initialization of the hardware by a driver it is ready for use and can be used by applications. Applications then open the device by retrieving a handle to the device from the driver and start an application specific initialization. The hardware is now in use and ready for reading, writing and control operations. When the application is done using the hardware, the driver is notified by the application it no longer needs the handle and the device is reinitialized.

Initialization

Before the device can be used, a handle has to be retrieved from the kernel if an operating system is used. Through this handle the device is known by the kernel and the requested operation will be directed to the appropriate device. After the handle has been retrieved initialization may be needed to put the hardware in the desired state.

When looking at devices like DSP boards which, under normal circumstances, have no kernel, a handle can be defined as the base address of the control registers. Initialization has to be done on the registers with an offset to the base register and can be initialized in the same way as with an operating system, with a difference to use real hardware addresses instead of a handler. This is possible because the location of the addresses does not vary because additional hardware can in general not be added to the device.

When the hardware is in the desired state the initialization is finished.

Read, write and I/O operations

Read, write and I/O operations need to be done on the previous retrieved handle or I/O address. The function to get and put data from and to the hardware is in most cases a simple read or write operation. If reading and writing is more difficult most of the times libraries are used that convert the more difficult operations into a simpler version.

Deregistering

When the application wants to terminate, the hardware may have to be reinitialized. It could be possible that, for example, the FPGA configuration can lead to dangerous situations when the control from the CPU is removed. For this kind of actions the application can reinitialize the hardware into a safe state. After the hardware is in the desired state the retrieved handle has to be returned to the kernel in order to notify the kernel that the device is no longer needed. The returning of the handle, in case a device can be opened just once, must be done to allow other applications to open the device.

2.5.4 Code generation process and generated application

In order to describe the possible solution, knowledge of what happens when code is generated and executed on the embedded stack is needed.

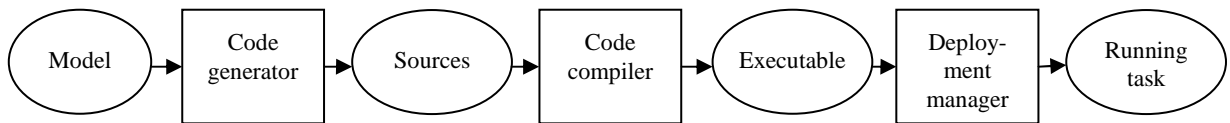


Figure 10: Transformation process

The actual transformations are seen as black boxes, the square boxes in Figure 10. For now it is not important to know the details of these black boxes in order to understand the process. The black boxes will be explained in later chapters.

The C code that, for example, 20sim can generate is build from templates. These templates contain the framework of the application and the model specific code is inserted into this framework by 20sim. When 20sim has generated C code after the model is completed these files need to be transformed by a black box into an executable application that can be run on the embedded stack, or any other processing device. The embedded stack or other processing device will be called *target* from now on. After the executable has been generated the executable is transferred to the target by a black box and started there. Possible solutions on how to communicate to the hardware are given in the next section.

2.5.5 Possible solutions

There are two moments in the conversion process the hardware can be connected to the software. The first moment is the moment before transforming the code into the executable. If the hardware specific code is inserted into the generated code at this moment, the generated application ‘knows’ how to handle the hardware.

The other possibility is to use standard hardware functions in the application and build an intermediate layer on the target that transforms the general hardware data transfer calls to hardware specific calls. This approach is called *wrapping*. The principal of the wrapper is shown in Figure 11.

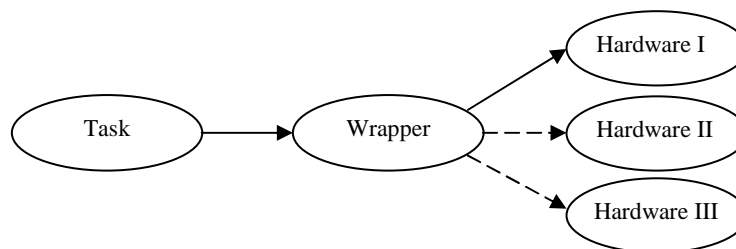


Figure 11: hardware call wrapper

Advantages of the first method are that the hardware can be accessed faster because the commands are directly send to the hardware and that no additional software is needed to do the data transfer. Advantages of the second method are that the model stays relatively simple and that other hardware can be connected without changing the executable.

2.6 Conclusion

The tools can not only be used with 20sim, but with any tool generating c code. The tool chain has an open structure, which allows the user to use the tools with any kind of software development environment. Parts of chain or the whole tool chain can be used to do rapid prototyping of any kind. If, for example, only code compilation and uploading to the stack is needed without the need of hardware connection, the chain can be started with the compiler. The result will be that the sources are compiled and the generated executable can be uploaded with the deployment manager. The only options that have to be changed are in the configuration file.

It is possible to generalize hardware in such a way that it can be connected to the model without changing the model. The hardware connector, explained in paragraph 3.1, uses code modification to connect the hardware to the model because this is the most general form and can be easily ported to other architectures. With this approach the DLL does not have to be used anymore in 20sim and signals can be directly connected.

To use already designed models that use the DLL ‘trick’ with the tool, the functions previously defined in the libraries have to be inserted into a hardware description template as described in Appendix III Hardware configuration files. The DLL functions must be removed from the model, signals must be reconnected by the hardware connector and the model must be rebuilt with the tool chain. The rebuilding process is fully automated and should be easy. The only part that can lead to difficulties is the template implementation part because no tool is yet available to generate a template and it has to be done by hand.

3 MSc tools implementation

3.1 Hardware connector

This method of the modifying code can be easily integrated into 20sim and is not depending on any software running on the target. This makes it easy portable to other targets. In the next paragraph the hardware connector will be explained and all the detail of the implementation will be mentioned.

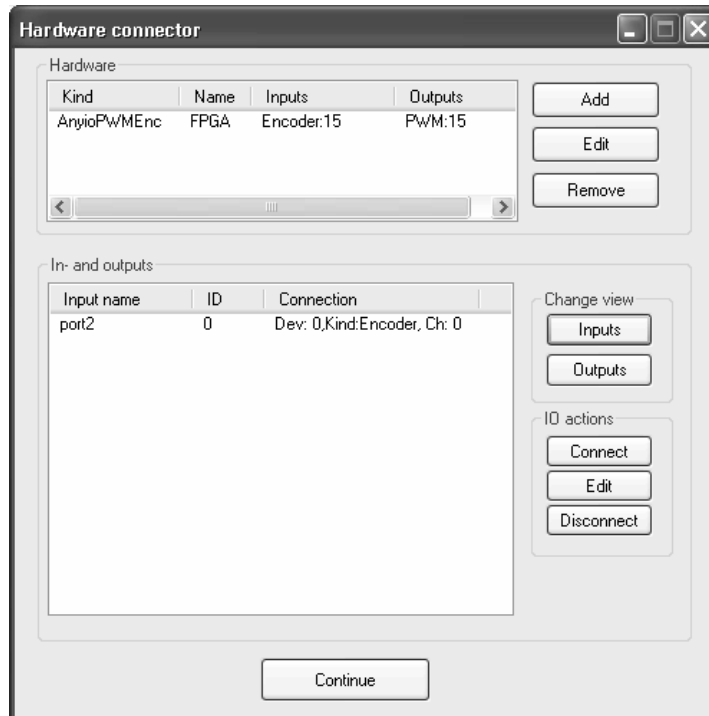


Figure 12: Hardware connector

3.1.1 Graphical user interface

Figure 12 shows the graphical user interface of the hardware connector. The top list box contains the hardware that is currently used. The *Name* field will contain the name of the configuration that has been specified by the hardware configuration designer. *Kind* contains the kind of hardware that is used. In and outputs will contain the kind of in and outputs that are in the configuration in the form kind:number. If more than one kind is available the next kind is separated by a comma and added to the list of inputs or outputs.

The bottom textbox contains the in or output of the current model. With the add and remove buttons hardware can be added and removed. When the add button is pressed a file chooser dialog is shown. The configuration files are all in the XML standard and contain the functions that are needed to control the hardware.

If the connect button or the edit button is pressed a dialog box as shown in Figure 13 is opened. The device, IO kind and channel can be selected here.

With the continue button the settings will be converted, the code will be modified and the application will be closed to be followed by the code generator.



Figure 13: Connect a channel

3.1.2 Implementation of the hardware connector

When 20sim generates the code, a list of variables is also generated in the template file made for this project. This list contains all the variables used in the model and also all the inputs and outputs of the model. The hardware connector reads the list at start up to extract the in and output names. After the list has been processed it checks if a hardware configuration file is already present.

The hardware configuration file is explained in Appendix III. In this appendix the use of the hardware configuration, as well as the hardware description files is described.

When hardware was connected to the model in a previous run these settings will be loaded, checked for changes in inputs and outputs of the model and put into the hardware manager to speed up the generation process. It is assumed that in most cases the hardware connections do not change in case of a model changes.

When hardware is added or removed, the description information of that device is added or removed to/from the hardware configuration file. When channels are connected to or removed from inputs or outputs of the model, the hardware configuration file is immediately updated, by adding or removing the connection in the hardware configuration file.

When continue is pressed, the hardware connector will only need the hardware configuration file, because this file contains all the necessary information.

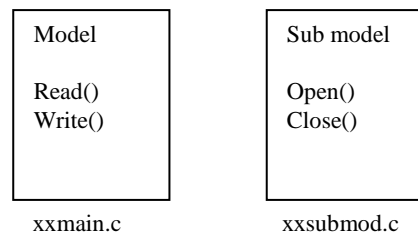


Figure 14: Standard c model of 20 Sim

To be able to smoothly integrate the hardware connector into 20sim changes have been kept small to the 'standard' c code generation files. Figure 14 shows the organization of the 'standard' template files. The model file does the read and write operations and the sub model file opens and closes the hardware.

In this first setup of the hardware connector only two files, xxmain.c and xxsubmodel.c, contain the hardware dependent information. In future versions of the hardware connector it is recommended to scan all the template files to get rid of the file naming dependency of the hardware connector and have a more flexible configuration that can handle 20sim modifications of the 'standard'. This has not yet been implemented, because of timing constraints of this project.

The hardware connector first reads the sub model file xxsubmod.c and scans it for tokens to be replaced. This file will contain after modification the open and close routines of the hardware. First the include files, needed to compile the source, are added. After that the global variables and open and close routines of all hardware devices are added. If necessary, channel initialization and removal is added to the open and close routines respectively.

The actual transfer of data from the model to the hardware is added in the main loop of 20sim. First the include files, necessary to compile the source, are added. After that, the global variables and the read and write routines are added. If necessary, data scaling functions are added as last to the main file to be able to use any kind of format. 20sim uses variables of the double format. Most IO devices use integer values so that in most cases transformation of the value is needed.

The model files do now have all information needed to control the hardware. The hardware connector starts up the compiler, after modifying the project configuration file by adding the libraries and the include paths that are needed for the hardware access functions.

3.2 Compiler assistant

The compile process of the sources generated by the modelling tool and modified by the hardware connector is depending on what hardware is used. To be able to use the hardware the compiler needs to know where the headers are and what libraries must be linked to the final executable.



Figure 15: Code compiler dialog

When the compiler assistant is started, first the variables used in the model will be parsed and added to configuration file. This step is necessary to generate the information about the variables needed in following applications like the deployment manager and the CCE environment.

All the preceding applications add library and header file location information to the configuration file. The code compiler started by the compiler assistant will extract the information from the configuration file and start compiling the sources with the appropriate settings. The output of the compiler is redirected to the text box on the GUI. In case of an error the textbox will contain this information in order to be able to correct the error and have an overview of what is happening. Warnings will also be shown in the box but will not break the chain. In case no errors have occurred the deployment manager will be started. An advantage of using the compiler assistant is that the compile process is now controlled.

An advantage is that all library and header dependencies are now handled by the compiler assistant. This prevents manually editing the script files, which needs knowledge of compiler flags and script languages. The code compiler allows the user to generate an executables without knowledge of compiling sources and script languages.

3.3 Deployment manager

After the code has compiled, the program and hardware configuration specific files have to be uploaded to the embedded stack. The deployment manager has been developed with the constraint that other targets must be usable. Linux is now the only supported target, because this is the only target the stack daemon is available for. If DSP is used, a plugin to the USB connection has to be made which uploads the executable without a stack daemon. The DSP cannot run a stack daemon, because DSP has no OS and can thus not run multiple applications by default because it has no multithreading support.

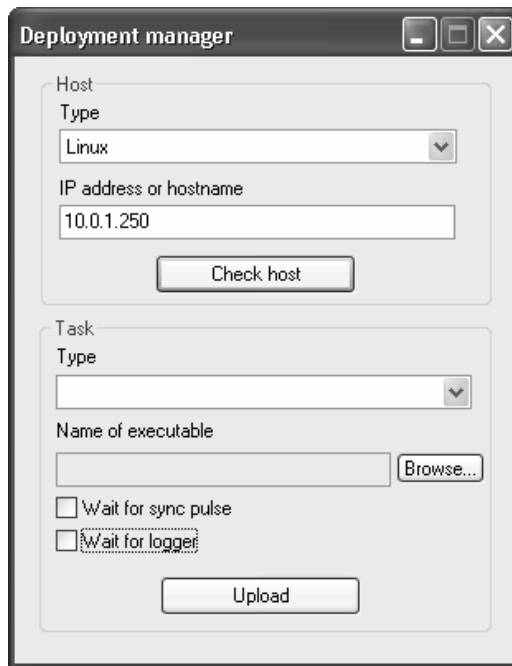


Figure 16: Deployment manger GUI

Figure 16 shows the deployment manager GUI.

Features:

Before uploading the stack needs to be checked if it is an expected state. Checking the stack can be done by pressing the ‘Check host’ button. The deployment manager will try to contact the stack. If the stack is unreachable a message will be shown telling to check the address of the host. If the connection is successful the GUI in Figure 17 will be shown. The stack in this example has one 20sim task running called Testmodel. The nodes of the model contain the variables that are in that node name category.

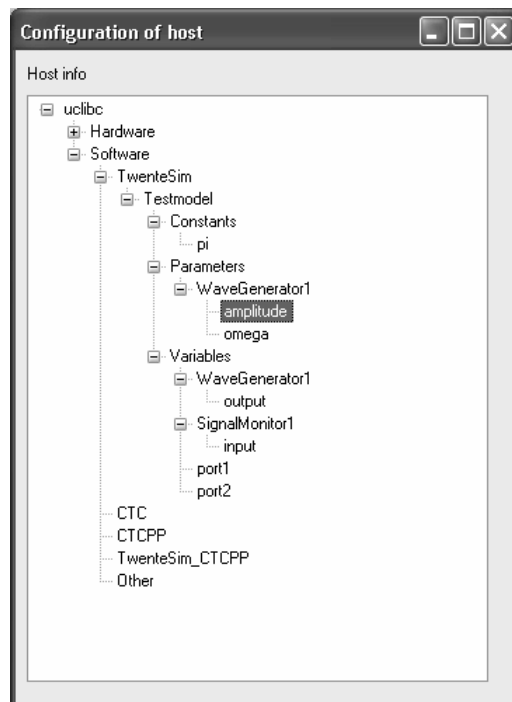


Figure 17 Overview of configuration

If there are any tasks running on the stack that were started by the stack daemon the deployment manager can kill these tasks. Tasks that are not started by the stack daemon can not be killed because

no housekeeping information of the process is available. Killing a task is done by selecting the to be deleted task and pressing delete.

Viewing and modifying parameters and variables is also possible with the deployment manager. To view a value of a parameter or variable the item has to be selected. After the selection the F2 button has to be pressed (Normal change command of Windows). The GUI in Figure 18 will be shown. The name of the parameter or variable will be in the top of the dialog (in this case the amplitude of WaveGenerator1 as shown in Figure 17). The value shown is the value of the moment the get value command was processed on the stack. The value can be changed to the appropriate value and written to the stack with the 'write' button. Cancel will close the window without changing the value on the stack.

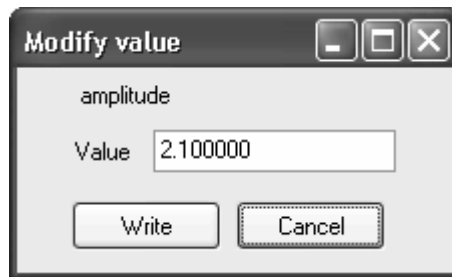


Figure 18: viewing and possibly changing value GUI.

Uploading:

Tasks can be uploaded in three states:

- Direct start
- Synchronized start
- Logging start

The direct start mode is used when synchronized start and logging are both not selected. After the task and configurations have been uploaded the task will be started by the stack daemon. The synchronized start will result in a task that has opened the hardware but is not started yet. The *sync server*, explained in paragraph 5.4.2, will listen to the CAN bus for a start event. The start event will trigger the start of the application. The logging start will result in a waiting task on the stack. The task will wait for a logging event from the client to start the task. The purpose of the logged start is that the logger will start the task and that results from the model are logged from the first moment. The first time stamp in the logged values will be time 0.

Usability of the various upload states is given in paragraph 4.3 with the command prompt logging commands.

Before the tool chain, uploading of the task and configuration was also done by scripts. A disadvantage is that addresses had to be known before the script was started. If, again, a fault occurred the scripts would exit and not inform the user about the fault. Deletion of tasks was not available, and starting task had to be done by hand.

3.4 Conclusions

The hardware connector allows the user to connect any kind of hardware to any model. The hardware connector stores the connected channels to make it possible to change models and not having to reconnect the hardware to the signals.

The compiler assistant compiles with the flags defined and links the needed libraries to the executable. The output of the compiler is redirected to the GUI to allow the user to debug any faults that occur.

With the deployment manager the user can select a host, check the host and upload the task with the advantage of error messages in case of an error. Target platform inspection is supported and the user has control of tasks that are uploaded with the Deployment manager. The deployment manager enables rapid deployment of task without knowledge of the internals of the target platform.

The deployment manager can also be used to retrieve identifier information of processes, variables and parameters. To get the information the node has to be selected that the information is needed from and the space bar has to be pressed. A dialog will appear with the identifier information. This is very useful in combination with command line commands.

4 Command and Control Environment

The Command and Control Environment (CCE) is a set of tools that can send information to- and extract information from the embedded stacks. The framework and a set of command line tools have been built that can perform the tasks. The only thing not built is graphical user interface. The protocol may have to be extended with other function that where not thought of at this time.

4.1 Purpose

Rapid prototyping demands a flexible development environment that can generate control software and makes it possible to tune parameters. Starting, stopping and restarting of tasks must be possible with only a few clicks in order to get results faster. To compare result from simulations and the HILS the embedded stacks must be able to transfer data to the development station. In order to be able to do multiple runs of the same test with different values the development station must be able to change values in the stack applications without having to rebuild the software.

4.2 Interfaces

The UTMSc.dll is a library written for this project that contains all the functions that are implemented in the protocol. Appendix IV gives a detailed overview of all the functions and defined words used in the DLL. The DLL contains interfaces that can be used from applications. A description on how to use the DLL is given in this section.

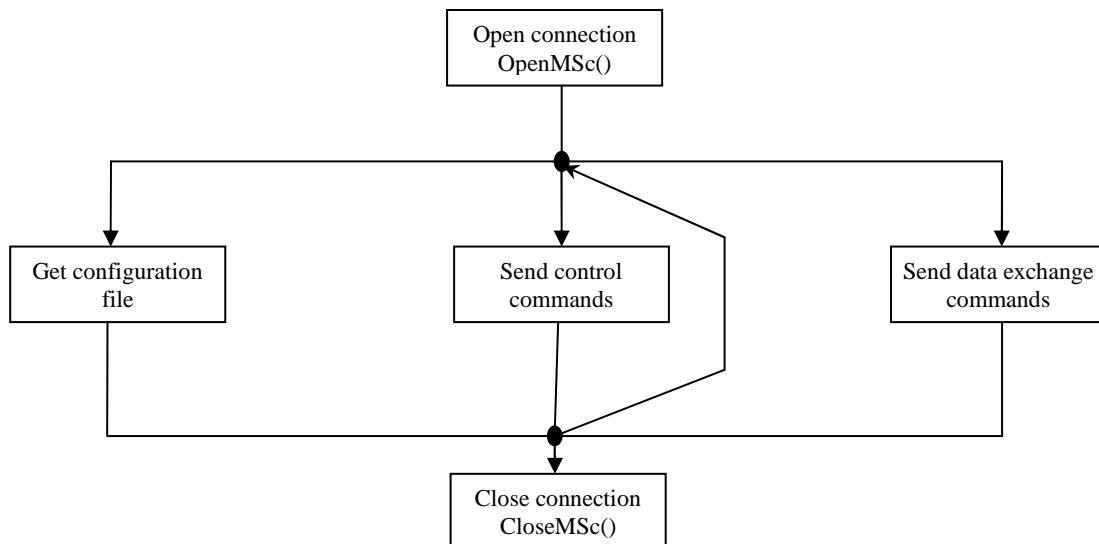


Figure 19: State chart of protocol.

Before the connection can be used it first has to be opened as shown in Figure 19. The four mechatronic stacks and the two simulation PCs must be controlled and therefore six connections at the same time must be possible. The DLL supports six opened connections at the same time. After a command has been send, a new command can be send. If no more commands have to be send the connection must be closed in order to let other stations connect to the stack daemon.

The configuration file that resides on the stack daemon makes it possible to connect the CCE to already running tasks. Information about the running tasks and variables of these tasks are stored in the configuration file. The DLL can retrieve the configuration file from the stack. The configuration file also contains identifiers of processes and variables.

4.3 Synchronized start

To start distributed 20sim models and other distributed software experiments over multiple systems a synchronized start is needed. A synchronized start starts the application on all the participating devices

at exactly the same time. In the simulation phase this is done implicitly because all the controllers run in one simulation environment and there are no real-time constraints. On the embedded stacks the start of the experiment has to be done on every stack separately.

A timestamp is defined as the number of control loops that have been performed from the start of the controller. The result of a synchronized start is that the logged values on all devices started by the synchronized start have the same timestamp if running at the same frequency. Comparison of the real values with the simulated values can be done without any pre-processing because the timestamp of every sample should be the same.

The control loop frequency of the mechatronic stacks in the HIL setup is chosen to be 1 kHz. The sample frequency of the simulation PCs is ten times faster than the controller frequency as shown in Figure 20.

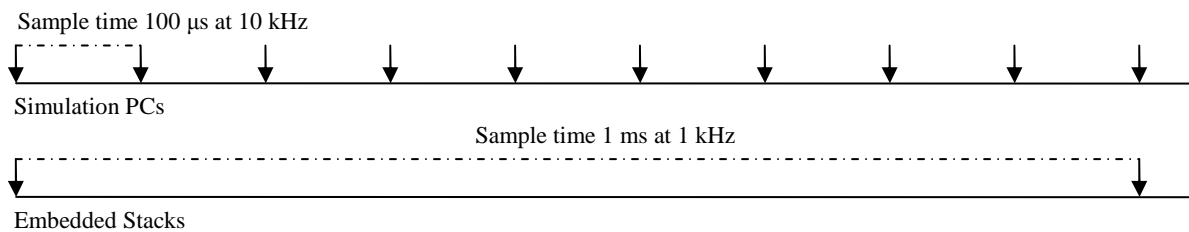


Figure 20: Sampling time

In order to get a timestamp that is the same as in the simulated situation, all the stations need to be in the same cycle. The time can be calculated by the sample frequency. If the simulation PCs are used the start command needs to be propagated through the system in maximal 100 μ s. If only the stacks are used the start command needs to be propagated through the system in less than 1 ms to keep every station in the same step, because they run at ‘only’ 1 kHz.

The maximum propagation time may have a maximum latency of 100 μ s in order to be sure that all the stations are in every situation in the same step.

In the following section a comparison of some methods to accomplish a synchronized start is described. First a small introduction to the techniques is given where after a more detailed explanation is given.

4.3.1 Investigated options

- **FPGA pin:**
The Anything IO board has 72 digital IO pins. One of these pins can be used to send or receive the start signal.
- **CAN:**
On the stacks a CAN bus controller is available which can communicate to all the systems in the HIL setup. CAN is able to broadcast messages to all the connected devices.
- **Ethernet:**
All the systems are connected by an Ethernet connection. It is possible to broadcast messages and in this way signal all the systems to start the experiment.
- **Parallel port:**
All the boards have a parallel port. These ports have 8 pins that can be used as IO port. All the systems can be connected to a master that sends the start signal.
- **Serial:**
 - **232:** Serial 232 is a point to point connection. A daisy chain, as shown in Figure 21, could be build to use this option.
 - **485:** Serial 485 is a multi master system. All systems can be connected to the same line and in this way a broadcast can be generated, but additional hardware is needed.

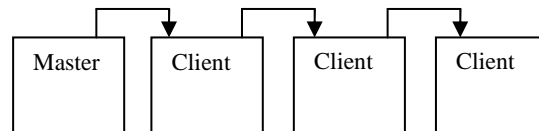


Figure 21: Daisy chain

- **USB**
All the boards have a USB port. All the USB ports can be connected by a USB bridge and in this way reach all devices to send the start signal.
- **Fire wire**
Additional hardware is needed to support fire wire because there are no firewire boards on the setup. Fire wire has a synchronized broadcast that can signal the systems.

4.3.2 Detailed descriptions of synchronized start options

FPGA pin

The FPGA can use external signals as a clock signal. There are 2 pins that can be used as an external clock. If a setup would be build with this option, one system will be the master and all the other systems slaves. The master generates a signal on a pin. This pin is connected to the other slave systems clock pin. The signal is converted to an interrupt which signals the system the experiment must be started.

Advantage:

- Fast, because the interrupt is generated the moment the signal arrives.
- Easy connection, only one wire is needed to connect all the devices.
- Cheap, the IO board is already there.

Disadvantage:

- Dangerous, if a fault is made in the configuration and there is more then one master on the line the outputs of the two master systems will short circuit.
- Elaborate, all the FPGA configurations have to have support for the start option. The driver and link drivers need to be adapted to support the start interrupt.

CAN

CAN is a serial connection between all the systems. In this case, there is again one master and the other systems act as slave. All slave systems can be put in a listening state. When the master broadcasts the send message, all the slaves can be started. The master is will also read its own broadcast and will started at the same time as the slaves.

Advantage:

- Fast, because an interrupt is generated the moment the message arrives. A disadvantage is that the board is located at the ISA bus which is slower then the PCI bus. But because al the systems suffer the same latency, it doesn't have to be a problem.
- Easy connection, the CAN bus is already connected to all the devices.
- Cheap, the IO board is already there.

Ethernet

All the systems are connected by Ethernet, because it is used to program and control the systems. If it is used to synchronize the start of an experiment it has to be taken into account that a hub is used and not a switch. The difference between the two is that a switch contains logic that does the routing which can cause latency and a hub is a passive device that just passes the messages.

Two protocols can be used, TCP and UDP. The advantage and also the disadvantage is that UDP has no correction protocol. The packet is sent and assumed to arrive. With TCP transmission faults are corrected, but this is a major disadvantage, because no knowledge is about how long it took to correct

the error. TCP has also the disadvantage that it can not send broadcasts and every participating member has to be notified. It is thus preferred to use UDP to start the experiment and have an acknowledgement from the receiving systems while the experiment is already running.

Advantage:

- Fast, application can wait for the broadcast.
- Easy, Ethernet is already connected.
- Cheap, Interface is already there.

Disadvantage:

- Experiment demands, because an acknowledgement has to be send, the system cannot be used at its full speed or else the message will never be send.
- Latency, the server will not receive its own message and synchronization of the master with the slaves can be a problem.

Parallel

The parallel port has 8 bidirectional IO pins. In this option again one system will be the master. All the slaves will have one of their IO pins connected to a pin on the master. The moment the experiment starts the master will change the value on all the connected pins. In this way the clients will be signalled.

Advantage:

- Cheap, the parallel port is on the systems.

Disadvantage:

- Dangerous 1, parallel ports are very sensitive for faults. In case of a cable mismatch or if the master is wrongly configured permanent damage can be the result.
- Dangerous 2, all the nodes need to be on the same supply. If one of the supplies has a dip in the voltage, the connected system or all the other systems will burn their parallel port.
- Usage, the parallel port is now already used to drive the status display on the HIL setup.
- Latency, the server will not receive its own message and synchronization of the master with the slaves can be a problem.

Serial 232

There are two serial ports on all the systems. RS232 is a point to point connection. In order to connect all the systems together a daisy chain needs to be built. This means that the second serial port of is connected to the first of the next. The message will propagate through the system when all systems echo the message from the first to the second serial port.

Advantage:

- Easy, all devices need to be connected by two null modem cables
- Cheap, all the systems have two serial ports.

Disadvantage:

- Slow, the message has to propagate though all the systems at a maximum of 115200 Bits/s. Because of the daisy chain the transmission time has to be multiplied by the number of systems. The first system has started the transmission time earlier than the last station.

Serial 485

On the PC104 boards there is on communication port that can be set in RS485 mode. This mode makes it possible to build a network with on single cable. The setup will contain one master that broadcasts the message to all the systems.

Advantage:

- Easy, just one cable that connects to all the systems RS485 ports

Disadvantage:

- Cost, for the two simulation PCs additional RS485 cards have to be bought
- Latency, the server will not receive its own message and synchronization of the master with the slaves can be a problem.

USB

By means of a bridge multiple USB hosts can be connected to each other. It is possible for the host to communicate to another host. Broadcasting messages is not possible because the protocol defines an endpoint. And thus every slave has to be informed about the start event.

Advantage:

- Easy, connecting all the hosts to a switch

Disadvantage:

- Not synchronous, because all the systems need to be signalled sequentially.
- Costs, a USB 6 port hub and 6 host to host cables need to be bought.

Fire wire

Firewire card can be daisy chained to each other. The protocol supports isochronous channels which can be used to signal all devices. A disadvantage is again that, like USB, all slaves have to be signalled after each other.

Advantage:

- Easy, the devices can be connected by a standard firewire cable.

Disadvantage:

- Not synchronous, because all the systems need to be signalled sequentially.
- Cost, additional hardware has to be bought for the pc104 stacks and the simulation machines.

4.3.3 Conclusion

	Ease of connecting	Chance on damaging	Time of sync propagation	Costs to this project
FPGA	+-	--	++	++
CAN	++	++	++	++
Ethernet	++	++	+	++
Parallel	+-	--	+-	++
232	+	++	--	++
485	+	++	+	-
USB	+	++	-	--
Fire wire	+	++	++	--

-- is really bad, - is bad, +- is normal, + is good, ++ is very good

As can be seen CAN is the preferred option and the option that is used in this project. The hardware is already there, easy to connect, low probability on wrong connection and very fast.

When measurements were performed on the performance of the synchronized start an expected but unwanted behaviour came up. When using multiple controllers with unsynchronized clocks drift may occur. When the experiments were started synchronously drift was observed of one millisecond in a ten minute run. An external synchronization is thus needed to keep the controllers in the same calculating period. The CAN bus can also be used for this purpose, but it is advisable to do a new inventory of synchronization methods. If, for example, a digital input is taken to synchronize the controllers a signal generator could be used to clock the devices. In this way a variable clock is possible.

4.4 Command line programs

For the general functions, command prompt applications have been built. The set makes it possible to manipulate the tasks running on the stack.

Starting and stopping of task is not possible with the command line programs, but this functionality is available in the deployment manager that allows visual information of the stack in order to make life easier. The functions that can be performed by the command line tools are:

- Getting values of variables and/or parameters on the stack (*getval*).
- Changing variables and/or parameters on the stack (*setval*).
- Automatic logging of variables and/or parameters values to a file (*logval*).
- Manual logging of variables and/or parameters values to a file (*startlog*, *getlog*).
- Sending a synchronised start command (*sendsync*).

With this set of tools all functionality needed for rapid prototyping are given. All commands take parameters that can be passed to the application in the form *-argument identifier=value*. An example is:

```
getval -a=MS1.utwente.nl -p=1234 -i=123 -v=21,3,5,24
```

This example will get the values with identifiers 21, 3, 5 and 24 of process identifier 123 from port 1234 of the stack with the name MS1.utwente.nl.

In the next section the optional parameters are surrounded by {}. The identifiers of processes and variables/parameters can be retrieved by the deployment manager. The section on the deployment manager (paragraph 3.3) describes how to retrieve the identifier information.

4.4.1 Sending a sync pulse (*sendsync*)

Starting of experiments synchronously over multiple controllers and/or simulation PC's can be done by uploading a task to every controller and simulation PC with the wait for sync mode in the deployment manager. The *sendsync* command will send a synchronize package on the can bus and takes the following parameters:

a: Address of the stack

{p}: Port number on which the stack daemon is running. Default is 1500.

The stack must be connected to the CAN interface in order to be able to send the synchronized start command. The stack may participate in the experiment, in which case the command will start the local waiting tasks also. If the stack does not participate the signal will only be send to the CAN bus because no local application is waiting.

4.4.2 Getting Values (*getval*)

The retrieval of values can be done with the *getval* command. The *getval* command can take the following parameters:

a: Address of the stack

{p}: Port number on which the stack daemon is running. Default is 1500.

i: Process identifier of the process where the variables are located.

v: Identifiers of the variables that have to be retrieved. The argument is in the form: *identifier {, identifier }* The pattern between the brackets can be repeated for every value that needs to be retrieved.

The retrieve command will output the time stamp of the values and the values themselves. The time stamp is used as an indicator on at what point in time the sample is taken. The output of the values is given in the form:

ID α contains β

α and β in the retrieved output contains the appropriate identifier and value respectively.

The retrieval of values is very useful for quick checking of a variable or some variables. The command can not be used for logging because the retrieval interval depends on Windows which is by definition not real-time. The function can however be, for example, coded in a loop to see every second how a signal is evolving or incorporated in 20sim to display results.

4.4.3 Manipulating values (*setval*)

The manipulation of values can be done with the *setval* command. The *setval* command can take the following arguments:

- a: Address of the stack
- {p}: Port number on which the stack daemon is running. Default is 1500.
- i: Process identifier of the process where the variables are located.
- v: Identifiers of the variables that have to be changed with the value that needs to be written. The argument is in the form: *identifier: value{, identifier: value}* The pattern between the brackets can be repeated for every value that needs to be written.

The command will return a message with the status of the write action (success, fail). Values can be written at any point in cycle, even when a task is waiting to be started by a synchronized start or logging event. It is possible in this way to do several runs with the same task but other values. To do this the task is uploaded with the wait for sync option. The task will be started but not yet running its control loop. The values that need to be changed can now be changed to the appropriate values. After the values haven been changed the synchronize signal is send to start the task and the task is running with the modified values. To repeat the process in order to tune the control loop the task can be deleted and uploaded in the wait for sync option again. The values can be tweaked and this loop can repeated until the desired behaviour is met.

4.4.4 Automatic logging of values (*logval*)

There are two kinds of logging, automatic and manual. The difference is that by manual logging the values have to be retrieved by hand, while automatic logging does it all automagically. The reason for implementing manual logging is explained in the paragraph of manual logging.

The *logval* command can take the following parameters:

- a: Address of the stack
- {p}: Port number on which the stack daemon is running. Default is 1500.
- i: Process identifier of the process where the variables are located.
- v: Identifiers of the variables that have to be retrieved. The argument is in the form: *identifier {, identifier }* The pattern between the brackets can be repeated for every value that needs to be retrieved.
- t: Simulation steps that need to be logged. The stack daemon has no notion of the speed on which the controller is running, for that reason ticks have to be given. If a controller runs at 1kHz and the time to log values is 1 second a 1000 ticks have to be given as the ticks argument. The ticks can be calculated by multiplying the frequency by the log time.
- f: Filename of the file where the values have to be logged.

The *logval* command will start the logging process on the stack and will try to retrieve the values every half second. When the stack daemon is still logging it notifies the *logval* command that logging is still active. After another half a second the *logval* command will try again. This pattern will repeat until the stack daemon is ready logging. The values will than be stored to a file in the comma separated format that look like this:

```
Timestamp,value{,value}
{Timestamp,value{,value}}
```

The comma separated format has been chosen because a lot of software packages understand the commonly known standard.

When tasks are uploaded in the wait for log state they can be started with the automatic logging command. The result will be that the log will contain the values from the start point up to the time that has been specified by ticks.

4.4.5 Manual logging of values (*startlog,getlog*)

Manual logging of values separates the logging process into two commands. One starts the logging process and another to retrieve the values. Starting the logging process is done by the *startlog* command that can take the following parameters:

- a: Address of the stack
- {p}: Port number on which the stack daemon is running. Default is 1500.
- i: Process identifier of the process where the variables are located.
- v: Identifiers of the variables that have to be retrieved. The argument is in the form: *identifier {, identifier }* The pattern between the brackets can be repeated for every value that needs to be retrieved.
- t: Tick that need to be logged. The stack daemon has no notion of the speed on which the controller is running, for that reason ticks have to be given. If a controller runs at 1kHz and the time to log values is 1 second a 1000 ticks have to be given as the ticks argument. The ticks can be calculated by multiplying the frequency by the log time.

The command to retrieve the values is the *getlog* command that takes the following parameters

- a: Address of the stack
- {p}: Port number on which the stack daemon is running. Default is 1500.
- f: Filename of the file where the values have to be logged.

The manual logging functions come in handy when a distributed control that uses the synchronized start is used that has to be logged from the first cycle. First all stacks are loaded with the tasks in the wait for sync state. With the manual command logging is started on all the stacks. After all stacks are in the logging state and waiting for the sync pulse the pulse is send to all of the stacks and all stacks will start and log from the beginning for a certain period of time. After the logging has finished all the log files can manually be retrieved from all the stacks to be compared with the simulated results.

5 Embedded stack

The embedded stacks used in the HIL setup run on an embedded version of Linux. The embedded version has a subset of the runtime functions that can be used in Linux. The subset is called micro C libraries (uClibc) (Andersen, 2004) which is a subset of the GNU C libraries (Glibc) (GNU, 2001). The kernel is a standard version modified with RTAI to support real-time behaviour. RTAI uses a set of kernel patches that make Linux have real-time capabilities. A set of modules must be loaded into the kernel to do the actual scheduling and other real-time functions.

5.1 Linux device driver standards in the Anything IO driver

When this project started a device driver to control the hardware of the anything IO board was available. This driver was developed at the University by (Groothuis, 2004). The driver was able to load the FPGA that is on the board and do IO operations on the board. The driver was a first version and did not use the Linux standard approach of Linux kernel drivers (Rubini and Corbert, 2001). An example is that the FPGA had to be programmed by a write operation to the board. A write operation is normally used to do output data transfer from the CPU to the device. To be able to maintain the driver more easily the driver was rewritten according to the Linux standards.

5.2 Anything IO device driver

A device driver is a special kind of kernel module. Kernel modules in Linux are used to run in the kernel and perform a certain task there. A difference between a kernel module and an application is that an application has a main function. When the main function of an application terminates the application terminates and memory is released. A kernel module has no main function. Instead it has several functions that can be used by everyone if they are defined as such.

The kernel module always has at least two functions: `init_module` and `cleanup_module`. The `init_module` is to initialize the kernel module and tell the kernel what functions are available and to register all resources needed by the module. The `cleanup` does the opposite. It tells the kernel the functions are no longer available and releases all resources.

A device driver is a special kind of kernel module because it accesses hardware. When a device driver is loaded a driver can scan if any hardware is available or can take parameters when loaded that tell where the hardware is located. An advantage of automatic hardware scanning is that no hardware information has to be given to load the driver. A disadvantage is that all possible locations must be scanned for the hardware.

5.2.1 Init and cleanup

The Anyio driver is implemented to not take any parameters. An advantage of the Anything IO boards is that they reside on the PCI bus. The PCI bus has a register of devices that are connected. The kernel has this register of devices that can be used by any driver. The driver asks the kernel at start up if any Anything IO device is available. The resources of the found device will be returned to the driver that allocates the memory resources. This loop is continued until the kernel can find no more devices. The devices are then available for use and the driver exits its initial phase.

The cleanup function loops through all the registered devices, releasing and deregistering the resources.

5.2.2 The /proc and /dev file systems

When the Anyio driver is loaded the kernel can access its functions and use the device. In order for applications to use the device, an entry point that is accessible to the applications has to be added. The normal way in Linux is to register the device under the /dev file system. The user can open and close

the device by using its /dev name, for example /dev/anyio0. By this device handle read, write and control operations can be performed on the device.

To get information from a device or a group of devices Linux has a special kind of file system called the “/proc” file system. The /proc file system has a normal directory structure that contains ‘normal’ files. If the file is read a special function in the driver is executed that outputs the status of the devices in text file format. When data is written to the file, another function in the driver is executed that can take any kind of format. This method of reading and writing is implemented in the driver. Programming and other control functions of the FPGA is available in the current version with the implementations these functions. Reading the files gives information about the board or boards. Writing to the files programmes, resets or activates a configuration in the FPGA.

5.2.3 Read and write operations

The read and write functions are currently not implemented. The purposes of the Linux read and write functions are to do a transfer of multiple bytes from or to the board. An advantage of using the read and write function is that user space programs can access the device by standard read and write functions of the Linux kernel. The /dev entry is used for these functions. The reason the read and write functions are not yet implemented is that there is not yet any FPGA configuration that requires read or write function to large amounts of bytes. The moment that a FPGA configuration is available, a standard way of transferring large blocks of data must be determined and used in future FPGA programs. The current approach allows reading and writing to and from the device in a different way.

5.2.4 Input and output control operations

The Linux kernel has a special way of communication, other than read or write, between program and driver. Some devices have special functions like, for example, setting the baud rate of a serial device. These functions can not be called read or write functions, but do need to be accessible from applications. For this kind of functions I/O control is implemented. The Anything IO has functions that must be implemented as I/O control to satisfy the Linux standard. Functions that are currently implemented are FPGA control functions and special read and write functions of a byte, integer, or word from or to the device. The FPGA I/O control functions now make it possible to control the FPGA from within the application. With the previous version of the driver the application could not program the FPGA. To be sure that the right FPGA program was loaded into the FPGA an extra check was implemented to check the name and version of the running FPGA program. A disadvantage of this method is that the FPGA program is active before the application is started which can lead to undesirable side effect like uncontrolled steering of I/O. In the new version of the driver, the application has full control over the FPGA and can program and start it at any moment in the application. Resetting the FPGA is also possible to return to a failsafe mode, in case of any error.

When a FPGA program is compiled the compiler will determine the value of the unused pins. It will sometimes occur that the interrupt pin, when unused, gets set by the FPGA configuration. Linux will receive an interrupt from the FPGA and try to service it if an interrupt service routine is implemented. Because the pin cannot be reset it will stay high. The Linux kernel will again execute the interrupt service routine. A life lock is the result because only the service routine is executed sequentially. To prevent this behaviour the interrupt service routine can be enabled and disabled by I/O control operation for configurations that need the interrupt.

5.3 LXRT extension of the driver

All Linux system calls like reading, writing and I/O control depend on the Linux kernel. Because Linux is by definition not real-time these functions cannot be executed in real time.

LinuX Real-Time (LXRT) is an extension to the Real Time Application Interface (RTAI, 2004) developed at the DIAPM (Dipartimento di Ingegneria Aerospaziale - Politecnico di Milano) of Milan, Italy and makes it possible to execute kernel functions in real-time by a special interface. The Anything IO board is used in real-time experiments and a real-time interface is needed.

I/O control functions are managed by the Linux kernel. Any real-time application using these functions was left to real time characteristics of Linux and can never be real time. To the interface of the Anything IO driver two real time entries have been added, reading and writing. With these functions reading and writing of a single or multiple bytes, integers or words is possible in real-time. The overhead of the LXRT interface is in the order of micro seconds (RTAI, 2004). Measured result are 12,1 us with I/O, 7,9 us without I/O.

5.4 Stack daemon

This paragraph describes the management program running on the mechatronic stack that is called *stack daemon*. The stack daemon is the communication gateway between the stack and development station. All commands from the development station to the stack are sent to the daemon. The daemon processes the command and performs the requested action on the stack.

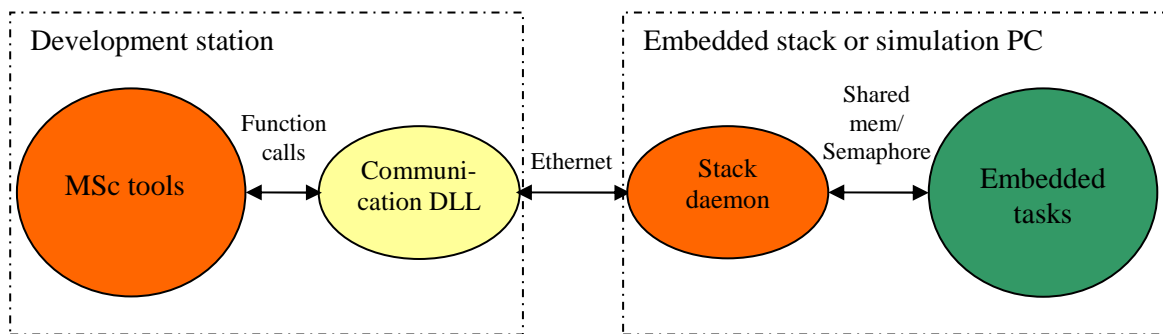


Figure 22: Stack connection overview

5.4.1 Protocol

The connection between the communication DLL and the stack daemon is of the server client type and are connected as in Figure 22. The stack daemon is the server that processes requests from the DLL. After the connection has been established command can be send to the stack daemon. An advantage of the client server model is that the embedded stacks, that run the server side, do not need to waist any processing power when no client is connected.

To keep the protocol simple and easy to process, a general structure for all packets has been chosen. In this way the stack daemon can determine what action block should be performed and overhead is kept low.

Commands are split up in groups of command types in order to make processing on the server side faster. The server can determine from the command which part of the stack daemon code should be executed. Every group has the same general packet format and is extended if necessary.

There are five groups of commands:

- Task commands
- Programming commands
- Exchange commands
- Logging commands
- Utility commands

The commands are sent by TCP/IP packet to the stack daemon.

General packet:

cmd	<data>
-----	--------

Every packet contains a 32 bits integer value with the command. After the command the data field is filled with the group specific data. The data field can have a variable length.

Task commands:

cmd	task ID
-----	---------

The data field will be filled with the process ID of the task. There are five commands that can be sent to the stack daemon to control an application. The commands are start, stop, suspend, resume and kill. The commands perform the following:

- Start: Starts a task.
- Stop: Stops a task and can be restarted.
- Suspend: Stops the given task from executing. The process information is stored. From this point the task can only be killed or resumed.
- Resume: Resume a previously suspended task.
- Kill: Stop the task from executing and delete it from the memory.

The following commands described can have a length field. The length field contains the length of the message after the length field. This is done to be able to separate the packages on arrival and to check if the complete package is received. Applications using TCP/IP will not receive packages in the same order and of the same length as they are sent by the sending application, because of the OSI layers. If, for example, five packages are send the receiving side may have to do ten read actions. It may also be possible that al bytes are received at once. The described behaviour is typical for TCP/IP and a result from the protocol. Error checking is not necessary in this case because the TCP/IP protocol already takes care of that. The data field will be filled with the XML string that belongs to that program.

Programming commands:

cmd	length	housekeeping info	program/configuration
-----	--------	-------------------	-----------------------

A programming command transfers a task or configuration to the mechatronic stack. The message will contain housekeeping information for the stack daemon. The housekeeping contains information about what is send and what to do with the configuration or task after upload. An example is the wait for sync tag, which notifies the stack daemon the task should be started by the SyncServer, explained in paragraph 5.4.2. After the housekeeping information the package contains the task or configuration files.

Exchange commands:

Get:

cmd	length	PID	indices
-----	--------	-----	---------

Set:

cmd	length	PID	index	value	index	value
-----	--------	-----	-------	-------	-------	-------

Exchange commands make it possible to read or write every variable in the model. This prevents variable selection in advance to the actual simulation, as is the case with the previous version developed by Control labs.

If the command is a request for values, the data field will be filled with a process identifier and identifiers of the variables to retrieve. PID contains the identifier of the task and indices contain the indices of the variables. The stack daemon will return the timestamp and all values in the order of the request.

Answer:

timestamp	values
-----------	--------

If the command is a write of values, the data field will be filled with a process identifier and identifiers of the variables to write immediately followed by the value to write. If more than one value is written the pattern index value must be repeated for every value.

Commands that do both at the same time are not possible because there is no read/write command.

Logging commands:

Start logging:

cmd	length	PID	indices	Duration
-----	--------	-----	---------	----------

Get Log:

cmd	data
-----	------

The *start logging* command will start a task on the stack that logs the values given in indices to be logged for a certain time. The PID field contains the identifier of the process the values need to be logged from. The maximum time and/or values that can be logged depend on size of the internal memory of the stack or simulation PC. Values are first logged into the internal memory and the size of the internal memory limits the maximum logging size. After logging is completed, the development station has to retrieve the values itself by the *get log* command. If logging is still active the values can not be retrieved to guarantee real-time behaviour and logging of all the data. If the log has not been retrieved the logging can not be started again to prevent data loss.

Utility commands:

cmd	<data>
-----	--------

At the moment no utility function uses the data field. The data field is available for future commands.

GetConfig:

GetConfig gets the configuration from the stack as it known to the stack daemon at that moment in time. The data field is empty. With the GetConfig command, applications that connect to the stack can retrieve the status of the stack. This option allows disconnecting from the stack daemon and at a later point in time continuing without having to know anything about the status of all tasks. Keeping the status locally could also be implemented but has a big disadvantage. For example if a task is has terminated a local configuration will not be updated without a connection. The stack daemon will receive a terminate signal from the task and update its local configuration. If the GetConfig command is issued the configuration without the terminated task will be given.

SyncStart:

The stacks that participate in the synchronous start first have to be initialized. First a task with a Sync="yes" option has to be uploaded to the stack. This will place the uploaded task into a waiting queue for a synchronized start. After all the participating stacks are in the waiting state they can be started with the SyncStart. This command has to be sent to any stack that has connection to the CAN bus. A participation stack can be used, but also a different stack that has access to the CAN bus can be used to send the command.

Sync start sends the start command to the CAN bus and start a task simultaneously on multiple stacks. The data field is empty.

5.4.2 Implementation

The implementation of the stack will be explained by a 20sim generated control loop. Other applications can be used in the same way but are not available the moment this document is written.

Housekeeping

When the stack daemon is started, an empty XML document is generated to support the internal housekeeping. The document contains information about the hardware and the software. At the moment only the software information is available. Hardware information and information of the CPU are implemented as dummy interfaces and can be filled with information about CPU usage and the status or configuration data of the hardware.

When a task is uploaded the housekeeping information, such as variable names and variable identifiers of the task, is added to the internal document. When the configuration is downloaded to the development station the document can be parsed to extract the stack status. After the information is added and the task is started, the process identifier is added to the configuration in order to allow access to the application. When the task quits or is killed by the user the process information will be removed from the configuration in order to keep the document up to date.

The reason XML is used for the housekeeping is that the embedded stack can just add the received information to its internal structure after adding the process identifier information. In this way the processing is done at the development station kept away from the stack.

Memory access and starting

In normal operation the stack daemon and the application share one shared memory and two semaphores as shown in Figure 23.

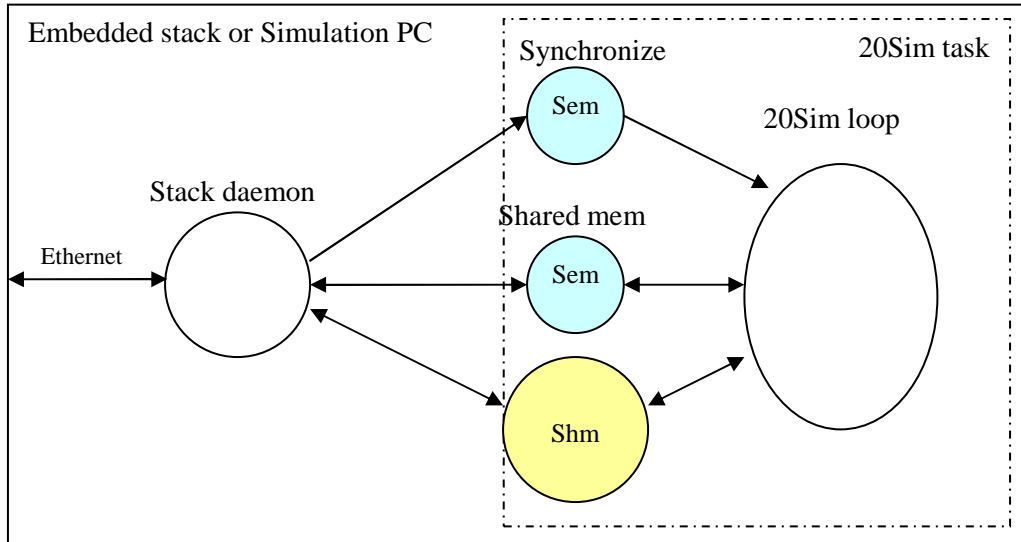


Figure 23: internal connection on the stack

The shared memory semaphore is to guard the memory from mutual access by the stack daemon and the 20sim task. It has been implemented as a resource semaphore that has the nice feature of priority inheritance. Priority inheritance prevents priority inversion which can cause deadlock of a process (Buttazo, 2002). When the stack daemon or the application wants to read the memory the semaphore is checked, if it is there the semaphore will be taken and the memory will be read or written. If the semaphore is not there the application will wait for it to reappear and then read the memory.

The synchronize semaphore in Figure 23 is used to give the stack daemon the ability to start the application at any give point in time. If the application has been uploaded without the synchronized start or logging option, the application will start as soon as it is uploaded that is, the stack daemon will signal the application right away. If the synchronized start is used, a separate thread, called *sync server*, is started to listens to the CAN bus for a synchronize signal. If the synchronize signal is received, the sync server will signal the application to start as shown in Figure 24. This multithreaded approach has been chosen to give the user the ability to use the stack daemon even when a task is waiting to be synchronized. If the multi approach was not chosen the stack daemon would be listening to CAN bus and to user could not access it anymore. Waiting applications had to be started before they could be aborted which is an unwanted result.

If the logging option is chosen the application will be started after the logging command has been send from the development station. The logger will signal the task to start by the synchronize semaphore. This option makes it possible to log values from $t = 0$.

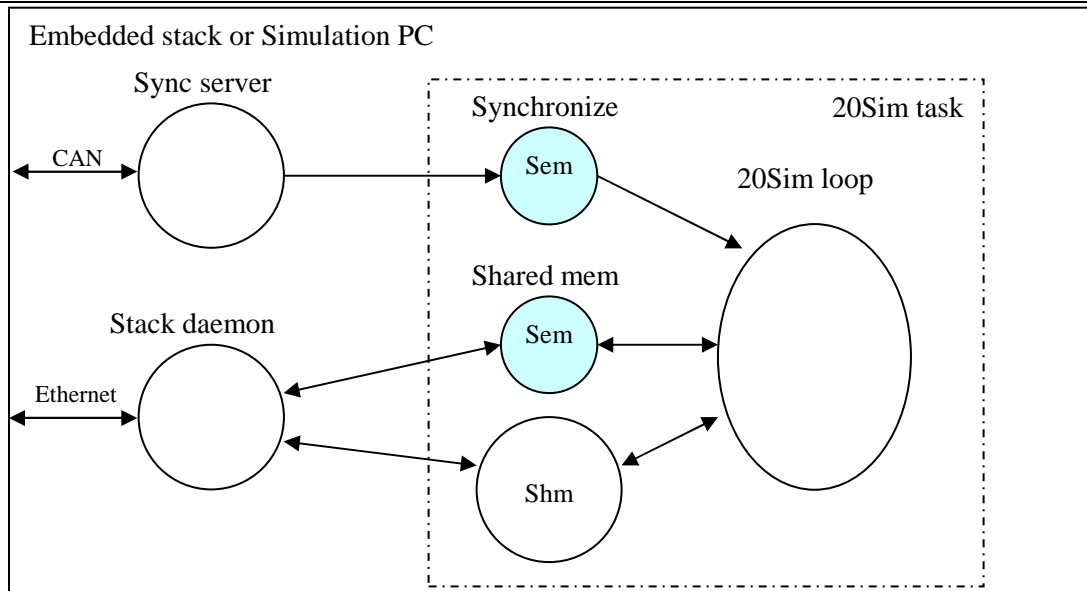


Figure 24: connections when waiting for a synchronized start

For the logging of values, another semaphore is used to synchronize the stack daemon with the task that needs to be logged as shown in Figure 25. This semaphore has been implemented as a binary semaphore. If the semaphore is given and the semaphore has the value 0 the value will be raised to one. If the value of the semaphore is one it stays one in contrary to normal semaphore that keeps counting. If the semaphore has the value one, another process can take the semaphore. If it wants to take it again it has to wait until some process gives the semaphore again. In this way synchronization of two processes can be achieved with the advantage that the taking process does not need to exist.

Logging

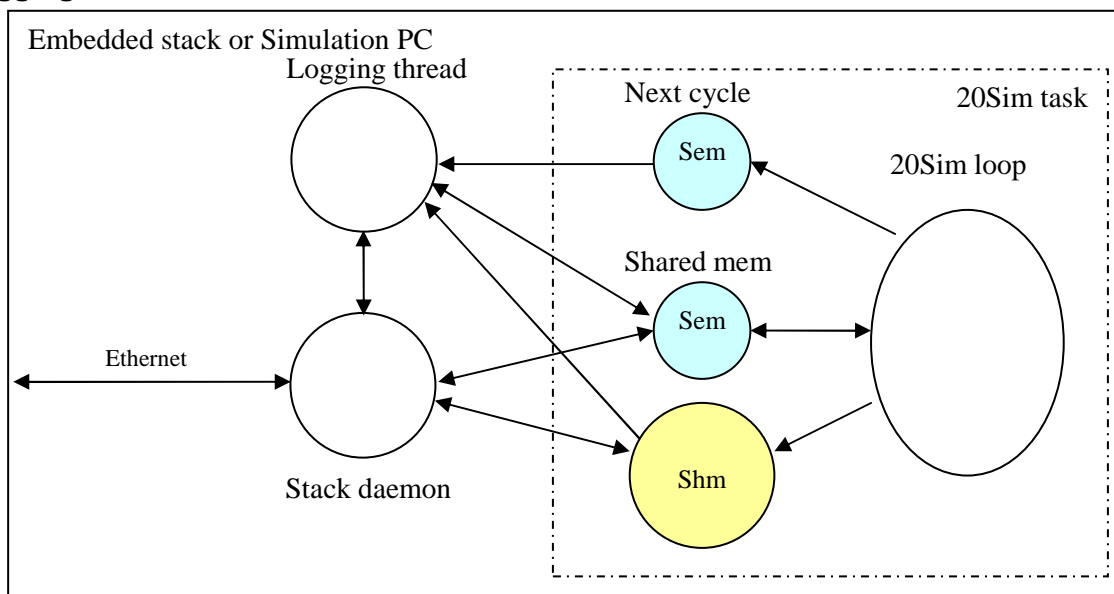


Figure 25: Stack daemon in logging mode

If logging is requested by the user the stack daemon will start a separate task that takes the semaphore and stores the current value of the requested variables. The separate thread is used to provide the possibility to stop a logging action because the stack daemon will still be running. All other commands will also be possible, so changing variables while logging is an option. The application that started the logging action can even be disconnected and connected at a later stage when, for example, logging is finished.

After storing the values the logging thread will cycle to a next take action of the binary semaphore and gets blocked until the task has made another loop and gives the semaphore again. The logging thread

will do this loop for the requested amount of times. After the loops have been done the stack daemon will be signalled that the values are ready for transfer. The client has to retrieve the values by itself. If the logging action has not yet been completed the client will be informed that the logging process is still busy by returning an error value. If the logging has been completed the values will be returned to the stack daemon. The next retrieval of log values will return the values to the client.

Memory map

20sim has an option of using favourite variables. With the previous CCE used at control lab products, it was necessary to insert all variables that could be monitored and modified into the favourites. With the new tool chain all variables can be modified and monitored which has the advantage of not having to select variables at beforehand.

The 20sim generated code uses a static array to store all the data. In the new design, developed for this project, the static array is copied into a shared memory after every cycle. At the beginning of the cycle the memory is copied from the shared memory to the array. In this way a little overhead is generated by the copy action, but all variables can be accessed at runtime.

Measurements have been done on a model provided by OCE with 164 variables. The calculation time of the model without the copy action takes 56 us. The calculation time of the model with the copy action takes 62 us. The copy action takes 6 us of processing time for 164 values, which is 6% of processing power for a 10 kHz controller and 6‰ for a 1 kHz loop.

The modified 20sim loop looks like Figure 26. The initial memory is copied to the shared memory before entering the loop to keep the first loop the same as the following loops and to be able to retrieve and modify values when the task is waiting for a start event. When the loop is entered the memory is copied from the shared memory and the calculations are performed. After the calculations the memory is copied back and the loop is blocked by the timer. If the timer expires the next loop is performed.

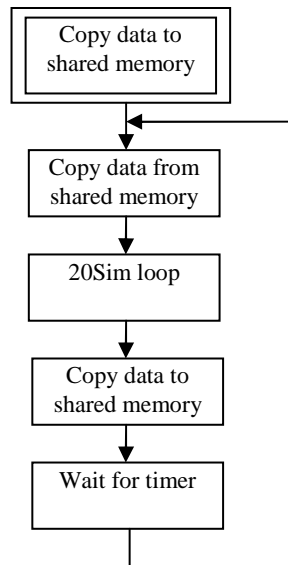


Figure 26: 20sim loop for MSc

6 Demo setups

To demonstrate the usability of the tools a demo setup from the University called 'linix' is used (see Figure 27). This setup was also used in project of (Groothuis, 2004). The results will be compared to see if the results from this tool chain are the same as with the scripts, which should be.

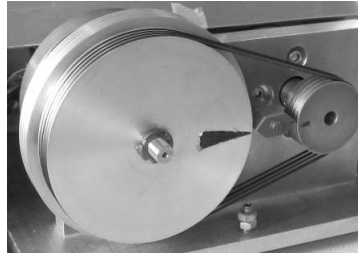


Figure 27: Picture of Linix

6.1 Hardware

The used hardware will contain one PC104 board and one simulation PC. In the first two tests the PC104 and a simulation PC will be used. The simulation will be faster than real time using the direct link developed for this project and the second simulation will be the real-time simulation with the inverse PWM and encore FPGA configuration developed by Groothuis. In a third test the PC104 will steer the real linix with the same controller as used in the real-time simulation. The HIL setup will be controlled from a development PC.

6.2 Software

The developed tool chain will be used to generate the control software. Two controllers will be generated, but with different hardware. The first controller will be connected to the directlink FPGA configuration, which consist of 15 32 bits and 15 16 bits input registers and 15 32 bits and 15 16 bits output registers. The simulation PC will also be connected to the directlink FPGA configuration, as explained in paragraph 6.3. This setup will be demonstrated with the synchronized start method. The model that was used in the previous project will also be used for this project, with the difference that the templates of this project will be used.

The second controller will be connected to the hostmot FPGA configuration which consists of 15 PWM outputs and 15 encoder inputs. This controller will be used in the real-time experiment and to steer the real 'linix'. The plant for the real-time simulation will be build with the inverse PWM encoder FPGA configuration. This configuration can read PWM signals and steer encoder pulses. From the controller there should be no difference between the setup in the simulation and the real linix.

The on-line parameter modifying will be demonstrated by building the model with the parameters to turn the small wheel one turn. After modification of a value the big wheel will make one complete cycle.

6.3 Direct link

Direct link is a FPGA configuration developed for this project.

6.3.1 Purpose

After a model has been simulated in 20sim and code has been generated testing the model without looking at the real-time behaviour is a good test to see the effect of the signals to the model. If the example of Figure 28 has to be simulated on the HIL setup the controller must run on the mechatronic stacks and the plant on the simulation PC. To transfer the data between the PCs, a link is needed that can transport the data. To test the accuracy of signals that is needed the accuracy of the D/A converter and the quantizer must be selectable.

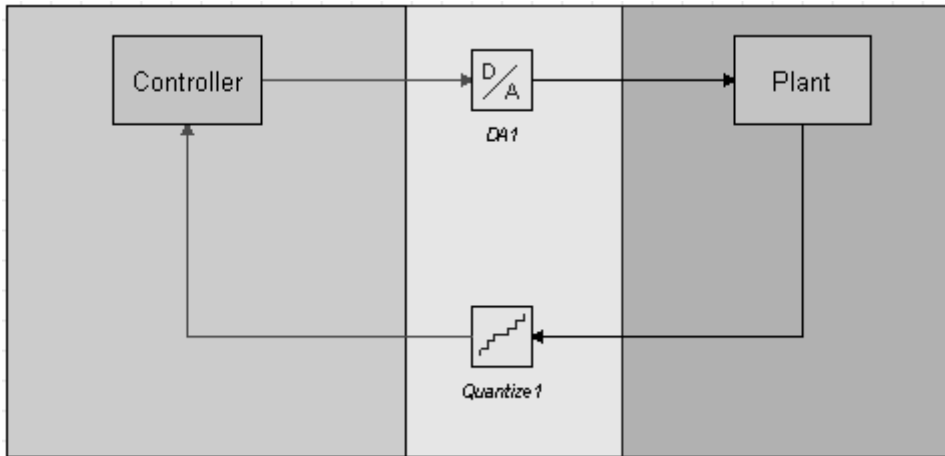
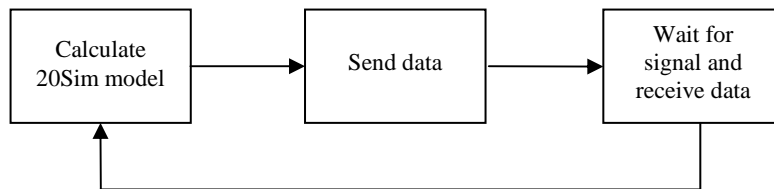


Figure 28: 20sim model example

The Direct link FPGA configuration makes it possible to transfer data from one board to another. When the data has been completely received by the receiving side an interrupt is generated to signal any waiting task the data is ready. In this way the controllers can calculate the model, send data and wait for the other side. This loop does not have any real time behaviour but is a good first test.



Another usability of the configuration can be the faster than real-time simulation. Because the controllers just wait for the other side the loop of controller plant can performed at maximum speed. After calculating the values at one side the data will be send to the other side and calculated there. In this way it is possible to do simulations faster then in the real environment which can also result in rapid prototyping.

6.3.2 Implementation

The configuration consists of two buffers. Data of the input and output memory can be read, data of only the output memory can be written. Reading the output buffer can be used to check the last written data.

Schematic layout of the configuration is given in Figure 29.

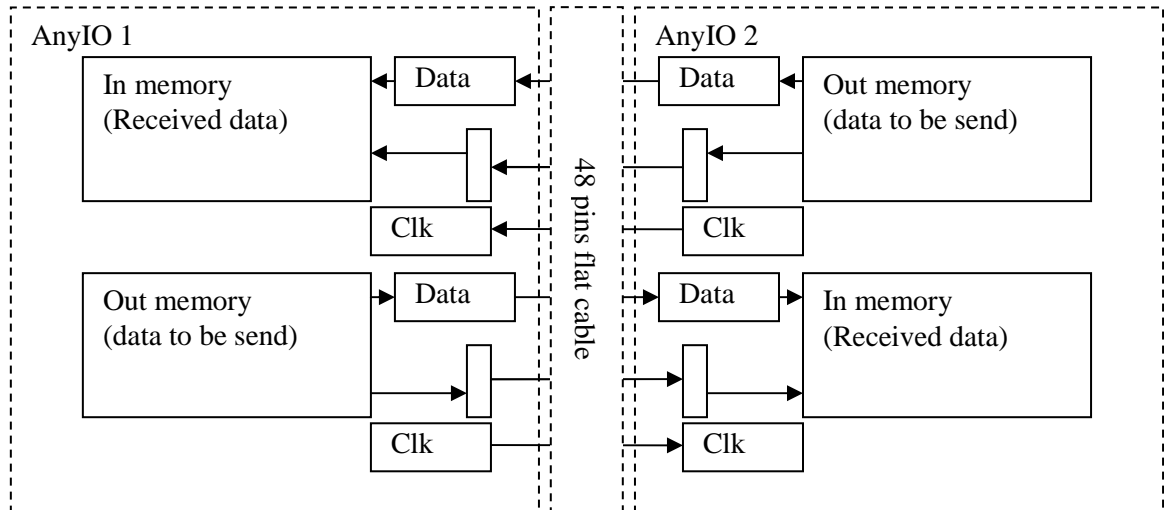


Figure 29: Schematic view of the direct link.

Pin layout:

Output pins	Function
24	Master Clock
25	Sending
26-33	Data out

Input pins	Function
48	Receive clock
49	Receiving
50-57	Data in

Port three and four need to be cross connected between the two anything IO boards. A loop back test is also possible. For the loop back test, port three has to be connected to port four on the same device.

Adress range	Bit width	Address separation	Memory	Register #
0x1C00-0x1C3C	32 Bit	4 Bits	Out	0-15
0x1840-0x185E	16 Bit	2 Bits	Out	32-47
0x1C60-0x1C9C	32 Bit	4 Bits	In	0-15
0x18A0-0x1BE	16 Bit	2 Bits	In	32-47

The control registers are located at 0x18FF and 0x1CFF. When data is written to this address, the board will start sending the data to the output at the frequency of the PCI bus divided by four which is 4/33 MHz. All registers will be sent to the other side in one byte at a time basis. Every transfer of data takes:

$$((15 * 4) + (15 * 2)) * 4/33\text{MHz} = 90 * 1.33 \mu\text{s} = 120 \mu\text{s}.$$

When reading from this address the interrupt flag will be reset and a value will be returned containing the interrupt count.

6.4 Results

The results of (Groothuis, 2004) and this tests shows an error in a cyclic pattern. This might come from drift in the controllers because the clocks are not synchronized. With the faster than real-time simulation, made possible by the direct link, the test can be repeated. It is assumed that the occurrence of errors will not occur because the controllers are synchronized by their output.

6.4.1 Simulation versus real-time HIL

Figure 30 shows the results of the simulation and the HIL setup running in real-time. The S_Pos and S_PWM are the signals in the 20 simulated model. The C_POS and C_PWM are the signals from the HIL setup running in real time. The error between the two is shown at the bottom. The results are the same as measured by (Groothuis, 2004). The error is the result of drift in the controller boards.

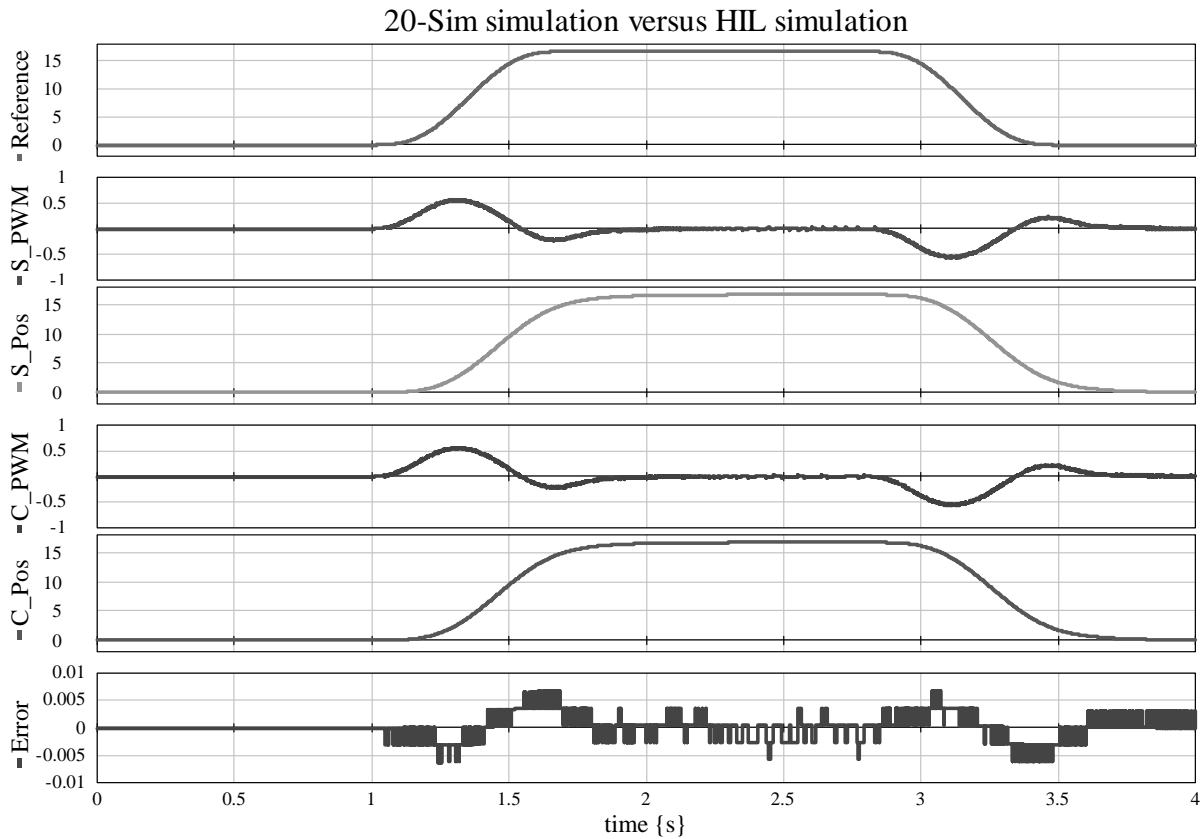


Figure 30: 20sim simulated results versus HIL

6.4.2 Simulation versus real plant

Figure 31 shows the results of the 20sim simulation and the measured results from the real plant. The S_Pos and S_PWM are the signals in the 20 simulated model. The R_POS and R_PWM are the signal measured from the real plant. The error between the actual position of linix and the simulated position is shown at the bottom. The error is larger that the error measured by (Groothuis, 2001). The cause of this larger error is in the model of the Linix. Time has a large influence on the setup, bearings wear out which causes larger resistance. Another aspect is the power supply that has to be tuned to the model. A small change in the supply settings result in a bigger or smaller error. What can be concluded is that the tool chain generates models with the same behaviour as the scripts, which was the fact to proof.

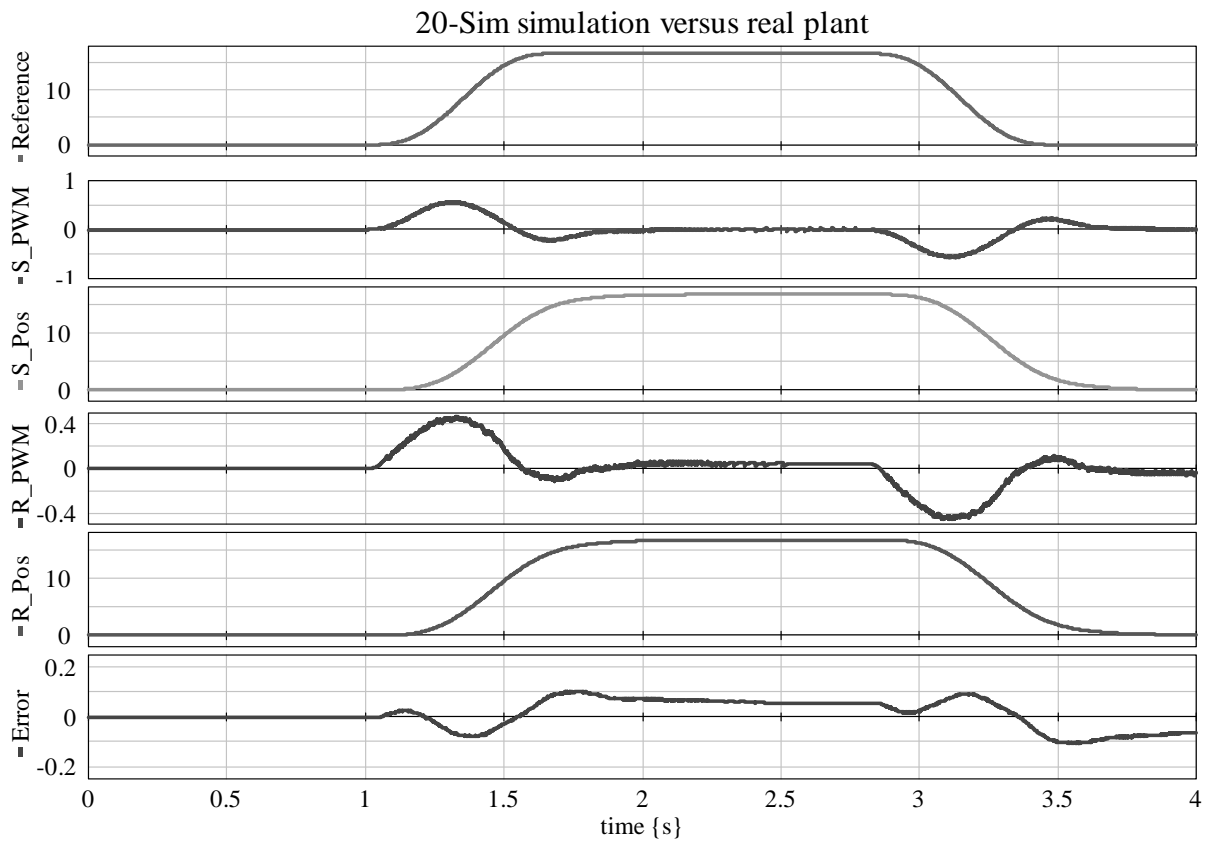


Figure 31: 20sim simulated results versus real plant

7 Conclusions and recommendations

7.1 Conclusions

Method

As a result of this project a means of approach has been introduced that can be used to generalize hardware. It is possible to connect to hardware to generated code from any application that can generate code and a list of in- and output signals into template files. The method completely abstracts the software model from real hardware. It is also possible to use the same model on different hardware and even different targets. To allow easy connection of the signals from the software to the hardware a tool has been designed and written that makes it possible to make the hardware connections.

Tool chain

The process of code generation to deployment of a task has been generalized in this project. The generalization makes it possible to build a tool chain that can perform the complete trajectory from connecting hardware to the model to deploying and controlling the task. The tool chain, that was designed and built also, makes it possible perform the complete trajectory of generating a task to deployment of that task through a graphical interface. After installation, the tools make it possible to use the mechatronic stacks and simulation PCs without any knowledge of compilers, scripts and Linux.

CCE library

The command and control library makes it possible to do rapid-prototyping with a mechatronic stack. The library is fully functional and can be used in a command and control environment. All functions, except the uploading function, are available through command prompt applications. The library can start and stop tasks on a mechatronic stack from a windows machine through a graphical user interface. Some functions, like retrieving and modifying values, have been implemented in the tool chain and can be accessed through a graphical user interface. A synchronized start mechanism has been designed and implemented to start distributed controller and HIL experiments and can be accessed through a command prompt application. Reading and writing of any variable has been implemented in the library and is available through the graphical user interface that is also used for the deployment of a task. Logging of any variable is also possible but only through a command prompt link to the library.

7.2 Recommendations

- Most of the bugs reported by testers of the hardware connector tool where XML related. To be sure the configuration files do comply with the specification an editor with syntax checking of configuration files should be designed and written.
- The hardware manager uses the same approach (replacing tokens) for parsing files as 20sim does and could be integrated into 20sim to add functionality to the real-time toolbox.
- Synchronized start now uses the Linux scheduler but this should be changed to a real-time interface to get better results (now 1ms..100us).
- When the synchronized start performance was measured it was noticed that the drift in the models on multiple stacks was significant. In order to use the stacks in a distributed setup a synchronization mechanism should be designed and implemented.
- The hardware connector parses the files ad hoc. The parameters from the configuration files are assembled and inserted into the code. The new implementation of the hardware connector should first build a hardware tree and than parse the generated tree into the tokens in the sources to get more transparency.
- Port the stack daemon to DSP to be able to use the CCE dll on a DSP board and benefit from that environment. Because a DSP lacks the use of multiple threads, the templates could be

changed, to allow the stack daemon to be compiled into the final executable and allow access through the USB port.

- Make the stack daemon Ethernet connection multithreaded to be able to connect multiple development stations to one stack daemon. This could be very useful in educational environments where more users must be able to change tasks, or variables on the stack. If more applications are build that must be connected to the stack daemon the single threaded approach is not sufficient.

Appendix I Compiling Linux based sources on MS Windows

This document describes how to build uClibc for windows. Cygwin and uClibc allows cross compiling for Linux from a windows based host.

Cygwin installation:

Get the install executable form the Cygwin page <http://www.cygwin.com> and install.

Select these extra packages to use uClibc in cygwin (if you press view in the top right corner the list will be alphabetically organized to ease selection):

- auto-make
- auto-conf
- bison
- flex
- gettext
- gettext-develop
- gcc
- make
- patch
- patch-utils
- wget

UClibc build:

Start Cygwin by using the desktop shortcut.

Make a directory opt in cygwin by entering:

```
mkdir /opt
```

Get the tarball from the UT cvs repository (Develop directory) and download it into the opt directory of cygwin. In case cygwin was installed on the C drive it will be c:\cygwin\opt. Extract the zipped file in the opt directory using your favourite compression tool and go to the sources directory in cygwin by entering:

```
cd /opt/uclibc/sources
```

To set all the flags of the files (including the execute flags) enter:

```
Chmod 777 *
```

Go up one level to get in the uclibc directory by entering:

```
cd ..
```

To start the compile process enter:

```
make
```

After a while a few options will be asked. It is advised to use the following:

- I-386 (option 7) as answer on the first question.
- Generic I386 (option 1) as answer on the second question.

The rest of the options can be left as they are (just enter Enter).

The uClibc toolchain will now be build. Go out and have lunch, get some coffee and wait... The process of compiling takes 45 minutes on a 2400 MHz AMD Athlon.

After to uClibc compilation process is completed the tool chain is in /opt/uclibc/toolchain_i386

Appendix II Building the root file system from scratch (the hard way)

This document is the guide on how to build the embedded root file system for the embedded boards. If a new file system (fs) on the embedded boards of the HIL setup is needed this document can be a guide. In case the hard way is too hard, (Groothuis, 2004) has written a set of scripts that automate these steps. A disadvantage of using scripts is that, when using open source software, configurations may change daily. In order to understand the internals and enable the reader to build the root file system, even when some small changes are made to the sources, this document is in a step by step way.

The embedded stacks use a smaller version of the GNU c libraries. In order to get applications running on the stacks a tool chain, that compiles the executables to use the smaller version, is needed. This tool chain will be the base of the root file system. All libraries needed on the file system are generated by the tool chain. After the tool chain is built the standard utilities, e.a. login and file utilities need to be built. Busy box is a single executable that behaves as most common Linux commands. The busy box executable needs to be built first. The heart of Linux, the kernel, is build next. In order to give the Linux kernel real-time capabilities a patch has to be applied to the kernel sources before building the kernel. Last but not least the RTAI kernel modules are built. The RTAI modules provide the real-time functions to the kernel.

After everything has been build, the new file system is built in a special directory. The directory will be mounted to a loop device of Linux. A loop device can make a file act as a file system. After the file system is ready, the loop device will be disconnected and the file system will be compressed to reduce disk space on the embedded device. The to be created file system will be in a directory called tmpfs.

Preparations:

Backup the old file system before building the new fs. In case anything goes wrong the old version can be restored.

A backup can be made by copying the rootfs.gz and vmlinuz from the embedded stack.

After the backup, the following packages need to be downloaded:

- uClibc buildroot (www.uclibc.org)
- Busybox (www.busybox.net) version 1.00
- Linux kernel (www.kernel.org) version 2.4.29
- RTAI (www.rtai.org) version 3.1

Tool chain:

The tool chain is part of the buildroot of uClibc. Extract the compressed uClibc file and type:

```
make
```

Select the options:

Tool chain options:

- Select Build/install c++ compiler and libstdc++

Package Selection for the target:

- Remove the busybox option (We do this manually)

Target Options:

- Remove all selection (We do this manually)

Quit and save the configuration. Now enter again:

```
make
```


After a while the process will ask some questions about the architecture. The following options must be chosen:

- C3

This will build the complete build root system for you (this may take a while!!!).

☞ If something went wrong in the menu selection, the menu can be opened again by entering:

```
make menuconfig
```

BusyBox:

Next busy box is compiled. Extract the busybox source and enter:

```
make menuconfig
```

The options that should be chosen are given below:

Install options:

- Change the Busybox installation prefix to ../tmpfs

Build options:

- Cross compiler (and set the prefix to the previously build tool chain
YOUR_DIR/buildroot/build_i386/staging_dir/bin/i386-linux-uclibc-)

Coreutils:

- Dos2unix/unix2dos

Debian Utilities:

- Run-parts
- Start-stop-daemon

Editors:

- Awk

Login/Password Management:

- getty
- login
- Use internal password...
- Support for shadow passwords
- Use busybox shadow...

Linux modules:

- Insmod
- Lsmmod
- Modprobe
- Rmmmod

Networking Utilities:

- ifupdown
- Netstat

Linux system utilities:

- Support mounting NFS file systems

Save new configuration file and enter:

```
make
```

Kernel:

We now need to compile the kernel. Extract the source to the directory. The kernel has to be patched first in order to be able to use RTAI. If you do not want to use RTAI (and not want to use any real-time applications) you can skip the patching section and directly go to the menu section.

Before we can patch the kernel the RTAI source needs to be extracted. In the RTAI source directory is a directory called patches. It is located under `rtai-core/arch/i386`. Select the patch that matches your kernel and copy it into the root directory of the kernel.

If your kernel version is not present, the patch can be modified to your kernel. Open the patch file and look for the kernel version of the patch, for the 2.4.27 version it looks something like this:

```
diff -uNrp linux-2.4.27/Makefile linux-2.4.27-adeos/Makefile
--- linux-2.4.27/Makefile      2004-08-08 01:26:07.000000000 +0200
+++ linux-2.4.27-adeos/Makefile 2004-08-14 23:58:39.000000000 +0200
@@ -1,7 +1,7 @@
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 27
-EXTRAVERSION =
+EXTRAVERSION = -adeos

KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
```

Before changing we make a copy by entering

```
cp hal16-2.4.27.patch hal16-2.4.29.patch
```

If we want to change it to 2.4.29 we have to modify

```
SUBLEVEL = 27
```

to

```
SUBLEVEL = 29
```

After changing the patch file we save it and can now apply the patch.

Go into the kernel directory and enter

```
patch -p1 < hal"your-kernel".patch
```

Check if all modifications are done. If a modification is failed, patch will tell what failed and write a '.rej' file. Most of the times it is easy to solve the patch error and perform the patch yourself.

We can now configure the kernel by typing:

```
make menuconfig
```

The following changes need to be made, all options should be built into the kernel (*) unless otherwise mentioned the module (m) should be built:

Loadable module support:

- Remove the 'Set version information on all module symbols'

Processor type and features:

- C3 Processor family
- Unselect 'Symetric multi-processing support'

General setup:

- Adeos support
- Unselect 'Power management support'

Parallel support:

- Parallel port support

Block devices:

- Loopback device support
- RAM disk support
- Default RAM disk support, set to the size needed. For example 32 MB is 32768
- Initial RAM disk support

Network device support:

- Ethernet (10 or 100 MBit) → Realtec RTL-8139 as module (M)

File systems

- DOS FAT fs support
- VFAT (Windows 95) fs support

Other options may be selected or unselected, but this is the minimal setup to get Linux running but

We can save the config again and build the kernel by entering:

```
make bzImage modules
```

RTAI:

The last thing that has to be build is RTAI. Go into the directory of RTAI and again enter:

Make menuconfig

Select and modify the following:

General:

- Set Linux source tree to your Linux source

To build RTAI enter:

```
make
```

Assembly:

Now the fun starts, all packages that are needed are built and the peaces can be assembled.

First make a clean virtual file system by entering:

```
dd if=/dev/zero of=rootfs bs=1k count=<SizeYouLike>
```

<SizeYouLike> has to be the size you specified in the kernel config, which is the size of your ram drive e.g. 16M. This command will generate a file filled with zeros and with a size will be the size you defined.

Before we can continue we have to have root rights (Enter su and give the root password or add sudo to the next instructions).

Setup one of the loop devices to the rootfs by entering

```
losetup /dev/loop0 rootfs
```

This command will connect the first loop device to the rootfs.

Make a file system on the device by entering:

```
mke2fs /dev/loop0
```

The file will now be partitioned as a file system

Create a directory that can be seen as the file system with the following command:

```
mkdir tmpfs
```

Mount the file system to the directory by entering:

```
mount /dev/loop0 tmpfs
```

The first thing that is copied to the new virtual file system is busybox. Go back to the busybox directory and enter:

```
make install
```

Copy all the libraries to the file system. This is somewhat tricky...

Go to the `buildroot/build_i386/root` directory. This directory has to be copied, with all the attributes intact to the tmpfs. This can be done by entering:

```
cp -a * ../../tmpfs
```

Due to an error in the buildroot the C++ libraries are not in the root. We have to copy them manually. Go to the `/buildroot/build_i386/staging_dir/lib` directory and enter:

```
cp -a *c++*so* ../../tmpfs/lib/
```

Now the kernel modules can be copied to the tmpfs. Go to the linux directory and enter:

```
make modules_install INSTALL_MOD_PATH=/<YourDir>/tmpfs
```

Where `<YourDir>` is the complete path to the tmpfs

Make a special dir for the rta modules by entering:

```
mkdir /<YourDir>/tmpfs/lib/modules/<your-kernel>-adeos/rta
```

Go to the rta directory and go to the modules directory. Copy the modules to the tmpfs by entering

```
cp -a * ../../tmpfs/lib/modules/2.4.29-adeos/rta/
```

The dev file systems needs several common devices. On the CVS server is a script called `mknod.sh` copy this script to your root of the build. Enter

```
./mknod.sh
```

The devices will be now be created under `tmpfs/dev`.

We now have a bootable version of an embedded system. If you would like to, you could test it. Make sure that there is another bootable Linux kernel because there is no network support yet.

To get the network running do the following: Change the `S40network` file, which is located in the `etc` directory of your temporary file system, to:

```
#!/bin/sh

./etc/sysconfig/network

# configure network
echo setup network

echo setting loop back interface
ifconfig lo up $LO_IPADDR

echo setting network interface
    ifconfig eth0 up $ETH0_IPADDR
    route add default gw $GATEWAY
```

Next make a directory called `sysconfig` and put a file called `network` into it. The network file should contain the following:

```
export HOSTNAME="<YOUR_HOST_NAME>"
export GATEWAY="<YOUR_HOST_GATEWAY_IP>"

export LO_IPADDR="127.0.0.1"

export ETH0_IPADDR="<YOUR_HOST_IP>"
export ETH0_NETMASK="255.255.255.0"
```

Because the kernel does not have the network adaptor compiled into the kernel it first has to be loaded.

Loading the network adaptor is done by installing the module. We make a new initialization file in the etc/init.d directory and call it S01modules

Create the file by entering:

```
Touch etc/init.d/S01modules
```

```
Change the flags by entering
```

```
Chmod 777 etc/init.d/S01modules
```

Now fill the file with the following:

```
#Load modules
/sbin/insmod rtai_hal
/sbin/insmod rtai_ksched
/sbin/insmod rtai_sem
/sbin/insmod rtai_shm
/sbin/insmod rtai_lxrt
/sbin/insmod mii
/sbin/insmod 8139too
```

The tmpfs can now be copied to the flash drive.

Appendix III Hardware configuration files

This appendix describes the configuration files used by the hardware connector.

Parameters:

Kind

The kind of device, e.g. FPGA, analogue or digital.

Device number

When multiple cards of a certain kind are present the user must be able to select a device id.

Configuration file

In case a file must be downloaded into the device the file must first be uploaded to the stack. This parameter contains the name of the local file that must be uploaded.

Includes

The include files that are needed in all the files that contain IO functions.

Globals

Global variables needed in all files. Variables that contain the token %handle% will be the same in all files. In the main file they will be declared and in the other files defined as external. Variables not defined as %handle% will be different in all files.

Options

Special compiler must be defined here. Special options can optimization options, but also libraries that need to be included.

Parameters:

Parameters must contain two children: Input and output

Input

The children of input contain the name of the input and the number of inputs of that kind.

Output

The children of output contain the name of the output and the number of outputs of that kind.

Startup:

Sequence

Some hardware must be initialized before opened (IO) and other hardware must be opened before it can be initialized (OI). The user must be able to select the sequence of events.

Children of startup:

Open

The open routine assigns a handle to device to be used at the reading and writing parts. The function(s) itself is put into the function attribute.

Initialize

Children of initialize:

Card

The driver sets all the in and outputs to a certain behaviour. It is also possible that a configuration file must be downloaded to the device. The function(s) itself is(are) put into the function attribute.

Channel

Some hardware has channels that need to be initialized separately. For these devices a channel initialization can be defined. The function(s) itself is(are) put into the function attribute. If the functions need to have variables the Global attribute must be filled with these values. The file where the initialize function is called is the only file these global variables appear. The global option is needed because C is used and C has the restriction that variables must be defined before the functions!

Transfer:

Scale

After reading and before handing the value to the generated code it is possible to scale the value to a certain range. With outputs the same must be possible.

Read/ Write

Reading and writing a channel have a certain function that must be executed. The 20sim generated functions must be replaced by hardware specific ones. The function attribute contains the function. The scale attribute contains the scaling function.

Shutdown:**Sequence**

Some hardware must be first closed and than reset (CR), while other hardware needs to be reset first and than closed (RC).

Reset

Bring back the card to a failsafe mode. The function attribute contains the function(s).

Close

Release all handles to the device. The function attribute contains the function(s).

Template XML document of hardware

Names between % signs can be filled by the Hardware Tool, names between \$ signs can be defined by the user.

```

<Device name>
  <Parameters DeviceID="$DevID$" Config="$FileLocation$" Includes="$IncludeFiles$"
Globals="$Variable;CommonVariable%handle%" CFlags="$CompilerFlags$"
Options="$CompilerOptions">
  <Input>
    <$TypeOfInput$ Nr="$NumberOfInputs" />
  </Input>
  <Output>
    <$TypeOfOutput$ Nr="$NumberOfOutputs" />
  </Output>
</Parameters>
<Startup sequence="$OI$" >
  <Open function="$OpenCode$" ></Open>
  <Initialize>
    <Card function="$CardInitializationCode$ %Configfile%" />
    <Channel>
      <Input Globals="$Variable$" function="$ChannelInitializationCode$
      %type% %channel%" />
      <Output Globals="$Variable$"
      function="$ChannelInitializationCode$ %type% %channel%" />
    </Channel>
  </Initialize>
</Startup>
<Transfer>
  <Read scale="$ScalingFunction$" function="Read code %type% %channel%" />
  <Write scale="$ScalingFunction$" function="Write code %type% %channel%" />
</Transfer>
<Shutdown sequence="RC" >
  <Close function="Close code" ></Close>
  <Reset function="Reset code" ></Reset>
</ Shutdown >
</Device name>

```

Tokens

The following tokens must be used in the template files that need to be processed.

%IO_INCLUDES%	Include files
%IO_GLOBALS%	Global variables
%IO_INITIALIZE%	Initialization functions. Initialization and opening of the Board as well as channel
%IO_READ_ROUTINE%	Read routines
%IO_WRITE_ROUTINE%	Write routines
%IO_CLOSE%	Closing routines. Reset as well as close

Appendix IV Command and control DLL

This appendix describes the functions and defined words used in the UTMSC DLL

```
#include <string>
```

Namespaces

```
namespace std
```

Defines

```
#define MSPFPGA 0
```

Send a PFGA program to the stack.

```
#define MSPCONTROL 1
```

Send a 20sim controller to the stack.

```
#define MSPCTC 2
```

Send a CTC program to the stack.

```
#define MSPCTCPP 3
```

Send a CTCPP program to the stack.

```
#define MSTKILL 4
```

Kill a process on the stack.

```
#define MSTSUSPEND 5
```

Suspend a process on the stack.

```
#define MSTRESUME 6
```

Resume a process on the stack.

```
#define MSGETCONF 7
```

Get the configuration of the stack.

```
#define MSPHWCONF 10
```

Send a hardware configuration to the stack.

```
#define MSGETVALS 20
```

Get values from the stack.

```
#define MSLOGVALS 25
```

Start logging values from the stack.

```
#define MSGETLOG 26
```

Get logged values from the stack.

```
#define MSSETVALS 30
```

Send values to the stack.

```
#define MSSENDSYNC 40
```

Send a synchronized start command.

```
#define KILLD 100
```

Kill the daemon.

Functions

```
EXPORT int InitMSc (char *stackName, int stackPort)
```

Initialize a connection to the mechatronic control stack at address stackName and port stackPort . The stackName may be the ip address or the logical name of the stack.

```
EXPORT char * GetConfig (int sockNr)
```

Get the configuration of the stack connected to sockNr .

```
EXPORT int ProgramMS (int sockNr, char *fileName, char *configData, int kind)
```

Send the file fileName to the stack connected to sockNr . kind is the kind of file that will be send to the cotrol stack.:

EXPORT int **ControlMS** (int sockNr, int pid, int Cmd)

Send the command Cmd to process pid on the stack connected to sockNr .

EXPORT int **GetValMS** (int sockNr, int pid, int number, int *IDs, double *values)

Get value or values of the IDs in process pid on the the stack connected to sockNr and put the in values . The buffer where the values are stored is filled with the time stamp as the first value. After the time the values will be returned. The allocated memory needs to be the number of values + 1 times the size of a double.

EXPORT int **SetValMS** (int sockNr, int pid, int number, int *IDs, double *values)

Set value or values from values of the IDs in process pid on the the stack connected to sockNr .

EXPORT int **LogValMS** (int sockNr, int pid, int number, int *IDs, int logLength)

Start logging of number IDs in process pid on the the stack connected to sockNr for a duration of logLength model ticks The buffer where the values are stored is filled with the time stamp as the first value. After the time the values will be returned. The allocated memory needs to be the number of values + 1 times the loglength times the size of a double.

EXPORT int **GetLogMS** (int sockNr, int pid, double *values)

Get the previously logged values from the stack. The DLL will allocate the memory for the value and thus the calling function has to free the buffer.

EXPORT int **SendSync** (int sockNr)

Send a sync pulse to one of the stacks. This stack will generate a CAN command to all the connected stacks to start the experiment.

EXPORT int **Killd** (int sockNr)

Kills the stackdaemon on the stack connected to sockNr .

EXPORT int **GetAnswer** (int sockNr)

Get the answer of the stack connected to sockNr on the last send command.

EXPORT int **CloseMSc** (int sockNr)

Close the connection sockNr to the mechatronic conrol stack.

Detailed Description

Function Documentation

EXPORT int CloseMSc (int sockNr)

Close the connection *sockNr* to the mechatronic conrol stack.

Parameters:

sockNr

Returns:

0 in case of succes, a value of SOCKET_ERROR in case of failure

EXPORT int ControlMS (int sockNr, int pid, int Cmd)

Send the command *Cmd* to process *pid* on the stack connected to *sockNr* .

Parameters:

sockNr, pid, Cmd

Returns:

0 in case of success, 1 in case of failure

Note:

Success of this function only tells the command has sucefully been send to the stack. Execution of the command is not guaranteed.

EXPORT int GetAnswer (int sockNr)

Get the answer of the stack connected to *sockNr* on the last send command.

Parameters:

sockNr

Returns:

The answer from the stack

EXPORT char* GetConfig (int sockNr)

Get the configuration of the stack connected to *sockNr*.

Parameters:

sockNr

Returns:

1 on succes, 0 in case of a failure

Note:

The returned pointer has been allocated by the dll.
DO NOT FORGET TO FREE THE MEMORY !!!

EXPORT int GetLogMS (int sockNr, int pid, double * values)

Get the previously logged values from the stack. The DLL will allocate the memory for the value and thus the calling function has to free the buffer.

Parameters:

sockNr, pid, values

Returns:

The number of values in case of a success, a negative value in case of an error

Note:

<note>

EXPORT int GetValMS (int sockNr, int pid, int number, int * IDs, double * values)

Get value or values of the *IDs* in process *pid* on the the stack connected to *sockNr* and put the in *values*. The buffer where the values are stored is filled with the time stamp as the first value. After the time the values will be returned. The allocated memory needs to be the number of values + 1 times the size of a double.

Parameters:

sockNr, pid, number, IDs, values

Returns:

1

Note:

Remeber to free values yourself.

EXPORT int InitMSc (char * stackName, int stackPort)

Initialize a connection to the mechatronic control stack at address *stackName* and port *stackPort*. The *stackName* may be the ip address or the logical name of the stack.

Parameters:

stackName, stackPort

Returns:

socket number on succes, -1 in case of a failure

EXPORT int Killd (int sockNr)

Kills the stackdaemon on the stack connected to *sockNr*.

Parameters:

sockNr

Returns:

0 in case of success, 1 in case of failure

EXPORT int LogValMS (int sockNr, int pid, int number, int * IDs, int logLength)

Start logging of *number* *IDs* in process *pid* on the the stack connected to *sockNr* for a duration of *logLength* model ticks The buffer where the values are stored is filled with the time stamp as the first value. After the time the values will be returned. The allocated memory needs to be the number of values + 1 times the loglength times the size of a double.

Parameters:

SockNR,pid,number,IDs,length

Returns:

1 on succes, 0 in case of a failure

Note:

<note>

EXPORT int ProgramMS (int sockNr, char * fileName, char * configData, int kind)

Send the file *fileName* to the stack connected to *sockNr* . *kind* is the kind of file that will be send to the cotrol stack:.

1: FPGA file

2: 20sim controller

3: CTC/CTCPP program

Parameters:

sockNr,fileName,configData,kind

Returns:

1 on succes, 0 in case of a failure

EXPORT int SendSync (int sockNr)

Send a sync pulse to one of the stacks. This stack will generate a CAN command to all the connected stacks to start the experiment.

Parameters:

sockNr

Returns:

Always 0

EXPORT int SetValMS (int sockNr, int pid, int number, int * IDs, double * values)

Set value or values from *values* of the *IDs* in process *pid* on the the stack connected to *sockNr* .

Parameters:

sockNr,pid,number,IDs,values

Returns:

1

Note:

Remeber to free values yourself.

Appendix V Scite, Doxygen, wxDev-CPP

For this project open source development tools were used. In this appendix is an overview of which applications were used and how they can be used.

Scite

Scite is an editor like notepad, but fully configurable. Scite is based on the SCIntilla Text Editor. It is possible to run scite on windows as well as on Linux.

Abbreviations

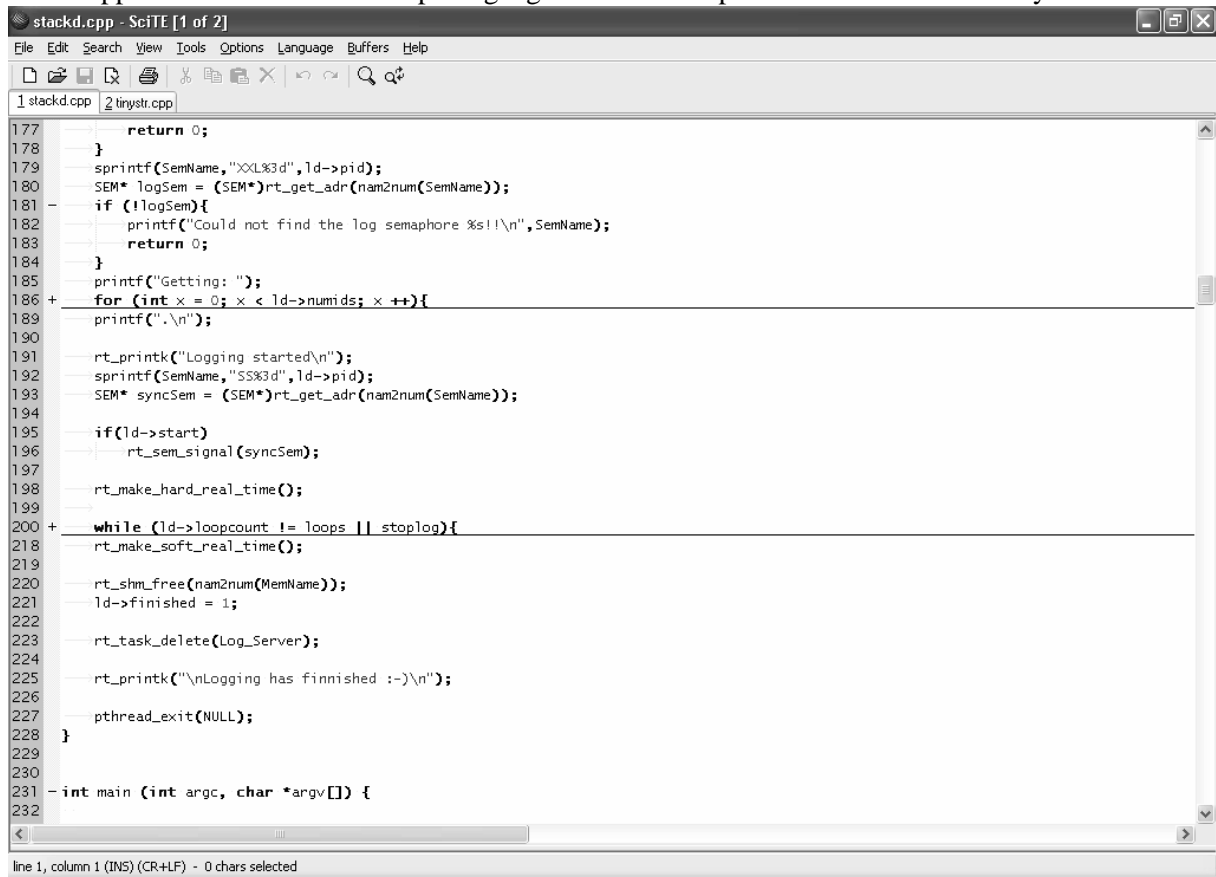
Abbreviations make it possible to translate a user defined word into an other word or series of words. With abbreviations it is possible to write code faster, because commonly used code can be put into an abbreviation.

Syntax highlighting

Syntax highlighting makes it possible to interpreted code faster because predefined words, numerical values and comments can be shown in a different colour.

User defined functions

Scite supports LUA which is a script language that makes it possible to build almost any user function.



```

stackd.cpp - SciTE [1 of 2]
File Edit Search View Tools Options Language Buffers Help
1 stackd.cpp 2 tinystr.cpp
177     return 0;
178 }
179 sprintf(SemName, "XL%3d", ld->pid);
180 SEM* logSem = (SEM*)rt_get_adr(nam2num(SemName));
181 if (!logSem){
182     printf("Could not find the log semaphore %s!\n", SemName);
183     return 0;
184 }
185 printf("Getting: ");
186 + for (int x = 0; x < ld->numids; x++){
187     printf(".\n");
188 }
189
190
191     rt_printk("Logging started\n");
192     sprintf(SemName, "SS%3d", ld->pid);
193     SEM* syncSem = (SEM*)rt_get_adr(nam2num(SemName));
194
195     if(ld->start)
196         rt_sem_signal(syncSem);
197
198     rt_make_hard_real_time();
199
200 + while (ld->loopcount != loops || stoplog){
201     rt_make_soft_real_time();
202
203     rt_shm_free(nam2num(MemName));
204     ld->finished = 1;
205
206     rt_task_delete(Log_Server);
207
208     rt_printk("\nLogging has finnishd :-)\n");
209
210     pthread_exit(NULL);
211 }
212
213
214 - int main (int argc, char *argv[]) {
215
216
217
218
219
220
221
222
223
224
225
226
227
228 }
229
230
231
232
line 1, column 1 (INS) (CR+LF) - 0 chars selected

```

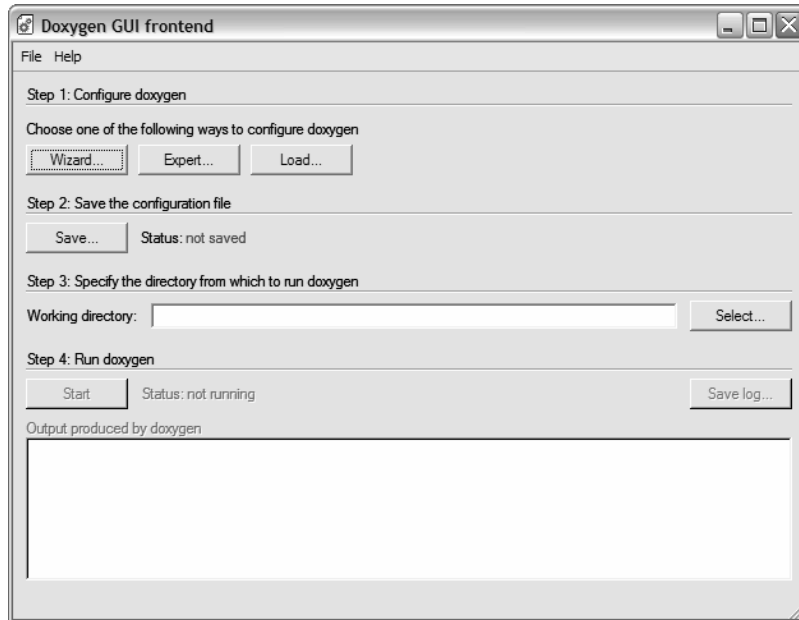
Download

<http://scintilla.sourceforge.net/SciTEDownload.html>

DoxyGen

DoxyGen is a tool that can interpret code and retrieve documentation from comments in the code. The interpreted comments can be formatted in many formats, the most commonly used are HTML, LaTeX, RTF and PDF.

Wizard



The wizard makes it possible to generate documentation without going into the doxygen configuration files. The configuration files can also be edited but a lot of knowledge of the doxygen internals is needed. The same wizard is available for Linux which makes it very easy to use on both platforms.

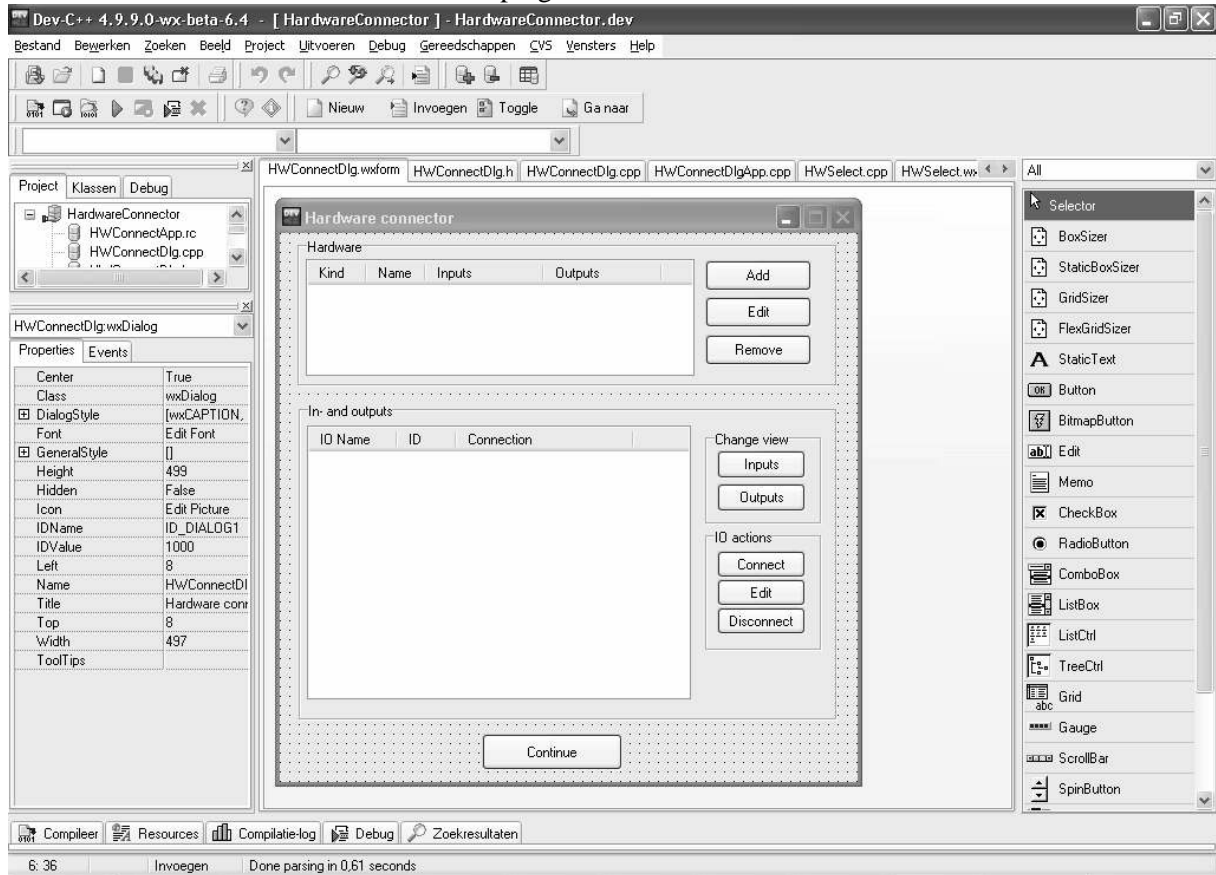
Download

<http://www.stack.nl/~dimitri/doxygen/download.html#latestsrc>

wxDev-CPP

wxDev-CPP is a free development environment that can build graphical user interfaces and runs on windows. It is a clone of Dev-CPP that runs on multiple platforms. Applications built with wxDevCPP are easy to port because wxWidgets is used. wxWidgets is a platform independent C++ library.

Below is a screen shot made while developing the hardware connector



As can be seen in the screen shot wxDevCPP has much the same look and feel of Visual C from Microsoft. For me, getting to know the environment took a real short period.

Download

<http://wxdsgn.sourceforge.net/index.html>

Appendix VI Common pitfalls

Building an embedded system is a complex task with a lot of dependencies. Because of the dependencies a fault is made very fast. In this appendix a list of common pitfalls is given in order to reduce the number of failures.

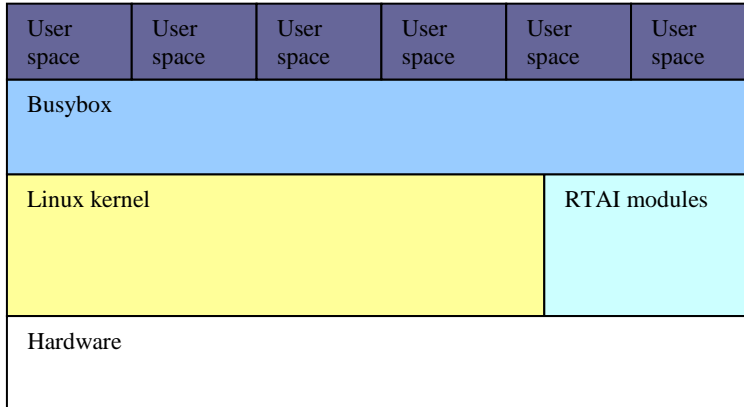


Figure 32: Layers in Linux

The tasks running on the embedded stack depend on the underlying boxes show in Figure 32. All the layers need to be build by hand and that's where the common faults are coming from.

The next section will treat every layer and point out the critical options.

Compiler

When using embedded stacks another version of the c libraries is used. The normal c libraries are in the order of tens of megabytes while the micro c libraries are in the order of megabytes. This gain in space is very valuable on embedded devices because more memory can be used for logging of data or memory requirements of final controllers can be smaller.

User space programs need to be compiled with the compiler that built the libraries installed on the embedded stack. If a version mismatch of the libraries and executable occurs the operation system will generate an unknown library error. Kernel modules do not depend on the compiler and can be compiled with any compiler. A point to look after is that the kernel includes are set to the appropriate kernel.

Kernel:

Before building the kernel, it has to be patched with RTAI. Make sure the patch is of the version of RTAI you want to use. Even small version mismatched may result in kernel panic faults.

The kernel can be compiled with any compiler.

RTAI modules:

As said before, the kernel modules of RTAI may be compiled with any compiler. The user space examples and libraries need to be compiled with the micro c compiler that has been used to build the libraries on the stack. An example of a library that has to be compiled with the embedded compiler is the LXRT library that needs to be linked with applications that use LXRT. Strange compiler errors will occur when the wrong version of compiler is used to compile the user space programs. A point to look after is that the kernel headers and normal headers point to the appropriate locations. A common fault is that the libraries are compiled against the normal Linux include directory /usr/include. Strange result will occur if this is the case.

BusyBox:

Busy box is a user space program and thus needs to be compiled with the embedded compiler. Busy box can be compiled statically and dynamically. The advantage of statically is that the embedded

station will always boot, because it does not depend on any libraries on the embedded stack. And disadvantage is that the busy box executable will be substantially larger.

User space programs

User space programs, as mentioned before, need to be compiled with the embedded compiler. A common mistake is that the program is depending on libraries and these libraries have not been compiled with the embedded compiler. Unresolved externals will be the result of this fault.

LXRT

When LXRT functions are used that do any scheduling, the task must have a real-time counterpart. A real-time counterpart is initialized by the `rt_task_init()` function. If this function has not been called and a LXRT function is called the LXRT module will crash with an OOPS. The fault is retraceable to the `get_current()` function that contains an empty pointer because the real time task does not exist.

References

- Andersen, E. (2004), *uClibc website*, <http://www.uclibc.org>.
- Bloodshed (2004), *DevCPP website*, <http://www.bloodshed.net/>.
- Broenink, J.F., G.H. Hilderink and A.W.P. Bakkers (1998), Conceptual design for controller software of mechatronic systems, *in: Proc. Lancaster Int. Workshop on Engineering Design CACSD'98*, Lancaster, United Kingdom, pp. 215-229, ISBN: 1-86220-057-2.
- Buttazo, G.C. (2002), *Hard real-time computing systems*, Kluwer academic publishers, Dordrecht, ISBN: 0-7923-9994-3.
- CLP (2002), *Controllab Products B.V.*, <http://www.20sim.com>.
- GNU (2001), *GNU C Library reference manual*, http://www.delorie.com/gnu/docs/glibc/libc.html#SEC_Top.
- Groothuis, M.A. (2001), *20-sim code generation for PC/104 target*, Individual Design Report, no 009R2001, University of Twente, Enschede.
- Groothuis, M.A. (2004), *Distributed HIL Simulation for Boderc*, MSc Thesis, no 020CE2004, Control Laboratory, University of Twente, Enschede.
- Heesch, D.v. (2005), *DoxyGen*, <http://www.stack.nl/~dimitri/doxygen/>.
- Jovanovic, D., G.H. Hilderink and J.F. Broenink (2001), Integrated Design Tool for Embedded Control Systems, *in: Progress 2001 Workshop*, F. Karelse (Ed.), Veldhoven, Netherlands, pp. 121-126, ISBN: 90-73461-26-X.
- Kathiresan, G. (2005), *wx-Devcpp*, <http://wxdsng.sourceforge.net/>.
- Mesa Electronics (2004), *Mesa Electronics*, <http://www.mesanet.com>.
- RTAI (2004), *DIAM RTAI - Real-time Application Interface*, <http://www.rtai.org>.
- Rubini, A. and J. Corbert (2001), *Linux Device Drivers, 2nd Edition*, O'Reilly and Associates Inc., Sebastopol, ISBN: 0-596-00008-1.
- Sanvido, M.A.A. (2002), *Hardware-in-the-loop simulation Framework*, PhD, Automatic Control Laboratory, ETH Zürich, Zürich, ISBN.
- Scintilla (2005), *Scintilla text editor*, <http://scintilla.sourceforge.net/>.
- SECO (2005), *M570 Manual*,
- W3C (2004), *XML*, <http://www.w3c.org/XML/>.
- Wijbrans, K.C.J. (1993), *Twente Hierarchical Embedded Systems Implementation by Simulation - a structured method for controller realization*, PhD thesis, Faculty of Electrical Engineering, University of Twente, Enschede, Netherlands, ISBN: 90-9005933-4.
- Wijbrans, K.C.J., J.v. Amerongen, A.W.P. Bakkers and J.F. Broenink (1993), Twente Hierarchical Embedded Systems Implementation by Simulation (Thesis) A structured approach to controller realisation on transputers, *J. A*, **34**, pp. 51-59.