University of Twente

EEMCS / Electrical Engineering
Control Engineering

FPGA-based control of the production cell using Handel-C

Jasper van Zuijlen

MSc report

Supervisors:
prof.dr.ir. J. van Amerongen
dr.ir. J.F. Broenink
MSc M.A. Groothuis

April 2008

Report nr. 008CE2008
Control Engineering
EE-Math-CS
University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Summary

The current platform in the Control Engineering group for developing control applications is based on the *von Neumann* architecture. However, alternative methods of implementing embedded software are available, among which are FPGAs. In the past, FPGAs were an inaccessible target, due to their limited size and hard-to-learn hardware description languages. With the introduction of the Handel-C language, FPGAs have become easier to use and, moreover, FPGAs have become bigger and cheaper. Therefore, FPGAs are becoming increasingly more interesting as an alternative in embedded system applications.

This report describes the porting of an existing, CPU based control application towards a general purpose FPGA. Due to its parallel nature, the production cell setup is used as a demonstrator. It consists of 6 parallel controlled motors which pass around blocks. The motors need to synchronize in order to operate correctly. The FPGA's biggest advantage is its ability to execute processes in true parallel. This makes the production cell a good demonstrator for this assignment.

Handel-C is based on ANSI C and is extended with CSP keywords to describe parallel processes and channels. Therefore, gCSP is an appropriate tool to design the controller framework. The framework incorporates a structural and communicational design, in which the loop-controllers reside. Loop-controllers for the production cell setup have already been designed in 20-sim, but need to be ported to Handel-C. The current loop-controllers suffer from performance degrading. When a considerable amount of blocks are in the system, it malfunctions.

A more generic framework for mechatronic systems is to be designed too, for currently no such framework is available for gCSP related designs. This could speed up the design of mechatronic systems in the future.

A general purpose FPGA does not support the floating point data-type, so a deviation from the standard data-type is made. In order to port the current, floating point based controllers, the integer data-type is used instead. This introduces two significant challenges among others:

- Porting of the motion profiles;
- Porting of the PID algorithm.

The structural and communications framework needs to comply to several requirements, implying the model to be distributable, de-centralized, CSP based and generic where possible. Every motor is assigned its own sub-system which contains enough intelligence to operate on its own, and is called a production cell unit. Each production cell unit includes functionality for safety handling and user input. The requirement of using only the FPGA implied the usage of the integer data type for calculations. This made the process of algorithm implementation time consuming, and needs to be simplified in the future. This is due to limitations in the current design tools.

The framework designed to implement the loop-controllers on a setup proves to be useful. It is applicable in any mechatronic setup and provides for all functionality required of a control system. The framework provides an easy way to communicate between units.

The outcome of this assignment shows that the FPGA is suited for basic embedded control. However, when designing for FPGAs, keep accuracy limitations of the integer data-type in mind.

Samenvatting

De huidige platformen voor het ontwikkelen van regelapplicaties, binnen de Conrtol Engineering group, zijn gebaseerd op de *von Neumann* architectuur. Alternatieve methoden voor het implementeren van embedded software zijn echter beschikbaar, waaronder FPGAs. In het verleden waren FPGAs een ontoegankelijke technologie vanwege de gelimiteerde grootte en moeilijk te leren hardware omschrijvingstalen. Met de introductie van Handel-C zijn FPGAs gemakkelijker te gebruiken en, belangrijker, FPGAs zijn toegenomen in grootte en gedaald in prijs. Om deze reden zijn FPGAs een steeds aantrekkelijker alternatief om te gebruiken in embedded systeemapplicaties.

Dit rapport beschrijft het omzetten van een bestaande, CPU gebaseerde, regelapplicatie naar een general purpose FPGA. Door zijn parallelle eigenschappen is de productiecel geselecteerd als demonstratieopstelling. De opstelling bestaat uit 6 parallel geregelde motors welke blokjes doorgeven. De motoren moeten hun bewegingen synchroniseren om correct te werken. Het grootste voordeel van de FPGA is de eigenschap dat deze parallel processen kan verwerken. Deze combinatie maakt de productiecel een perfecte demonstratieopstelling.

Handel-C is gebaseerd op ANSI C en uitgebreid met CSP sleutelwoorden om parallelle processen en kanalen te beschrijven. gCSP is daarom een geschikt gereedschap om het regelraamwerk mee te ontwerpen. Het raamwerk omvat structuur en communicatie waarin de regelaars zich bevinden. De regelaars voor de productiecel zijn al ontworpen met behulp van 20-sim, maar moeten worden omgezet naar Handel-C.

De huidige regelaars lijden aan prestatievermindering. Als er veel blokjes in het systeem zijn, gaat de opstelling slecht werken.

Ook moet er een meer generiek raamwerk voor mechatronische systemen ontwerpen worden. Momenteel bestaat zoiets niet voor gCSP gebaseerde ontwerpen. Dit zou het ontwerp van toekomstige mechatronische opstellingen kunnen versnellen.

Een general purpose FPGA ondersteund het drijvende punt datatype niet. Daarom moet er worden afgeweken van dit standaard datatype. Om de bestaande, drijvende punt gebaseerde ‚regelaars om te zetten, wordt gebruik gemaakt van het integer datatype. Dit brengt enkeleenkele uitdagingen met zich mee:

- Het omzetten van de bewegingsprofielen;
- Het omzetten van het PID algoritme.

Het structuur- en communicatieraamwerk moet voldoen aan enkele voorwaarden, stellende dat het model distribueerbaar, gedecentraliseerd, CSP gebaseerd en generiek waar mogelijk, moet zijn. Iedere motor heeft zijn eigen subsysteem welke genoeg intelligentie bevat om autonoom te opereren; een productieceleenheid. Iedere eenheid bevat bovendien veiligheids- en gebruikersbedieningsopties. De eis dat slechts en alleen de FPGA gebruikt mag worden, leidde tot het gebruik van het integer datatype voor berekeningen. Mede hierdoor nam het implementeren van de algoritmes veel tijd in beslag, iets wat verbeterd moet worden in de toekomst. Oorzaak hiervan zijn de beperkingen van het huidige gereedschap.

Het raamwerk, ontworpen om regelaars te implementeren op een opstelling, bewijst zijn diensten. Het is toepasbaar op ieder mechatronische opstelling en voorziet in alle gestelde functionaliteit. Het voorziet in een eenvoudige manier van communiceren tussen eenheden.

Het resultaat van de opdracht toont aan dat de FPGA geschikt is voor eenvoudige regelapplicaties. Hou echter bij het ontwerpen voor FPGAs rekening met de limitaties van het integer datatype.

Preface

With this report I conclude my education at the University of Twente.

It has been with great pleasure, that I did this assignment. However, I could not have done it without the help and support of several people, who I want to thank here.

Jan Broenink and Marcel Groothuis, I want to thank you for your guidance and support during my assignment.

I want to thank Paul Weustink for his illuminating ideas on embedded PID control.

In the positive critique and discussions section, I want to thank my fellow Msc weekly students. You helped me a lot by questioning my own decisions, which brought my work to a higher standard.

Also, thank you Marcel Schwirtz and Geert Jan Laanstra for your technical support. This project would have taken far more time, if it weren't for you!

I want to thank my parents, for giving me the opportunity to study at the University of Twente.

Finally, I want to thank my girlfriend, Marloes Blaauw, for standing by me and helping me write my reports, proofread and edit them.

All of the people I did not mention here, but did help me during my study, my thanks go out to you.

Jasper van Zuijlen

Utrecht, April 2008

Contents

1 Introduction

Current control applications within the control engineering group are oriented toward the *Von Neumann* architecture (Silberschatz et al., 2005). An alternative to this architecture are FPGAs. In the past, the limited size and hard-to-learn *hardware description* languages made the FPGA an inaccessible device. Nowadays, the FPGAs are becoming cheaper and bigger and easier to use, with the introduction of the Handel-C language (Celoxica, 2005).

This project uses a demonstrator in order to test the suitability of a general purpose FPGA as control platform. The production cell setup (van den Berg, 2006) is selected due to its parallel nature. It consists of 6 parallel controlled motors, which need to synchronize in order to operate correctly. Since the most important benefit of FPGAs is the true parallel execution of processes, the production cell makes a perfect demonstrator for this assignment.

This assignment is based on the outcome of several other projects within the ViewCorrect PhD project of Marcel Groothuis (Groothuis, 2008). It tends to solve the load related problems of the controller by Maljaars (2006) for the production cell setup (van den Berg, 2006). The chessway project (Kuppeveld, 2007) inspired to port the existing controllers to an FPGA. Experience within the Stan Ackermans Institute led to a feasibility study (van Zuijlen, 2007), which recommended the Handel-C language to be used in control applications.

1.1 Aim of this project

The FPGA is selected as a control platform because of several platform characteristics.

- *Parallel execution* : FPGAs execute processes in hardware, enabling true parallel execution of processes.
- *Hard real-time* : Each process is present in hardware when the FPGA starts executing. Every process always takes the exact same amount of time to execute, unaffected by the number of parallel processes.
- *High speed* : Because every process consists of dedicated hardware, high computational speeds can be obtained.

Currently the FPGA is used as an IO board, implementing PWM generators and encoder readers and other digital IO. Since the FPGA is able to do much more, a thorough examination of the FPGA is done in this assignment. This assignment aims to implement a full control application on a single, general purpose FPGA, including loop-controllers, structural design, communication and digital IO.

The production cell's current controllers are designed by Maljaars (2006). These controllers suffer from performance degrading when all six controllers are operational. This assignment aims to improve that behaviour, while still meeting the loop-controller requirements set by Maljaars. The software based gCSP models used by Maljaars were not applicable for this assignment. A new, more generic framework needs to be designed.

A general purpose FPGA also has some downsides. The floating point datatype, commonly used in embedded control, is not natively available. This can be overcome by adjusting the loop-controllers. Also, an FPGA has no easy debugging method available. Possibly, an external tool needs to be written, in order to log and monitor the setup.

The outcome of the examination should provide information on the possibilities to include the FPGA in the design methodology. The current flow of integration (figure 1.1) uses modeling tools, such as gCSP and 20-sim, which can generate code. Currently, ANSI C (Kernighan and Ritchie, 1988) is used as code medium. The 4C tool (Visser et al., 2007) then couples the ANSI C code to the embedded target.

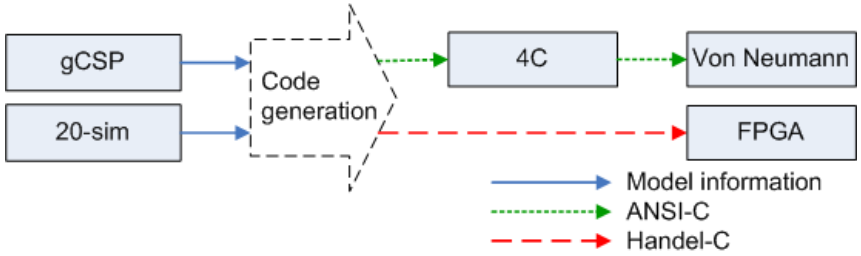


FIGURE 1.1 - Flow of integration for embedded control applications

In order to integrate FPGAs as a target within this work flow, an FPGA specific code medium is needed. An ANSI C based language is preferred, due to existing support. gCSP uses CSP to describe parallel processes, so CSP support is also useful. To this end Handel-C is selected, Handel-C is based on ANSI C and extended with CSP in order to describe parallel processes.

1.2 Design methodology

Not all phases of the design methodology (figure 1.2) are treated in this assignment. Phase [1] and [2] where already done by Maljaars (2006) using 20-sim and gCSP. This work can be reused in this assignment, however, the existing controllers need to be adjusted to the FPGA platform. Phase [3] involves the implementation of phase [1] and [2], which is the first part of this assignment. This phase is realised using the SDK of Handel-C, the *DK design suite*.

The realisation phase [4] can be split in two subjects: the realisation of the setup and the realisation of the controllers. The realisation of the setup has been done by van den Berg (2006). The realisation of the controllers makes up the second part of this assignment.

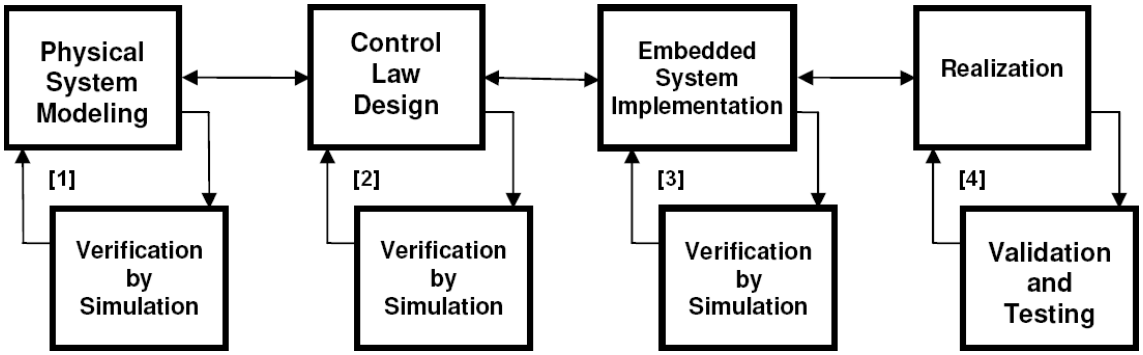


FIGURE 1.2 - CE design methodology (Broenink and Hilderink, 2001)

The tools used in this assignment are listed below:

- *20-sim* : In order to simulate the setup and design the controllers, 20-sim is used. The controllers created by Maljaars and the plant model of the setup already exist in 20-sim format. This should provide a fast and accurate way to design the Handel-C controllers.
- *Handel-C* : Handel-C and the associated development environment, the DK design suite, is used as the primary hardware description language in this project.
- *gCSP* : gCSP is used as a code generation tool, it can model parallel processes and generate the appropriate Handel-C code. Currently Handel-C code generation is not available in gCSP, so it needs to be implemented.
- *4C* : This tool enables 20-sim to directly connect to the production cell. 4C can control and log the setup, enabling fast problem solving and validation of the controllers.

These tools provide for the design, implementation and validation. The validation plan is formulated in the next section.

1.3 Evaluation

In order to evaluate the results obtained from this assignment, some topics need to be investigated during the design and implementation.

- *Maintain controller quality* : The FPGA based controllers should still meet the requirements stated in the report of Maljaars (2006). In order to prove this, the FPGA controlled position of the setup should be measured and compared to the simulated plant position and the desired position.
- *Check synchronisation* : In order to evaluate the quality of the framework, the synchronization between the controllers needs to be tested. This can be done by a durability test on the setup.
- *Deadlock checks* : Since gCSP will be used to model the framework, a deadlock check should be done on the system.
- *FPGA size* : The amount of functionality is limited by the size of the FPGA. An important point of investigation is what the approximate size of an FPGA should be, to incorporate the complete design.

1.4 Report structure

Background information on this assignment is provided in chapter two. The design and adjustments of the loop controller can be found in chapter three. It shows the difficulties of porting a floating point based application to an integer based platform. Chapter four shows how the resulting loop-controllers can be connected using a framework for structure and communication. Finally, the fifth chapter lists the conclusions which can be drawn from this report. The recommendations for future work and improvements can be found in this chapter as well.

Appendix A describes the tools used to monitor and debug the application. These tools are also applicable for demonstration purposes. The design of the PWM generators for effective motor control are printed in appendix B. The gCSP models can be found in appendix C, while the IO connections of every production cell unit are in appendix D. The pinout of the FPGA connectors is shown here as well. Lastly, the list of available PCI addresses can be found in appendix E. Details on FPGA usage of the selected FPGA can be found in appendix F.

2 Background

This chapter covers the background of this assignment; the used platform, programming method, setup and platform limitation solutions are elaborated. It should provide enough information, for those new or unfamiliar to this subject, to understand the rest of this report.

2.1 FPGA

The FPGA, or *field programmable gate array*, is a programmable logic device, evolved from more simple devices like the PLD, PAL and PLA (Valk, 1997). It is an answer to the search of a programmable logic device that does not have the vast prototyping costs like *application specific integrated circuits* (ASICs).

2.1.1 Technology

An FPGA (figure 2.1) consists of an array of identical blocks, called *configurable logic blocks* (CLBs) or *slices*, which are the smallest units in an FPGA. The CLB is used as a measure of the size of the FPGA. Another measure is the number of gates. The number of gates is correlated with the number of CLBs, however the exact number of gates per CLB differs per FPGA manufacturer and device type.

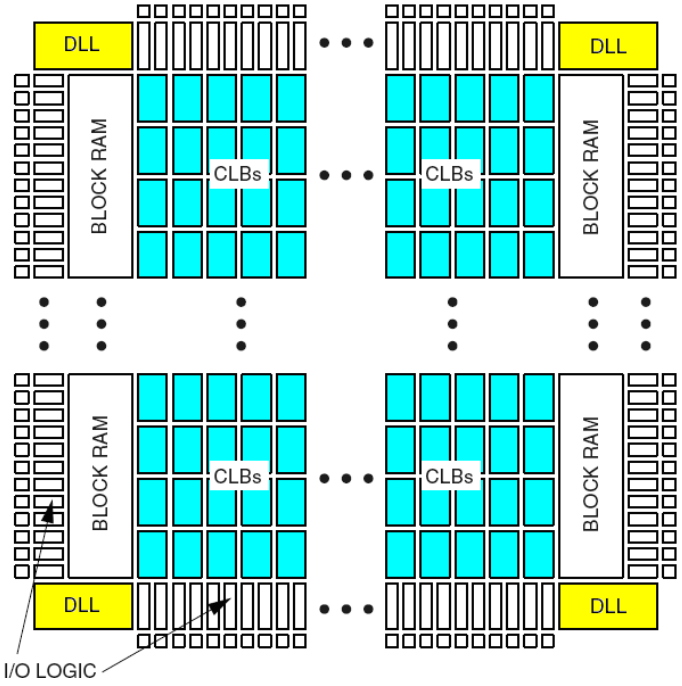


FIGURE 2.1 - A schematic diagram of an FPGA (Xilinx, 2008)

A typical CLB contains at least one *logic cell* (LC), which is defined as one flip-flop (serving as memory), a *lookup table* (LUT) and carry logic and is connected to a global clock (figure 2.2).

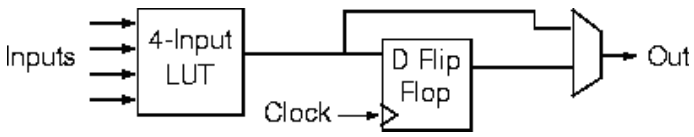


FIGURE 2.2 - A typical LC (Xilinx, 2008)

Multiple CLBs are connected through a grid, creating a two dimensional array of logic cells (figure 2.1). This forms an array of gates which can be of virtually any size.

2.1.2 Methods of use

The FPGA is used in two ways:

1 *ASIC* : As an application-specific integrated circuit when the *time-to-market* (TTM) is too short to develop a dedicated IC.
2 *Prototyping* : In the case of a longer TTM, the FPGA is the ideal-low risk prototyping platform before issuing an order for a batch of dedicated ASICs.

Some popular applications of FPGAs are listed below:

- *Combinatorial circuits*: One FPGA with a lot of different functions, i.e. replace multiple simpler logic ICs (like 74xx series ICs).
- *Glue Logic*: Using the FPGA to form a bridge between two devices, i.e. a hardware protocol gateway or just to link some signals to others as a switch.
- *FPGA- Based Computing engines*: This configures the FPGA as a dedicated computer. These systems are very fast and supply true multitasking through making separate pieces of hardware for a single instruction. For a detailed description see, (Vranesic et al., 1992) pages 8 and 9.
- *Hardware accelerated computing* : An FPGA can also be connected to a CPU, in order to aid the CPU. A nice example is the manticore (Mrochuk and Carson, 2008) project, where a *graphics processing unit* (GPU) is implemented on an FPGA.

2.1.3 Soft-cores, IP-cores

Some configurations, mainly ASICs, Glue Logic and Computer engines, are available via open source or are for sale. These products can be used as building blocks are called 'soft-cores' or '*intellectual property* (IP)-cores' and usually require only a common FPGA. Soft-cores can be used in an existing project, in order to speed up development time or to reduce the costs.

2.1.4 Hard-cores

Certain FPGAs embed other hardware on the FPGA chip, i.e. PowerPCs and *digital signal processors* (DSPs), which can be addressed using logic busses and registers on the FPGA. This way, a designer can use both FPGA and CPU/DSP specific techniques to obtain the best results for the designed application.

2.1.5 FPGA implementations comparing to software

FPGAs have several benefits and limitations when, compared to software applications.

- Benefits:
  - All processes are present as dedicated hardware;
  - True parallel execution of processes;
  - Reconfigurable hardware; an FPGA could be a PowerPC on one instant and a signal processing unit the next (and back again) (Smit et al., 2008);
  - No load-related performance issues. If an FPGA is configured, it will continue to do its job at a constant rate, independent of the system load;
  - Virtually infinite number of parallel processes (limited by the size of the FPGA);
  - Hard real-time: every process always takes the same amount of clock cycles to complete.
- Limitations:
  - General purpose FPGAs provide no floating point support, other FPGAs only limited;
  - An FPGA has a limited amount of logic cells to represent an application.

This makes FPGAs very suitable for applications were the functionality is bounded, such as a control application, and less suited for applications were functionality changes rapidly, such as a computer desktop environment.

2.1.6 Concurrent versus Parallel

One of the major benefits of an FPGA is its true parallel execution of processes. A common misunderstanding in computer terminology is the difference between concurrent and parallel. For example: a single core computer can run multiple processes concurrently. This means that

every process is granted a limited amount of run-time on a CPU, according to a scheduling algorithm (Tanenbaum and Woodhull, 1987). Effectively, only one process is physically running on a CPU at a time. However, the switching of the processes is usually very fast such that the user *perceives* the execution of the multiple processes as being simultaneous.

Current computers can embody one or more processor chips, containing one or multiple cores. These computers *can* run multiple processes at the same time (parallel), but only as much as there are processor cores in the system.

An FPGA can run as much parallel as fits in one device. Theoretically this would be an infinite number of processes, only limited by the size of the FPGA.

2.2 Handel-C

Handel-C is an ANSI C (Kernighan and Ritchie, 1988) based hardware description language, extended with CSP (Hoare, 1985) keywords. It is used as the *hardware description language* (HDL) in this assignment. This chapter briefly explains the origins of the language, as well as the main differences between Handel-C and ANSI C.

2.2.1 History

Handel-C was born out of the idea to create a way to map Occam programs onto an FPGA (Page and Luk, 1991). Around 1990 Ian Page created a programming model with a construct-by- construct mapping of a subset of Occam to hardware with the addition of timing semantics. This programming model was called 'Handel' after the famous composer (Spivey and Page, 1993). The strongest qualities of this model are in particular the:

- Single clock assignment;
- 'par' construct;
- Parallel implementation of expression trees.

This model allows for a very wide range of hardware implementations to be described within a single framework. In 1992 the Handel programming model was extended with a front end parser which mapped from a C-like concrete syntax into Handel. This language was called Handel-C, though nothing alike the current Handel-C implementation.

The first stand-alone, concrete syntax, Handel-C compiler was written by Matt Aubrey, a member of Page's Hardware Compilation Research Group at Oxford (Page, 1998).

In order to commercialize the Handel-C language, Ian Page founded the company *Celoxica* (2008), which exploited the Handel-C development environment up onto January 2008. Since then, the development environment and language have been taken over by *Catalytic* (2008).

2.2.2 Syntax

The syntax of Handel-C is much like ANSI C, but lacks floating point arithmetic. CSP keywords such as *par*, *seq* and *prialt* were added to the syntax to implement parallel related behaviour. As an example, a typical contruct for two parallel processes is shown in listing 2.1.

```
par {
    proc1();
    proc2();
}
```

LISTING 2.1 - Two parallel processes

Other major differences are in the variable declarations. In Handel-C every variable has a user definable width. In ANSI C the width of a variable is defined by either the language definition or the platform. For instance, an integer (int) is 32 bits on a 32-bits platform, 64 bits on a 64-bit platform and an ASCII character (char) is always one byte (8 bits).

In Handel-C everything is defined as an integer (signed or unsigned) with variable width.

A typical variable declaration in Handel-C would be:

```
int 24 aVar;
```

LISTING 2.2 - Variable declaration in Handel-C

This declares a signed variable `aVar`, 24 bits wide.

2.2.3 Timing

As opposed to regular HDL, such as VHDL, Handel-C does not have an explicit notion of time. Rather, it incorporates this simple rule:

> "Assignment and delay take one clock cycle. Everything else is free."
> (Celoxica, 2005)

This means, that only if a value is assigned to a variable or if the delay statement is used, one clock cycle passes. This also means that multiple assignments are not allowed on one line. These so called *side effects*, as shown in listing 2.3, are *not* allowed.

```
foo = bar++;
```

LISTING 2.3 - An example of side effects, not allowed in Handel-C

In this example the value of `bar` is assigned to `foo` (taking one clock cycle) and `bar` is raised by the value of one. The latter should also cost one clock cycle as the value of 1 is assigned to `bar`. However, since both assignments are on one line they should only take one clock cycle, but this is not supported. Listing 2.4 shows an example of an assignment of `foo` by `bar`, while `bar` is raised by 1 in the same clock cycle.

```
par {
    foo = bar;
    bar++;
}
```

LISTING 2.4 - Simultaneous assignment and incrementation

2.2.4 Motivation of use

This concludes the brief introduction of Handel-C. More information on the Handel-C HDL can be found in the Handel-C reference guide (Celoxica, 2005) and in a feasibility study on Handel-C for embedded control (van Zuijlen, 2007). This report describes the choice of Handel-C over SystemC, as well as other important aspects for the choice of Handel-C.

2.3 Production cell

The production cell setup by van den Berg (2006) was designed to resemble a real machine consisting of several devices that operate in parallel. It also had to be suitable for distributed control. Furthermore, the setup had to allow for safety implementations, but failures in this safety or the controller should not result in any mechanical damage. Ultimately, the setup had to be suitable to serve as a demonstrator.

The setup consists of 6 axis that operate simultaneously and need to synchronise to pass along metal blocks. In this report, each of these axis are called *production cell units* (PCUs). Each PCU is named after its dominant function in the system, shown in figure 2.3.

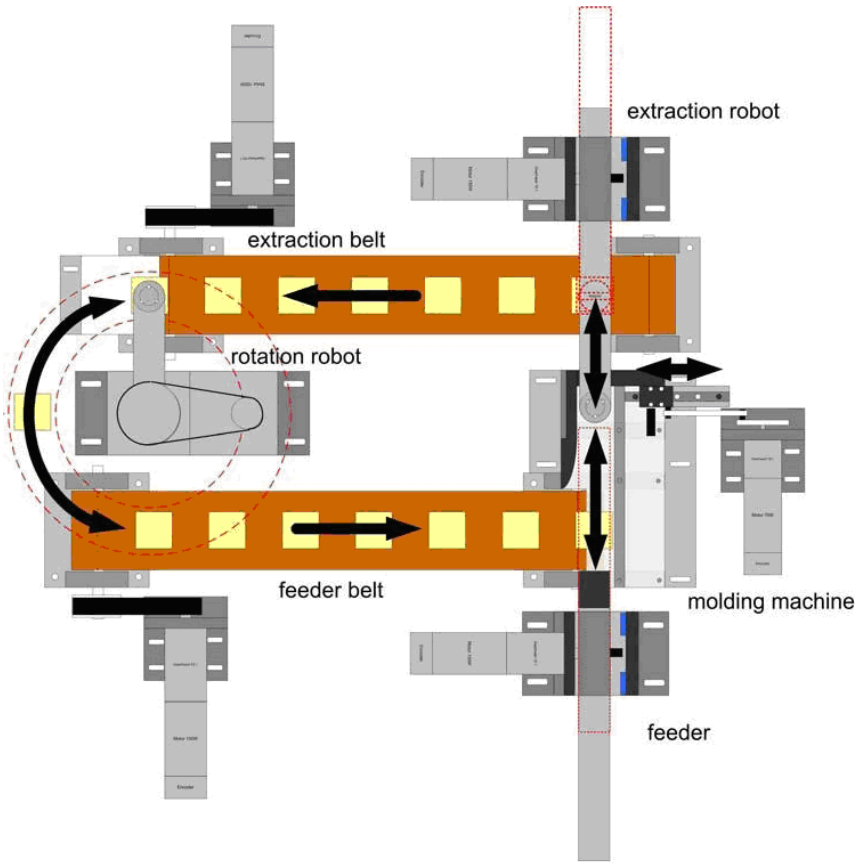


FIGURE 2.3 - The production cell setup

The operation sequence begins by inserting a metal block at the feeder belt. This causes the feeder belt to transport the block to the feeder which, in turn, pushes the block against the closed molder door. At this point, the actual molding would take place. After this, the feeder retracts and the molder door opens. This frees the way for the extraction robot, which can now extract the block from the molder. The block is placed on the extraction belt, which transports it to the rotation robot. Finally, the rotation robot picks up the block from the extraction belt and puts it on the feeder belt. Now the cycle starts again.

The belts allow for multiple blocks to be buffered, such that every PCU can be provided with a block at all times, allowing the setup to operate all axis simultaneously. The blocks are picked up using an electromagnet at the end of the extraction robot and the rotation robot.

2.3.1 Previous work

Several other software based solutions have been made to control the production cell, as shown in table 2.1.

Who	Method	Loop-controller design
van den Berg (2006)	Time table based	20-sim
Maljaars (2006)	gCSP	20-sim
Huang et al. (2007)	POOSL	20-sim
Orlic (2007)	SystemCSP	Not implemented

TABLE 2.1 - Currently existing controllers

All the controllers in table 2.1 are CPU based and therefore use floating point as a numerical standard. Van den Berg (2006) uses his own loop controllers, Huang et al. (2007) uses the loop controllers from Maljaars (2006), Orlic (2007) has not implemented loop controllers. The important differences, however, lie within the design of the software structure.

2.3.2 Current status

Two of the controllers of table 2.1 are still in use: the gCSP and the POOSL based controllers. However, both controllers have a few shortcomings.

The gCSP (Jovanovic et al., 2004) solution operates correctly in real-time mode for low CPU load. However, when more blocks are inserted and more PCUs requests CPU time, the load rapidly increases. When all 6 PCUs are operating, the CPU load exceeds 100% and so the loop controllers are starved. This results in malfunctioning of the setup.

The POOSL (Theelen et al., 2007) based solution is interpreter based. The interpreter is a normal OS program, not a real-time process and therefore the timing is not guaranteed when the load of the system is high.

Both controllers do not implement safety. This assignment aims to improve the controller behaviour, design safety functionality and allow for distributed control.

2.4 Floating point alternatives

As mentioned before, the FPGA does not natively support floating point. Since the loop controllers are based on floating point arithmetic, alternative mathematical models have to be investigated.

This section merely lists the available options to this assignment, in terms of floating point alternatives for Handel-C based FPGA design. Chapter three goes deeper into the material by comparing the options described here and selects one of the options as numerical representation for this project.

The alternatives can be split into two groups: FPGA based methods and off-FPGA based solutions.

2.4.1 FPGA based alternatives

There are roughly three FPGA based alternatives (table 2.2). All are oriented toward different numerical representations.

Representation	Implementation	Precision	Logic cell utilisation
Floating point	IP core / library	High	High
Fixed point	IP core / library	Moderate	Moderate
Integer	Native	Low	Low

TABLE 2.2 - Numerical comparison chart

Both floating point and fixed point are not natively supported in an FPGA and the mathematical models must therefore be added as an IP core. Handel-C provides a library for both the floating point format and the fixed point format. Therefore, all three alternatives in table 2.2 are considered as an option.

The ratio of precision versus logic cell utilisation is linked, which means a higher numerical precision uses more logic cells. As a result, a larger FPGA is needed when using higher precision compared to lower accuracy.

Fixed point and integer lose the most accuracy with the division operation. This is due to the limited fractional representation of fixed point and the lack of fractional representation in integer (Cooling, 2003). The accuracy in dividing can be improved by reducing the significance of fractional numbers (Karapetian, 2006). This can be achieved by up-scaling the input values for instance:

- Using a higher encoder precision;
- Smaller quantization steps of PWM values.

2.4.2 Off-FPGA based alternatives

So far, only FPGA based solutions were examined. Other options include using FPGAs with added functionality on-chip or even using additional devices to aid the FPGA.

Some FPGA devices incorporate *floating point units* (FPUs) as a hard-core on chip. Also, some off-chip alternatives exist to implement floating point (FP). Some more application related options are considered as well.

In this section only the alternatives are listed. Section 3.2 describes the choice of the alternative from the available options.

	Option	**Nature**
[1]	Hard-core FPU	Floating point math
[2]	Off-chip FPU	Floating point math
[3]	Motion profiles RAM/ROM lookup table	Reduced FP math
[4]	Remove derivatives in motion profiles	Reduced FP math
[5]	Simplify motion profiles	Reduced FP math

TABLE 2.3 - off-FPGA alternatives

Option [1] and [2] of table 2.3 are oriented toward floating point math. Option [1] these proposes the use of an on-chip FPU. Some FPGAs contain one or more hard-core processors (i.e. IBM PowerPC), which contain FPUs that can be used by the FPGA section of the chip for calculations.

Option [2] involves interfacing the FPGA with an external floating point able device, i.e. a CPU or stand-alone FPU. This way, the FPGA can outsource the calculations and save on LCs.

Finally, the motion profiles used in the controller could be simplified or altered, options [3], [4] and [5]. Since the motion profiles are created in the design phase, there is no need to explicitly generate and derive them on the FPGA. As an alternative, the motion profiles can be stored off-chip in RAM/ROM (option [3]). The FPGA would then use a bus to obtain the values of a certain motion profile from the RAM/ROM as a lookup table.

Maljaars' (2006) motion profiles are generated as a single positional signal. Velocity and acceleration are implicitly defined as derivatives of this positional signal. Deriving these signals is a computationally intensive job and could be replaced by explicitly defining the velocity and acceleration signals. The separate signals can then be further optimized.

2.5 Conclusions

This chapter treated the characteristics of the control platform, its limitations and propositions on how to overcome these limitations. It also introduced the demonstrator setup, and the associated existing controllers on which this assignment relies. Finally, some information on which code medium was selected to describe the behaviour of the controllers and the framework was given: the Handel-C hardware description language.

The next chapter describes the design, implementation and validation of the loop-controllers. It also elaborates on which floating point alternative(s) are chosen, in order to overcome the platform's limitations.

3 Loop controller redesign for FPGA usage

The core of this assignment is to implement loop-controllers for the production cell in Handel-C. The design for these loop-controllers has already been done in the past by Maljaars (2006) and they are used as a source for this assignment. Since the Maljaars controllers are CPU based, some adjustments have to be made on the existing loop-controllers. This chapter describes the validation of Maljaars' controllers and the conversion of the loop-controller and motion profiles from the CPU platform to the FPGA platform.

3.1 Starting point

Maljaars' (2006) controllers were designed using a model of a continuous time plant and a discrete time controller (figure 3.1). Digital IO connects the controller to the plant. It contains PWM generators and encoder readers among others. In Maljaars' case, the discrete time controller is realised on a CPU and the digital IO is implemented on an FPGA. This assignment implements both the discrete time controller and the digital IO on a single general purpose FPGA. In this chapter only the extraction PCU is shown. The other PCU's are configured in a similar way.

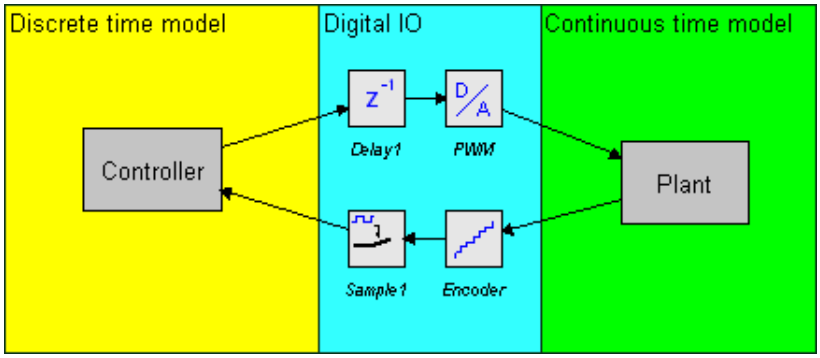


FIGURE 3.1 - Top level of the controller

3.1.1 Current implementation

The loop-controllers consist of two elements: a loop-controller and a motion profile generator, which provides the setpoints (figure 3.2). The plant model is shown in figure 3.3. Note that the H-bridge model is a duty cycle to motor current converter. The linear system in the H-bride contains the motor current and motor resistance transfer.

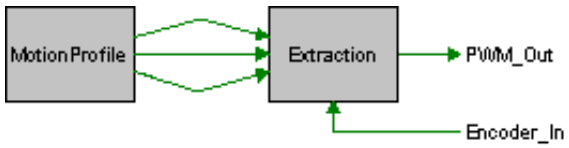


FIGURE 3.2 - Contents of the controller

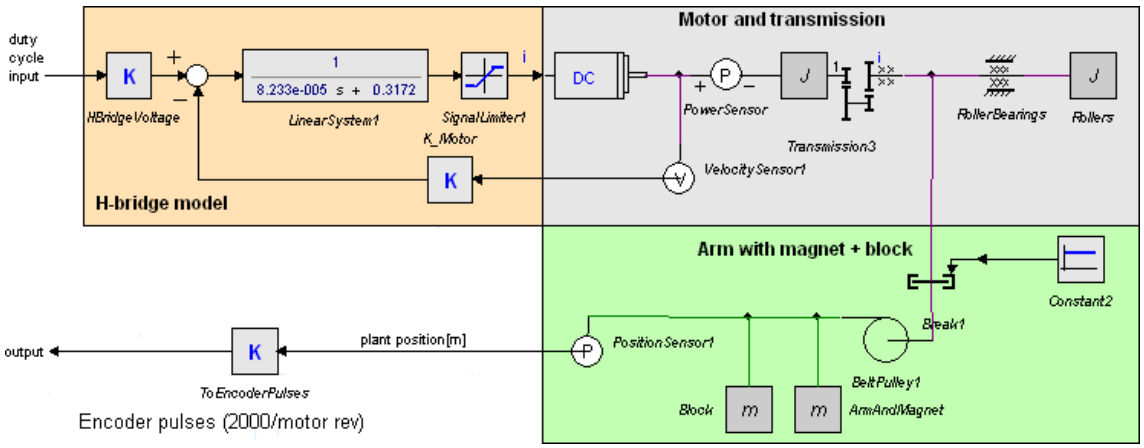


FIGURE 3.3 - Plant model extraction robot(Maljaars, 2006)

The motion profile block generates 3 signals: position, speed and acceleration. Both speed and acceleration are computed by means of time derivatives of the position. A typical motion profile signal, with its associated derivatives, is shown in figure 3.4, where FF means *feedforward*.

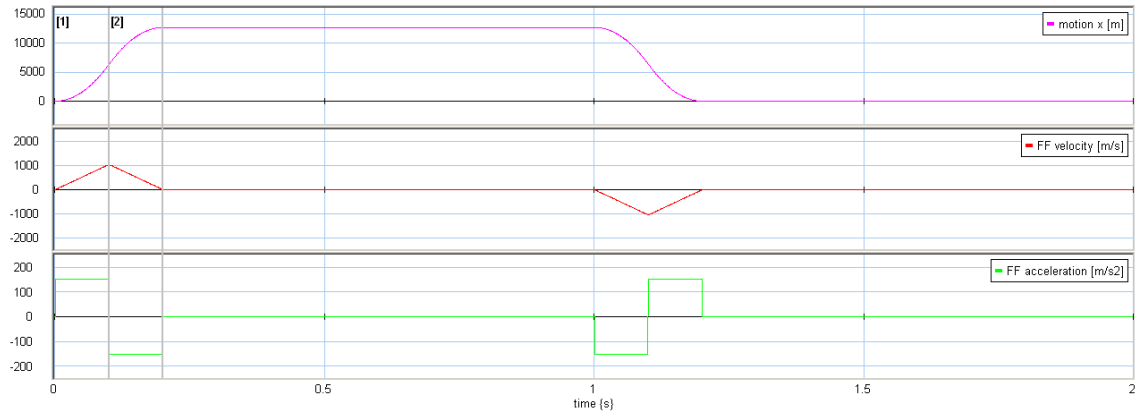


FIGURE 3.4 - Motion profile signals

The position signal in figure 3.4 indicates the arm going forward, holding its position for a while and then going backward. The linearly increasing and decreasing velocity ensures smooth movement. In [1], it causes the arm to slowly pick up speed. At [2], the arm slows down by linearly decreasing speed, reaching the end of the track with low speed. The motion profile indicates the desired motion. The loop-controller, as designed by Maljaars, is based on a PID controller with feed forward, as shown figure 3.5.

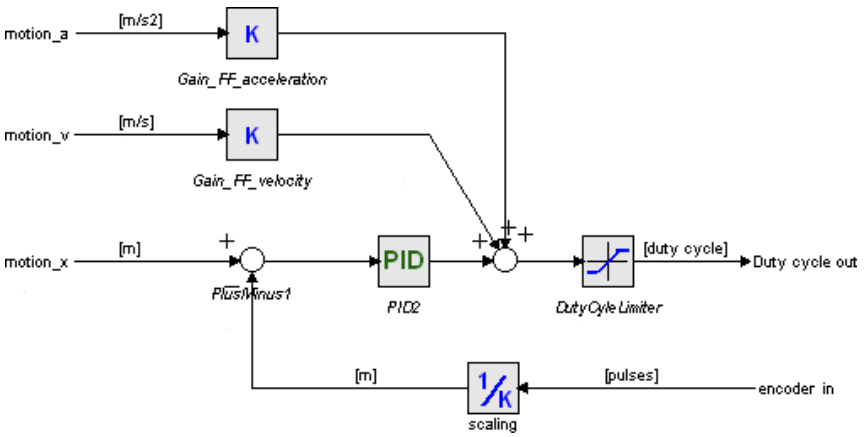


FIGURE 3.5 - PID controller with feed forward

The controller, in figure 3.5, uses the motion profile signals and the motion position (encoder) as input. The output is the duty cycle signal send to the plant. The PID controller uses the position signal of the motion profile as a setpoint for the PID. The real position of the plant — which is converted from encoder position into meters by using the 'scaling' block — is then subtracted from the motion profile position.

The resulting value is added to the attenuated speed and acceleration signals from the motion profile. The report of Maljaars provides more information on the actual values of the PID and attenuation signals. The 'DutyCycleLimiter' block ensures that the outputted value does not exceed the duty cycle extremes.

Maljaars showed that the controllers operate well in simulation and in reality. To confirm that this is still the case, despite wear and tear, the system of controller and plant is validated again.

3.1.2 Validation

In order to check if the current controller and plant are valid, the simulation results are compared to the output of the real setup. This involves a simulation using the model as discussed before, and another run of the controllers, this time replacing the simulated plant by the real setup. The results of the validation are shown in figure 3.6.

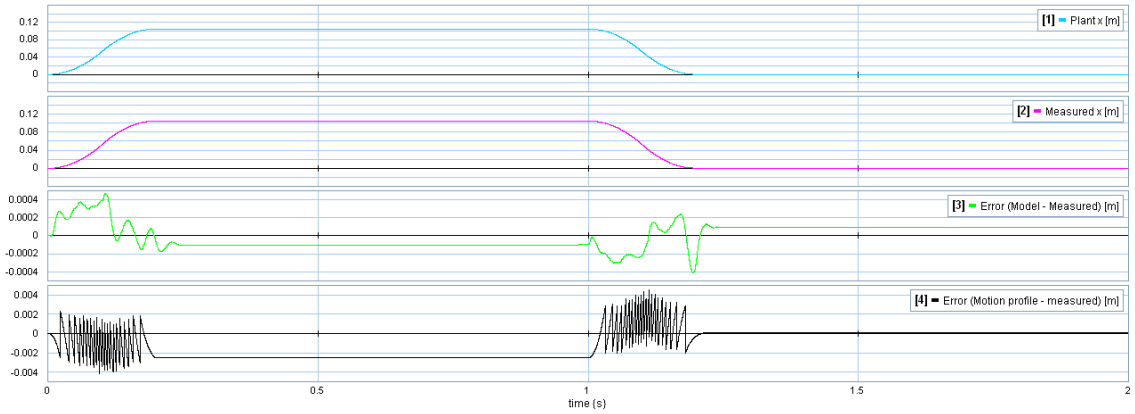


FIGURE 3.6 - Validation of the Maljaars controllers

3.1.3 Interpretation of the validation results

Signal [1] shows the simulated plant position, signal [2] shows the measured plant position. The error between signals [1] and [2] is shown by signal [3]. Signal [4] shows the difference between the motion profile position and the position of the real setup.

Signal [3] in figure 3.6 shows the difference between the model and the real setup. This error shows the quality of the model. This error is between -0.0004 m and 0.0004 m or -0.4 mm and 0.4 mm. Signal [4] shows the error of the desired position versus the actual position; the absolute error. The extremes of this signal are -0.004 m and 0.004 m or -4 mm to 4 mm, which is within the requirements of Maljaars. This proves that the Maljaars controllers are still valid and suitable to use in this assignment.

3.2 Choosing a floating point alternative

The Maljaars controllers are CPU and floating point based. As mentioned in section 2.1.5, the floating point data-type is not available on the selected FPGA.

3.2.1 Floating point unit alternatives

Section 2.4 introduced some alternatives to use floating point. The options introduced in this section are summarized in table 3.1.

	Option	Location	Benefit	Drawback
[1]	Floating point	On-chip	High precision	Very high logic utilisation
[2]	Fixed point	On-chip	Acceptable precision	High logic utilisation
[3]	Integer	On-chip	Native datatype	Low precision in small ranges
[4]	Hard-core FPU	On-die (Off-FPGA)	High precision, loop-controller native	Only available on special FPGA, high-price
[5]	Off-chip FPU	Off-chip	High precision, loop-controller native	Involves CPU scheduling
[6]	Motion profiles lookup table	Off-chip	Fast solution, low on FPGA utilisation	Requires interfacing a RAM/ROM module
[7]	Remove derivatives in motion profiles (define explicit)	Algorithm	Simplifying calculations, resulting in lower FPGA utilisation	Needs new 20-sim motion profile implementations
[8]	Simplify motion profiles	Algorithm	Simplifying calculations, resulting in lower FPGA utilisation	Lower precision

TABLE 3.1 - Alternatives for using the floating point data type

The first two options, floating point [1] and fixed point [2], can be implemented using the Celoxica floating point and fixed point libraries (Celoxica, 2005; van Zuijlen, 2007). Both data types provide enough accuracy for the application (this was tested with 20-sim, results are omitted). However, when porting a single motion profile from 20-sim to the FPGA, the results were too big for any FPGA on the market today. The average amount of utilisation was around 400% on a 200k gate Xilinx Spartan II. Since the setup contains 6 motors, it requires 6 motion profiles and 6 controllers. The requirement is set to use a single FPGA. Consequently, a $4 \times 2 \times 6 = 48$ times bigger FPGA is needed, which are currently non-existent. Possible solutions for this problem are CPU scheduling techniques. For example, the motors could reuse the loop-controller, reducing the number of loop-controllers to one. This would undermine the whole concept of this assignment (true parallel processing) however. Therefore, these points are left for further study.

Option [3] involves using integers as a base for calculations. This could be considered as a fixed point data-type with zero fractional bits. This data type is native to the FPGA and utilizes little logic cells. The drawback of this data type is the low precision of calculations, when the range of a signal is small, due to the absence of decimal representation. The setup of this assignment, however, uses high resolution encoders, so the precision should be large enough.

Option [4] involves using an FPGA with a FPU on die. Similar tests to the first two options were done on this solution. The available FPGAs (Xilinx Virtex series) where still over-mapped around 250% so this is not an option. The high price of these FPGAs are another downside.

Option [5] and [6] involve outsourcing (parts of) the control algorithm. Since this assignment aims on implementing a controller system on an FPGA, these options are ignored.

3.3 Conclusion

Given the available FPGA, option [3] was selected. To use the integer data-type, optimizations

[7] and [8] are needed to ensure a proper implementation. This choice elaborated in the next section.

3.4 Using integer based control

The choice of the FPGA native integer data type introduces some challenges. Since the current control design is heavily based on floating point, some implementations need to be revised.

3.4.1 Porting of the motion profiles

20-sims motion profile algorithm heavily leans on the floating point format to generate the position signal. Also, the velocity and acceleration signals are derived explicitly from the position signal. These algorithms take up a lot of FPGA space. In order to tackle these challenges the following steps are undertaken.

- Restyle the floating point based position signal by linear approximation (option [7] from table 3.1);
- De-couple the velocity and acceleration signals from the position signal by separate computation (option [8] from table 3.1).

The linear approximation technique involves dividing the existing position profile into a finite number of straight lines. Figure 3.7 illustrates this procedure. For clarity, the approximation is shown for the second part of the motion profile position signal, starting at 0.15 seconds. In the controller, the whole motion profile position signal is approximated.

The conversion enables the motion profiles positional signal to be represented in integer format. Two downsides of this technique and their related loop-controller effects, are:

- Added high-frequency noise due to the transitions
  Effect : Lower D-action to compensate for the added high-frequency noise
- Indication of movement instead of a real setpoint
  Effect : Lower or eliminate I-action to prevent overshoot due to I-buffer overload

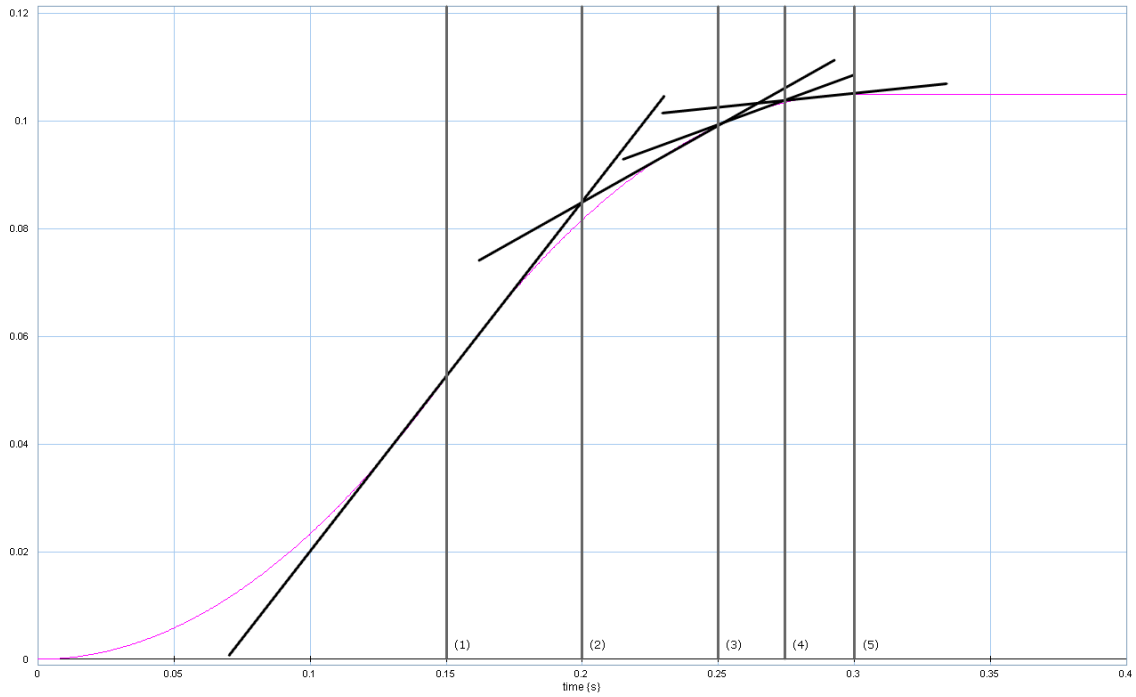


FIGURE 3.7 - Linear approximation of the position signal

3.4.2 Porting of the PID controller

The PID controller is based on a computer algorithm (Åström and Hagglund, 1995) and also relies on the floating point datatype (listing 3.1). To make the algorithm suitable for integer based control, the following conversions are necessary:

- Integer based parameters;
- Integer based math.

```
factor = 1 / ( sampletime + tauD * beta );
uD = factor * ( tauD * previous ( uD ) * beta + tauD * kp * ( error -
                    previous ( error )) + sampletime * kp * error );
uI = previous( uI ) +  sampletime * uD / tauI;
output = uI + uD;
```

LISTING 3.1 - The PID loop-controller algorithm

20-sim offers the integer datatype, but has a floating point based calculation engine. Currently, this leads to incorrect mathematical behaviour. The line in listing 3.1 describing `factor` for instance, is calculated as shown in listing 3.2, while the correct implementation to mimic integer based calculations is shown in listing 3.3.

```
factor = truncate (1.0 / (sampletime + tauD * beta));
```

LISTING 3.2 - 20-sims faulty implementation of integer math

```
factor = truncate (1.0 / (truncate (sampletime + truncate (tauD * beta))))
```

LISTING 3.3 - Correct implementation of integer math

Therefore, the integer based PID algorithm needs to be implemented using a different technique. Two options are available to obtain this:

- Port the current model to Simulink with the fixed point toolbox;
- Use an external DLL, which uses the integer data-type, with 20-sim.

The second option also provides an easy way to validate the controller after design by means of the — software-in-the-loop — 4C toolchain (Visser et al., 2007). This enables the comparison between 20-sim simulation generated ANSI C implementation and the Handel-C implementation. Therefore, the second approach is selected for the implementation of the simulated controller.

As mentioned in the previous section, the I-action proved to cause oscillations in the response. This is due to the fact that the plant cannot, and should not, exactly track the approximated position profile. This causes the I-buffer to charge too much, resulting in an instable response. Therefore, the I-action is eliminated and the resulting controller is a PD controller.

3.4.3 Validation

To validate the controllers, the controlled position of the plant was measured. The measured position signal is compared to the simulated plant and the motion profile position signal. The results are depicted in figure 3.8.

The differences between the plant and the motion profile are small, as in the original controllers. The difference between the measured position and the simulated position are still within the 5mm bounds, as required in Maljaars (2006). Also, the difference between the desired position of the motion profile and the measured position remain within the 5mm boundary. The stationary error of the controller is good enough for the production cell setup: 1mm deviation from the desired stationary position.

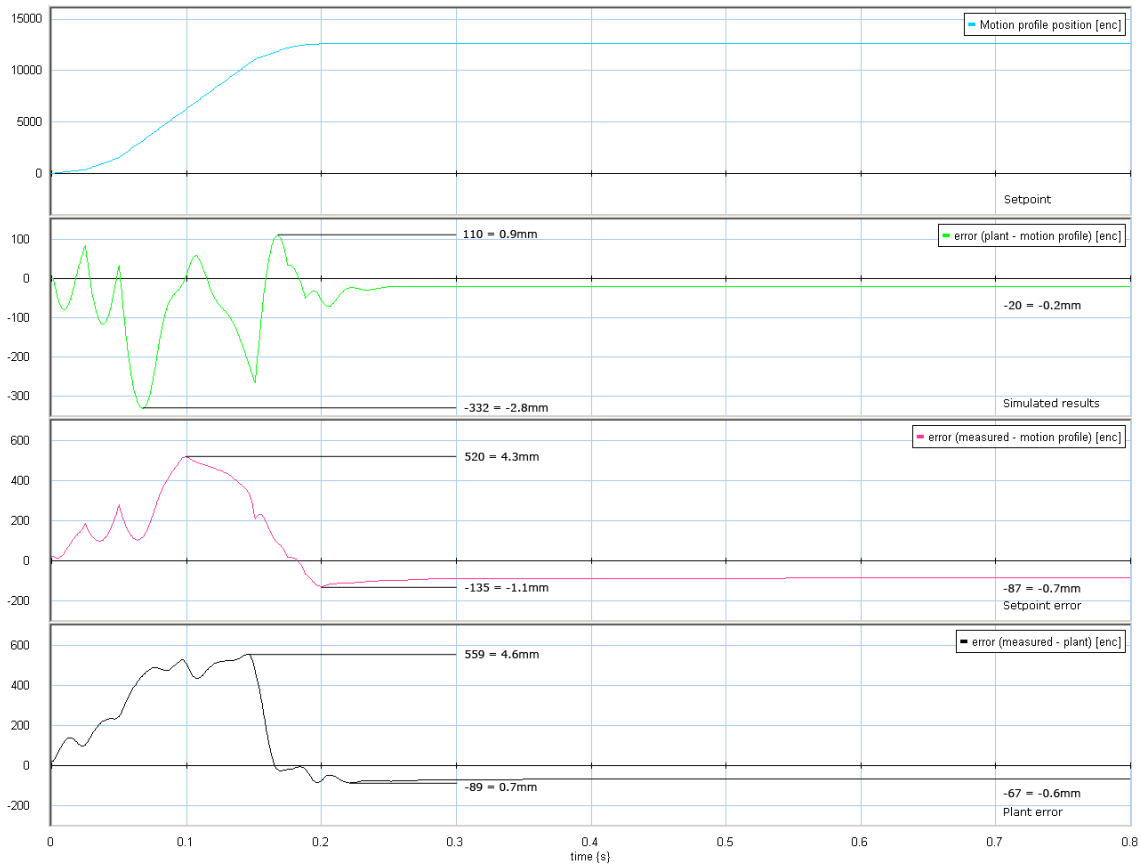


FIGURE 3.8 - Validation of the Handel-C based controllers

3.5 Conclusion

The porting of the floating point based controllers has been proved successful. The resulting motion profile and the PD based loop-controller still complies with the requirements stated by Maljaars. However, the integer based controllers involve a lot of manual labour, in order to operate. Some work still lies in making the whole process more automated. 20-sim currently does not support integer based applications , so another tool could be chosen to do this work better.

4 Structure and communication

This chapter describes the design and implementation of the structural and communication (S&C) framework, used to connect the various loop-controllers. This framework will be made generic such that future work can reuse this framework, whether the target platform is CPU based, FPGA based or otherwise.

The communication sequence for this setup is designed as well, along with a rudimentary example on how to use a safety layer. The actual implementation of a safety layer is out of the scope of this assignment and remains future work.

4.1 Requirements

As mentioned in section 2.3, the production cell setup is designed for distributed control. Hence, this should be incorporated in the S&C framework. The term *distributed* implies that every unit of the setup must contain enough intelligence to operate independently, rather than having a central point of intelligence. Furthermore, since Handel-C is based on CSP, it is sensible to use a CSP based design method. Finally, the framework must be reusable for other setups.

Summarizing the above points, the model should be:

- Distributable;
- De-centralized;
- CSP based;
- Generic (where possible).

4.2 Top level design

The requirements mentioned in the previous section provide sufficient information to design the top-level model. Every controller should be self sustaining and can therefore be modeled as a single unit. These units are called *production cell units* (PCUs).

Since the production cell setup has a preferred direction — explicitly: *feeder* > *molder-door* > *extractor* > *extraction belt* > *rotation* > *feeder belt* > *feeder* — normal communication is only necessary with the next PCU. This communication can be as simple as a 'ready' signal which, in CSP terms, is best implemented using a rendezvous channel. The normal communicational flow will be called the 'happy flow' of the system.

When a failure occurs, communication with both neighbours is desired. For instance, when the feeder is stuck, not only should the molder-door be opened; also the feeder belt should be stopped, in order to halt the flow of new material (blocks).

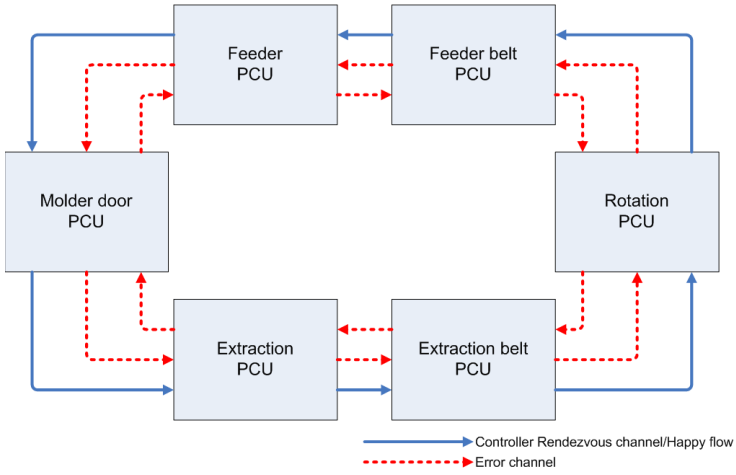


FIGURE 4.1 - Top level design

Considering all previously mentioned requirements and communicational flows, the top-level

diagram is designed (figure 4.1). The top-level design, couples all the PCUs together. The implementation of the PCUs is described in the next section.

4.3 PCUs

If every PCU is to operate independently, it needs to know its state, control its motion (or speed) and check if it is still functioning properly. Every PCU can be seen as a single control system. Such a system should contain the following contents (Bennet, 1988; Bennet and Linkins, 1984):

- Man-machine interface;
- Supervisory control;
- Sequence control;
- Loop control;
- Data analysis;
- Measurements & actuation.

These topics will be called the *demanded contents of a controller*.

Due to the nature of this setup, supervisory control is not appropriate here; every PCU is able to control itself. Based on the rest of the demanded controller contents and several other popular implementations (Bruyninckx, 2000; Bennet, 1988; Bennet and Linkins, 1984), a generic model is made for all PCUs (figure 4.2).

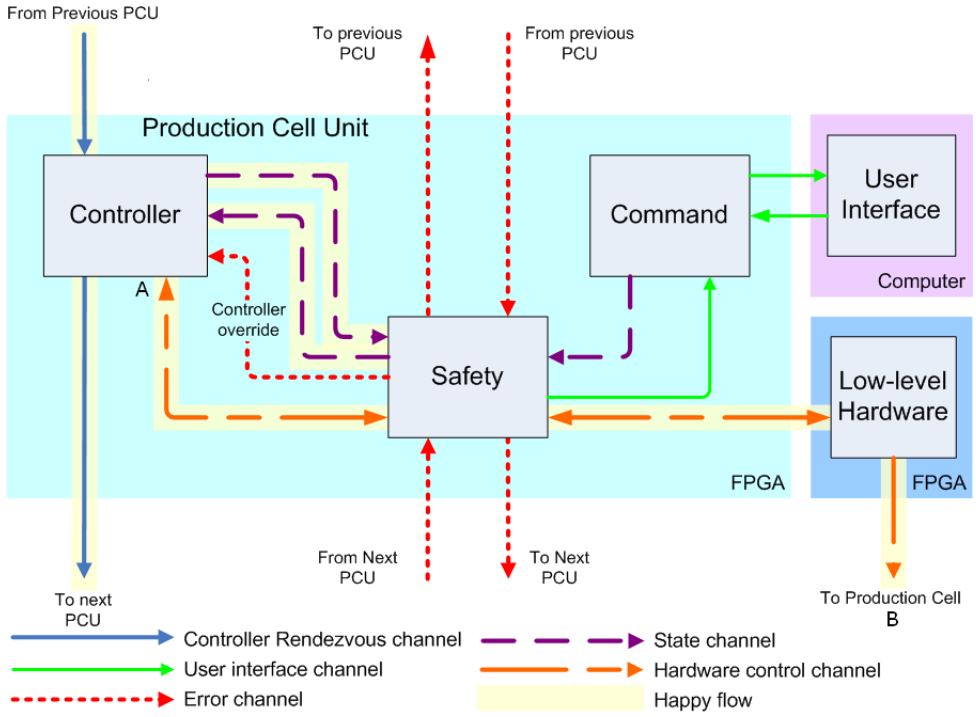


FIGURE 4.2 - Model of a production cell unit

Figure 4.2 shows a generic, distributable, de-centralized model of a PCU. It is implemented using several sub-blocks and communication channels.

Table 4.1 shows how the demanded contents of the controller are mapped onto the sub-blocks.

Sub-block:	**Contains:**
Controller	Sequence control Loop control
Safety	Data analysis + safety
Command	Man-machine interface
Low-level hardware	Measurements & actuation

TABLE 4.1 - Mappings of the demanded contents of a controller

Finally, the two communicational flows, both happy flow and error flow, are routed through the various sub-blocks, as shown in figure 4.2. The happy flow, which is highlighted, enters the controller and passes through the safety before re-entering the controller and finally addressing the next PCU. This implies that the happy flow is safely controlled.

The error flow passes through the safety only. This way the handling of non-safe situations is centralized within the PCU.

4.4 Safety

The implementation of the safety layer is guided by the general architecture of protection systems (Lee and Anderson, 1990) and is based on the work of Wijbrans (1993). The safety consists of three stages: the *exception catcher*, the *exception handler* and the *state handler* (figure 4.3).

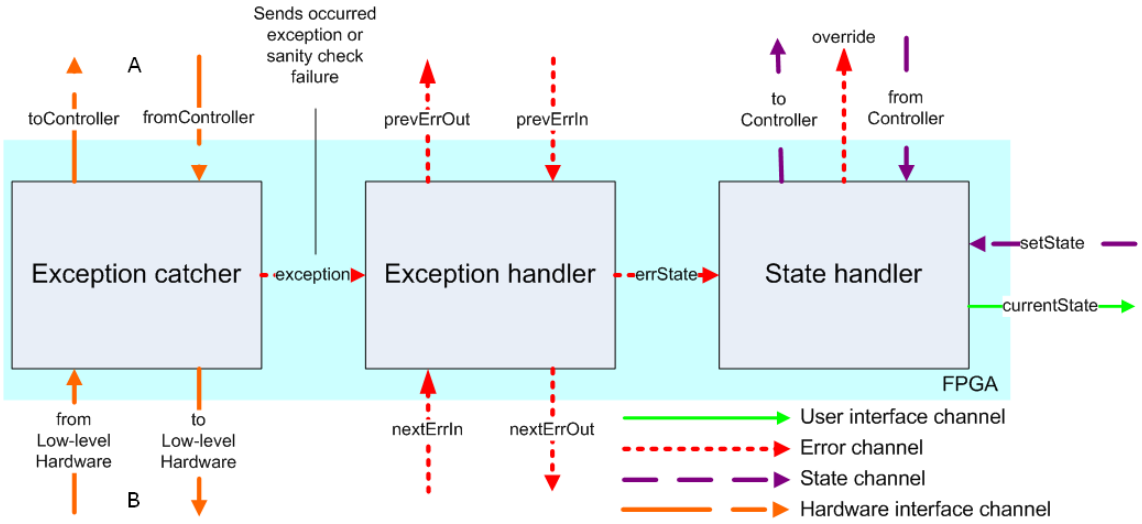


FIGURE 4.3 - Safety layer implementation

The *exception catcher* catches the exception (B to A) as well as the sanity check failures (A to B). The block generates an error message toward the *exception handler*. The *exception handler*, in turn, converts the error message into three logical states:

- Its own (safe) controller logical state via the *errState* channel;
- A safe controller logical state for the previous PCU in the chain;
- A safe controller logical state for the next PCU in the chain.

The *state handler* controls the logical states in a PCU and is the link between the happy flow and the error flow. Here the decision is made what logical state is being sent to the controller block (figure 4.2). It receives logical state information of three sources:

1 Exception handler;
2 User interface;
3 Controller block.

The order in the list above indicates the priority of the channel. The channel which is written first, is granted permission and that state is then copied to the *controller* and the user interface.

The highest priority channel is the *errState* logical state channel. This channel transports the safe logical state from the exception handler to the *state handler* when a failure has occurred. Once this channel is activated, this logical state will always be sent to the *controller* (figure 4.2). The *override* channel is activated as well, in order to keep the neighboured PCU in its logical state until the error has been resolved.

4.5 Controller

Each *controller block* (figure 4.4) contains the rules to control one of the six axis of the production cell. The *controller* is operated by the *sequence controller*, which communicates its logical states with the *state handler* (figure 4.3).

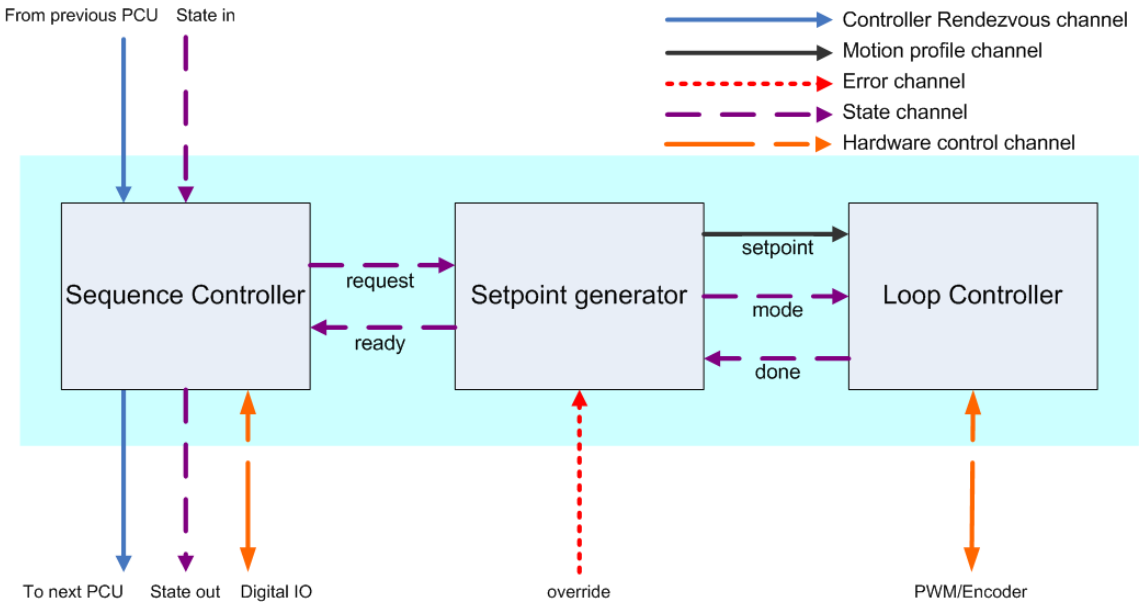


FIGURE 4.4 - PCU Controller

The *sequence controller* is the brain of the controller and acts on digital sensor inputs, generates the timing dependent behaviour of the PCU and controls the *setpoint generator*. The *setpoint generator* contains setpoints for the stationary positions of the PCU and is able to generate motion profiles to move between those stationary positions. The *loop controller* receives setpoints from the generator. Dependent on the mode set by the *setpoint generator*, it is able to run the:

- Homing profile;
- Regulator control algorithm (to maintain a stationary position);
- Servo control algorithm (to track motion profiles).

The homing profile mode is necessary because of the used motor encoders, which are incremental. This means that at start-up, the motor position is undefined. The homing profile brings the motor to a known position. After this, the encoder can be used to determine the position of the motor.

Both the *loop controller* and the *sequence controller* are connected to hardware control channels (resembles CT linkdrivers (Groothuis, 2004)). These channels interface the underlying hardware of the setup through the safety layer.

The diagram, shown in fiure 4.4 completes the S&C framework. The next section describes how the PCUs can communicate, using the framework as a communications network.

4.6 Sequence diagrams

The designed framework is now capable of safely controlling each PCU and is equipped with the necessary communication to other production cell units. The last design phase is to design handshaking, communicational and start up protocols between the PCUs.

4.6.1 Happy flow

When the setup starts up, all translational PCUs execute a homing action to bring the PCU to a defined initial position. Belts are velocity controlled and do not have an initial position; these PCUs do not need a homing action.

When the homing action is completed, the setup is idle until a new block is introduced to one of the belts.

Figure 4.5 shows the communications of the setup between all PCUs, starting from the point were a block is introduced to the *feeder belt*. A question mark in this figure indicates a query. A block can also be introduced to the extraction belt which would result in similar behaviour.

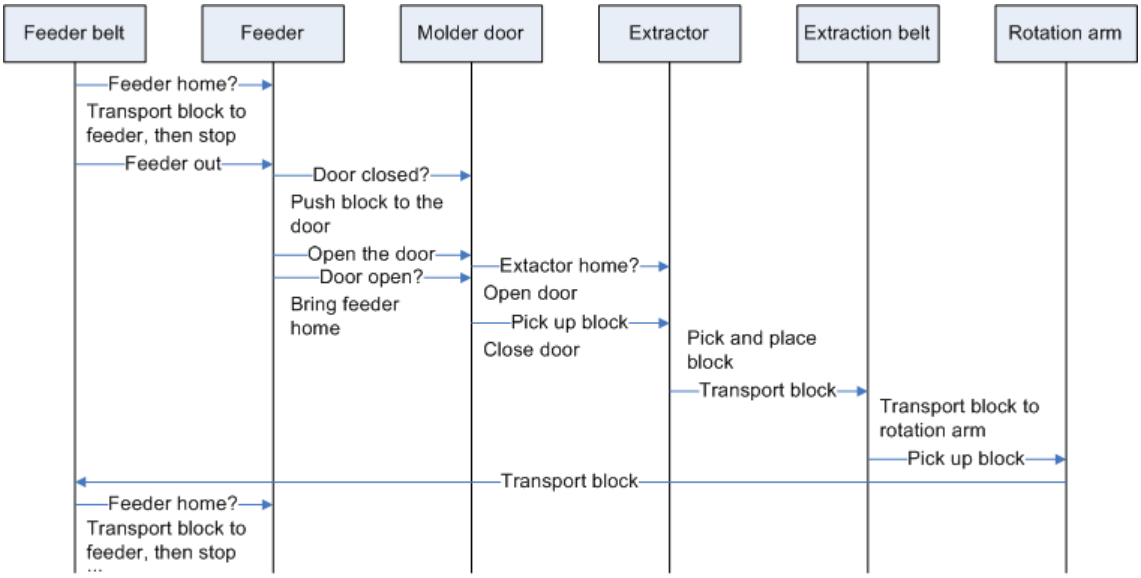


FIGURE 4.5 - Happy flow sequence diagram of the production cell controller (Huang et al., 2007)

The sequence diagram, shown in figure 4.5, runs in an infinite loop. Once a block is introduced, the setup does not stop anymore. This is due to the nature of the demonstrator; there is no extraction point. To solve this problem, each of the two belts is equipped with a watchdog timer. When a belt is transporting a block, and the watchdog expires time before the block reaches the end, the belt is stopped. This way it is possible to let the setup return to its idle logical state when all the blocks are removed from the system.

4.6.2 Error flow

In case something goes wrong, another course of action is required. The error flow is different for every PCU. As an example, the most destructive situation (for this setup) is given: two blocks caught in the *feeder* (figure 4.6). In this case two blocks are in between the *molder-door* and the *feeder*. Currently, when the *feeder* is directed to move forward, it pushes the two blocks against the *molder-door*. The *feeder* never reaches the intended position, issued by the motion profile, because a block obstructs the intended path. This causes the *feeder* to push even harder to reach the setpoint, causing damage to the *molder-door*. The controller can detect if the PCU is pushing too hard (high PWM output but no movement) and the *exception catcher* issues a sanity check failure when it occurs. In turn, the *exception handler* will generate the appropriate logical states for the *feeder* and its closest neighbours.

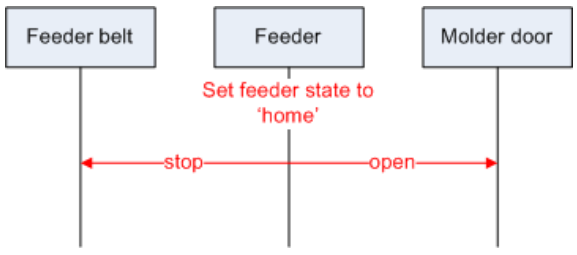


FIGURE 4.6 - Example error flow

4.7 gCSP as a code generation tool

The previously described models (figures 4.1–4.6) provide a solid framework for the production cell controllers.

However, manual implementation is error prone. By using code generation, a better, faster and more adaptable solution is possible. Keeping in mind that Handel-C is CSP based, and the requirement is set to make a CSP based model, the gCSP tool (Jovanovic et al., 2004) is a good fit. It

combines CSP based schematic diagram drawing, with support for channels, parallel processes and so on. gCSP also has code generation capabilities, however Handel-C code generation is not yet available. To this end, gCSP has been extended with Handel-C code generation, in order to generate code of the models mentioned in this chapter. Currently, a subset of the gCSP language is available for Handel-C code generation. This includes structure, code blocks and channel interconnects. The limitation prevents the use of gCSP readers and writers. These elements are declared in Handel-C code instead. As a result, no CSPm code can be generated from the models, not allowing a formal deadlock check one this models. For information on deadlocks in this system refer to Orlic (2007). The corresponding gCSP models are printed in appendix C.

4.8 Conclusion

In chapter 3, the loop controller was designed. However, loop controllers on their own are just functional blocks, not functional systems. In order to make the loop controllers work in a setup, a versatile and robust framework had to be created. This chapter stated several demands and requirements from which a structural and communicational framework was designed. The resulting framework provides a CSP-based, fairly generic approach to connect loop controllers to mechatronic systems. In order to port this framework to other applications, a few adjustments to the top-level communication channels may be necessary, i.e. bi-directional happy flow. When distributed control is implemented, a supervisory control system may be added to switch between controllers.

The implementation on the production cell is working correctly. The PCUs are synchronizing as planned, and a rudimentary safety is present.

The gCSP tool proved to be useful for implementing the framework. The Handel-C code generation saved time and prevented human error in the framework. However, some improvements need to be made to the code generation, in order to enable deadlock checks. Also, a gCSP library containing the designed framework among others would be useful to speed up the design of future projects.

5 Conclusions & Recommendations

5.1 Conclusions

This assignment proved that it is possible to control the production cell using only a general purpose FPGA and Handel-C as the hardware description language. Better performance then the existing CPU based solutions was obtained in terms of robustness. The setup now operates well, unaffected by the number of blocks in the system. However, the selected method of implementation, using just the FPGA, may not be sufficient in more demanding control applications.

Integer based control

The requirement of using only the FPGA implied the usage of the integer data type for calculations. This resulted in a lot of manual labour in terms of the PD controller algorithm implementation and the motion profile position signal approximation. Basically every controller algorithm had to be inspected for correct behaviour. In the future this process needs to be simplified. The results of the integer based controller proved to be good enough for this application.

Structural and communicational framework

The framework designed to implement the loop-controllers on a setup proved to be useful. It is applicable in any mechatronic setup and provides for all functionality required of a control system (section 4.3). The framework provides an easy way to communicate between units. In this assignment, the communication proved to be working correctly. No synchronization errors were detected on the working setup. A formal deadlock check was not possible however, due to limitations in the Handel-C code generation in gCSP.

5.2 Recommendations

Loop-controller implementation

The loop-controllers were implemented using the integer data type. It proved to work good enough for the control of the production cell. In retrospect, some points of improvement are found:

- *Integer based control*: The implementation of the loop-controllers may not be optimal. The floating point algorithm was adjusted to the integer data type, but better methods may be available. Some research on the subject of integer based control could result in a better algorithm.
- *Alternative implementations*: Other controller implementations where introduced in chapter three, but not pursued due to the stated requirements. These implementations could be further investigated, i.e. using RAM and/or an FPU.

Safety

The controller framework of chapter four provided for a safety layer. The actual implementation of safety was outside the scope of this assignment. Future work lies in the study and implementation of a proper safety layer.

Design trajectory

In this assignment 20-sim and gCSP were used to design the controller. Both tools had their shortcomings in this project.

20-sim

The integer data type support in 20-sim is low. This is due to the internal floating point mathematical engine that is used. Currently, calculations are only truncated at the end of a formula

whereas truncation of every step in the formula is needed. This is a point of improvement in 20-sim. Simulink does provide support for integer calculations by means of the fixed point toolbox. The use of this tool could simplify the design of the loop-controller. This would eliminate the need of an external DLL interfaced with 20-sim. 20-sim and MATLAB could also be coupled by using co-simulation (Damstra, 2008).

Both methods are viable workarounds until 20-sim has decent integer support.

gCSP

The gCSP tool still needs a lot of work for it to be an effective tool. Although it is a useful tool and a good concept, several points need to be addressed:

- *Improve stability*: Designing in gCSP is unstable at the moment. Some actions can render a model corrupt which would mean that a user would have to rebuild the model from scratch. A redesign of the model data structure could solve this.
- *Improve usability*: Currently basic copy-paste actions are not possible. This causes some delay in design, since every construct needs to be drawn by hand. Also, the tool does not support a library-like structure, such as 20-sim. The structural and communicational framework could be put in this library, for instance, for fast design and implementation of mechatronic systems. Finally, the model storage is large. A fairly straightforward model rapidly increases in size. If the tool is to be used for more complex models in the future, this needs to be addressed in order to speed up the time it takes to save a model. This can be solved by either or both:
  - Optimising data storage;
  - Storing only changes to the model, instead of storing the complete model.
- *Improve Handel-C code generation*: One of the biggest advantages of gCSP is the ability to generate code of a model. Currently, the Handel-C code generation template is limited, disabling the model to be fully implemented. It also excludes the generation of CSPm of a Handel-C targeted model, making formal deadlock checking hard, if not impossible. By adding Handel-C code generation the tool will be able to target the FPGA platform as well as the CPU platform.
- *Implement composite linkdrivers*: Currently, linkdrivers can only serve as an input or as an output to the outside world. Grouping linkdrivers to interface an (multiple) input-output device is currently not possible. Composite linkdrivers would expand the variety of target devices.

Since the implementation of gCSP leaves room for improvement, a possible solution would be a redesign of the tool. Pay attention, when redesigning, to follow the proper software development techniques. This way documentation from the very core of the program is assured. Also, if considered during the design, added functionality could be integrated by means of plugins.

A Production cell commander

An additional product of this assignment has been the production cell commander (figure A.1).

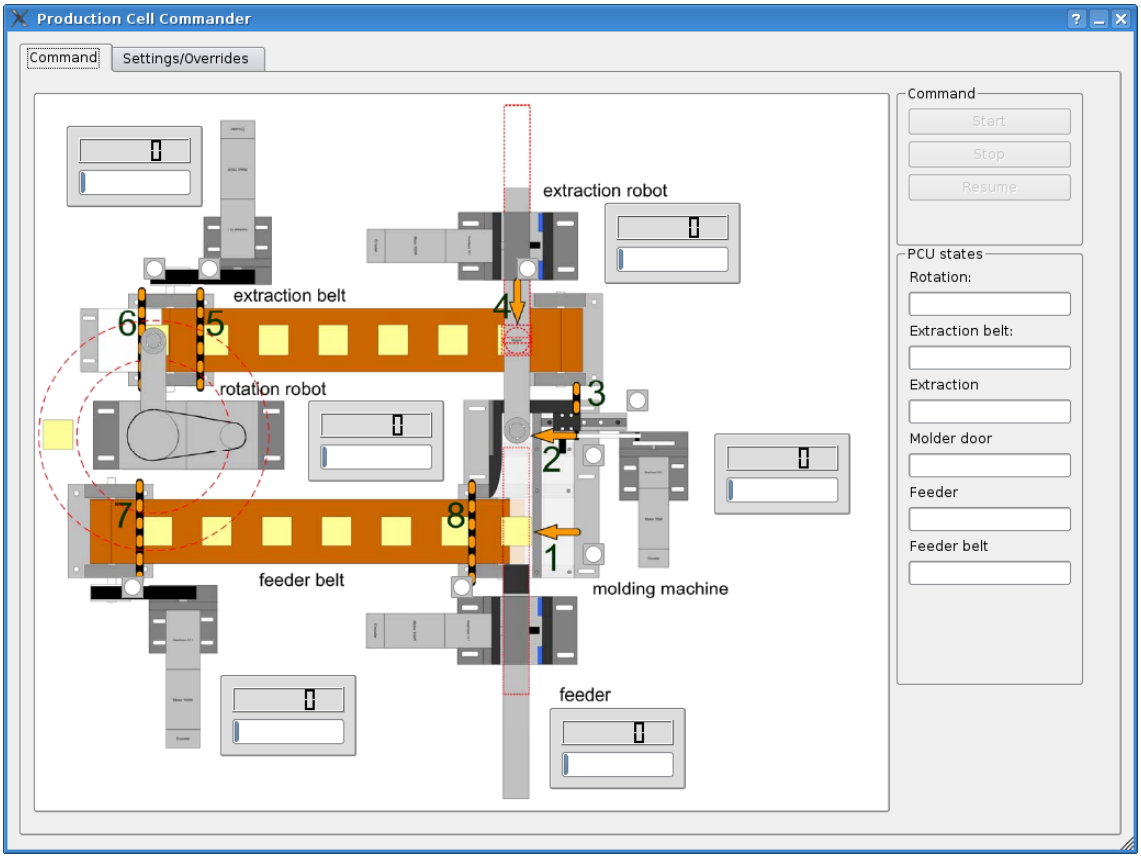


FIGURE A.1 - The production cell commander GUI

The production cell commander consists of two parts:

- The graphical user interface (GUI) of figure A.1;
- A data server, running on the target.

These two parts make up the commander, which is able to activate, disable and monitor the Handel-C controlled production cell. Both parts communicate over TCP/IP (figure A.2).

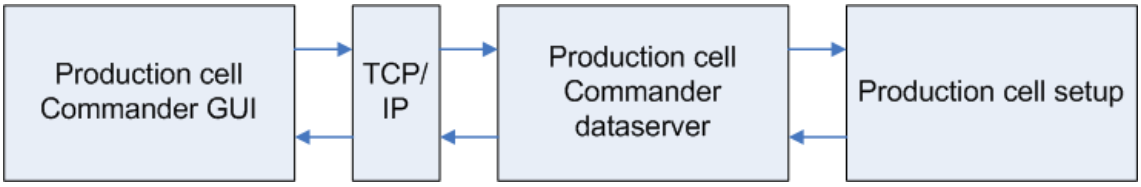


FIGURE A.2 - Communication structure

A.1 QT production cell commander GUI

The GUI has been made using trolltechs QT (Trolltech, 2008) and C++. QT is a framework for quickly making graphical user interfaces. QT is also platform independent, so, although it currently runs on Linux, it can easily be ported to Windows or MacOS.

A.1.1 Main screen

When started, the opening screen is presented. It consists of a static schematic drawing of the production cell with some dynamic dials, an interface to start or stop the setup and an overview of the state of each PCU (figure A.3).

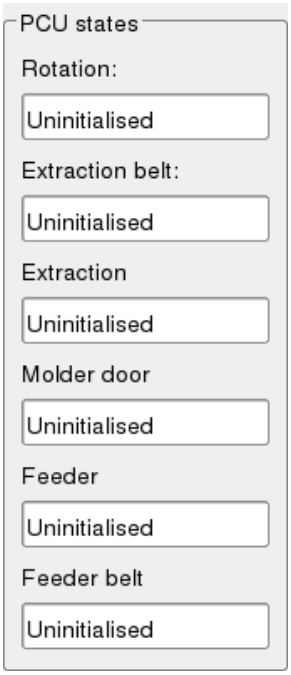
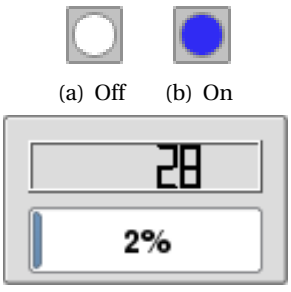


FIGURE A.3 - States overview widget

The indicators (figure A.4) on the schematic diagram include sensors (figure A.4(a) & A.4(b)) and the current counter value and encoder position for that motor (figure A.4(c)).



(a) Off (b) On

(c) PWM & encoder indicator

FIGURE A.4 - Indicators on the production cell GUI

A.1.2 Settings and override screen

The second screen (figure A.5) can be accessed by selecting the second tab called “Settings/Overrides”. This screen shows various selectors and buttons for manually controlling the production cell (currently not implemented), a server connection widget and an update speed control slider.

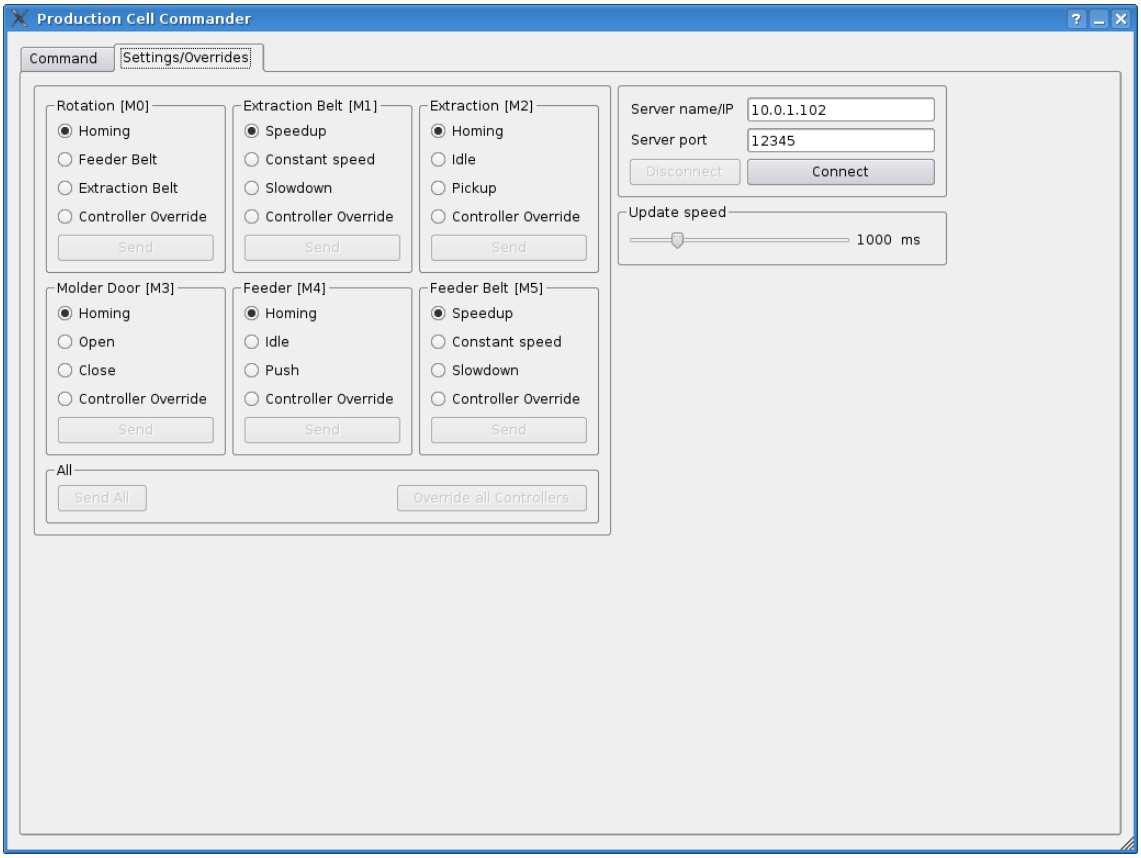


FIGURE A.5 - The settings and overrides screen

A.2 Production cell server

The second part of the server runs on the computer where the anything IO card is installed. The server fetches the log data of the anything IO card from hard-coded PCI addresses. In order to keep the server lightweight and portable, it is in a command line tool format.

```
PC104 server for QT.ProdCell.Commander

Usage:  ./server port [bitfile]
          |      |      |
          |      |      - optional bitfile to use when programming
          |      - The port number on which the server runs
          |        (must match the one in the GUI, default 12345)
          -exec name
```

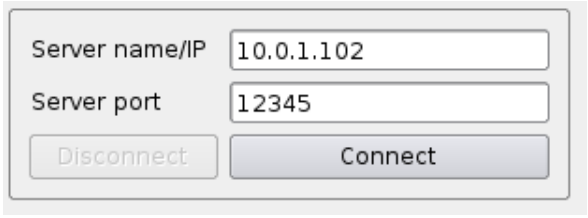
When the server is started, it expects one mandatory argument, containing a server port and a second optional argument, containing a path to a deviating bitfile. Normally the server uses the standard supplied bitfile, but a different bitfile can be supplied using this optional argument.

A.3 Connecting to the setup

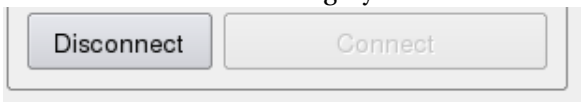
This step-by-step guide explains how to start the commander.

1 Start the server on the default port using the following command: ./server 12345
2 Start the GUI on the/another computer by double clicking the executable
3 Select the “Settings/Overrides” tab in the main screen

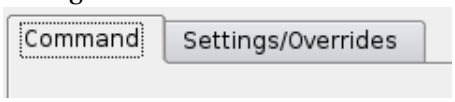
4 Connect to the server by typing the IP address of the server in the IP box and clicking on the connect button



5 Check the connect button grays out and the disconnect button is active



6 Change back to the command tab



7 Check there are no blocks in the system

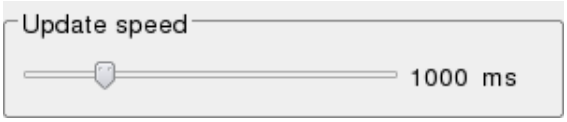
8 The start button will now be active, click it. The system will now start with a homing sequence, after this it will wait on a block to be introduced to the system. When done so, the system will operate and the production cell commander will monitor states and sensor, encoder and PWM values

A.4 Increasing the monitor speed

By default the update speed is set at one second. This is fairly slow for the setup, however, it is a good trade off between logging speed and computer load. To increase the monitoring speed follow these steps.

1 Select the “Settings/Overrides” tab

2 Locate the update speed slider, defaulted at one second

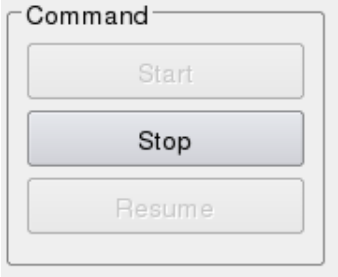


3 Move the slider to the left to maximal update speed of 1ms

A.5 Disconnecting from the setup

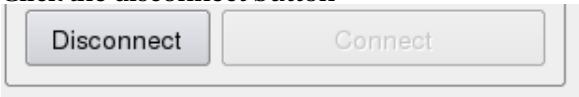
1 Remove all the blocks from the system

2 Stop the setup by clicking on the stop button



3 Select the “Settings/Overrides” tab

4 Click the disconnect button



5 Close the production cell GUI

6 Quit the server by using control+c

B PWM generator design

Due to the fact that Handel-C is a new language in the embedded control group, every component of the controllers had to be build from scratch. In this appendix the design of the PWM generators is treated.

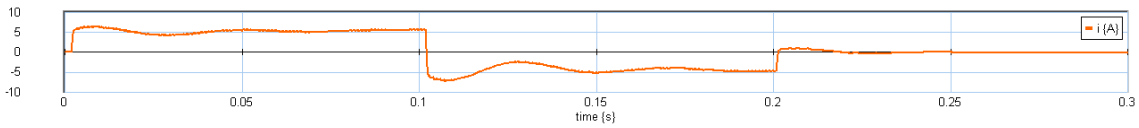
B.1 Current implementations

Since there are no PWM generators available in Handel-C, these needed to be programmed as well. Currently a MESA implementation is used, based on a 12 bit PWM generator (11 bits resolution and one direction bit). The frequency used in this generator was 16 kHz. This operates well, but it is unclear where the considerations for these particular values come from.

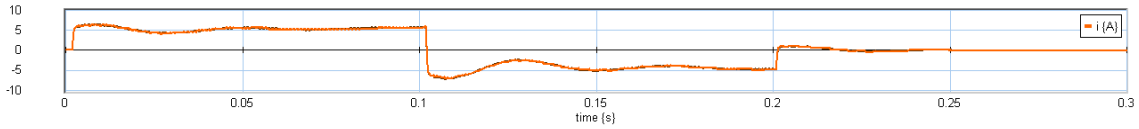
B.2 Design of the PWM generator

According to Valentine (1998) PWM frequencies must be chosen as low as possible. However, when chosen too low, an annoying beep can be heard. Since the production cell setup is a demonstrator, it must not produce too much ambient noise. A compromise must therefore be made between an efficient PWM generator and a quiet PWM generator. Also, the resolution must be high enough for the system to work precise enough, as to not induce oscillations.

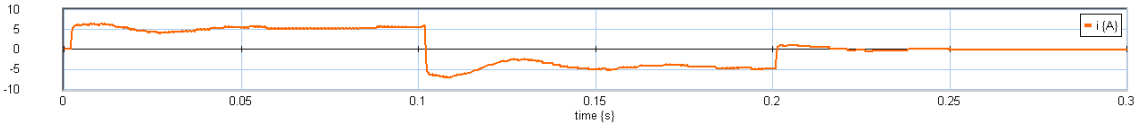
Since there is already a competent model (section 3.1.1) this can easily be tested. When altering the resolution, the motor current shows variations in the signal. A higher and lower resolution, then currently used, is tested and if a lower resolution does not result in a change of the signal, the lower resolution is selected. The results of 13 bits to 11 bits resolution is shown in figure B.1.



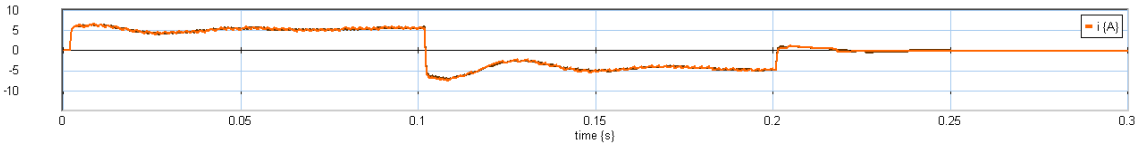
(a) 13 bits resolution



(b) 12 bits resolution plotted over 13 bits resolution



(c) 12 bits resolution



(d) 11 bits resolution plotted over 12 bits resolution

FIGURE B.1 - Motor current using different resolutions

This shows that a resolution of 11 bits is too low. Noise is clearly present when compared to the 12 bits signal (figure B.1(d)). The 12 bits signal is not too polluted when plotted on its own (figure B.1(c)) and compared to the 13 bits signal (figure B.1(b)) the differences are small. Therefore, the PWM resolution may be 12 bits or higher. However, the resolution is not only limited to the results in simulation. The maximum numerical value of the PWM resolution also determines the PWM frequency. The PWM generator uses a counter as a reference timer to its signal. This counter has a word width of the PWM resolution minus one bit, which is used to indicate

the direction.

The counter starts at zero and is raised by one every clock pulse. At certain time an overflow occurs, which effectively resets the counter to zero. This is the reference of the PWM period of frequency. A simple formula can be used to determine the frequency of the PWM generator. The denominator of this formula is the range of the counter. The PWM formula is printed below.

$$\begin{aligned} f_c &: systemclock \\ PWM_{res} &: PWMresolutioninbits \\ PWM_f &: PWMfrequency \\ f_{PWM} &= \frac{f_c}{2^{PWM_{res}-1}-1} \end{aligned}$$

The production cell hardware platform has two available clocks: 50 MHz and 33 MHz. In table (table B.1) the resulting PWM frequencies options are shown.

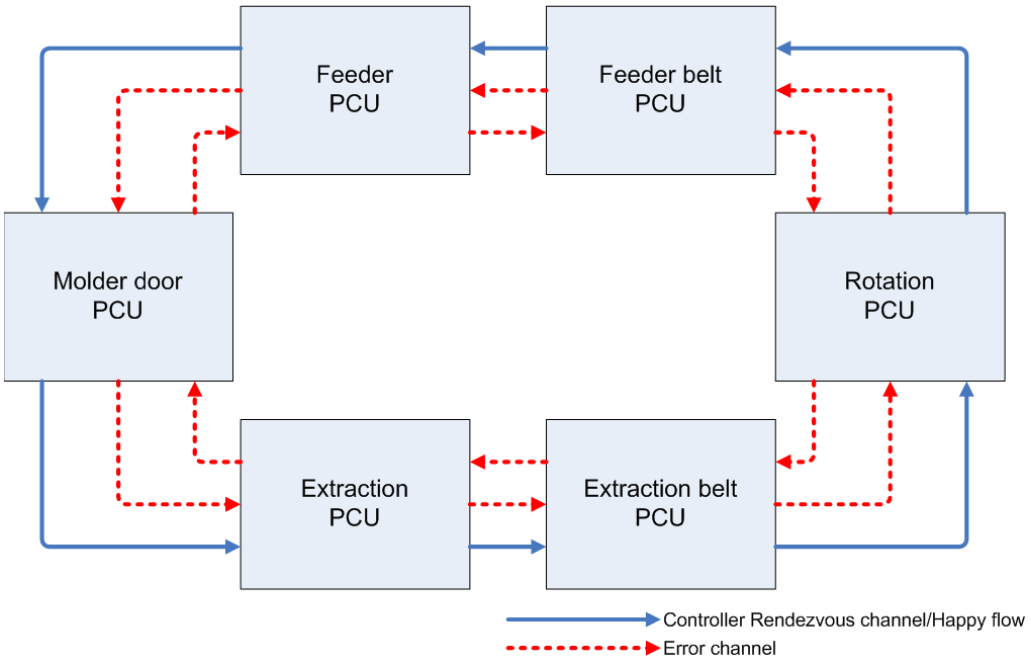
	50MHz	33MHz
13 bits (counter range: 0-4095)	12.2 kHz	8.1 kHz
12 bits (counter range: 0-2047)	24.4 kHz	16.1 kHz

TABLE B.1 - PWM frequency options

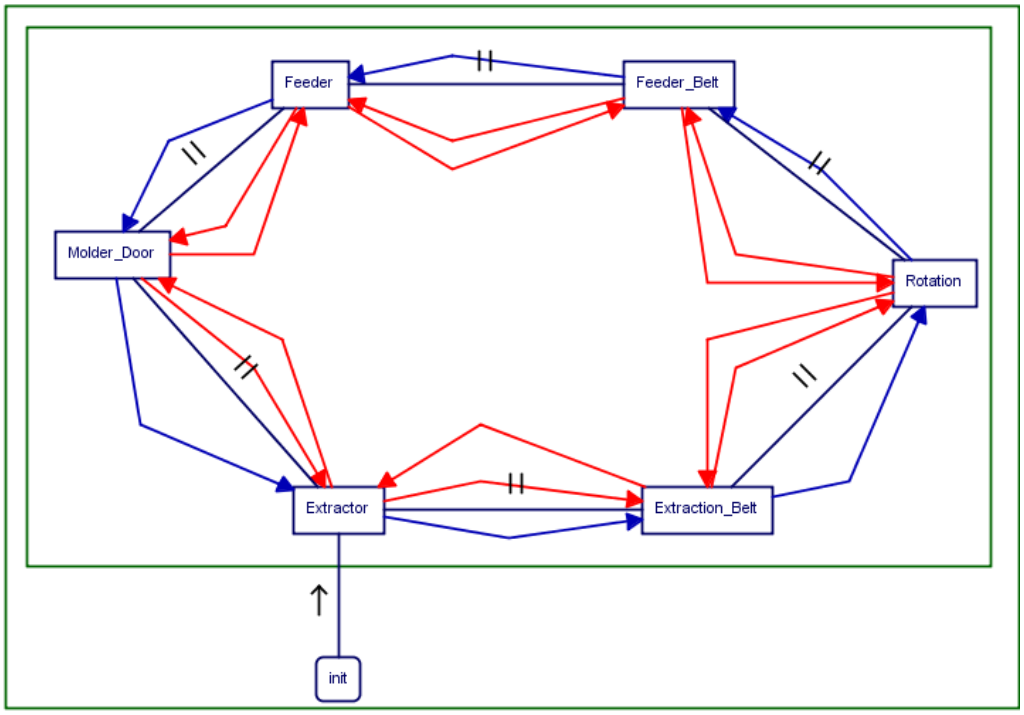
The PWM should be as low as possible to obtain the lowest power loss. Conversely, a too low frequency will result in an annoying beep. The noise can only be detected by testing on the setup. After testing, it showed that the 8.1 kHz and 12.2 kHz options produced a very annoying beep and so these are discarded. This leaves the 16.1 kHz and 24.4 kHz options and sets the PWM resolution at 12 bits. The human hearing spectrum ends at around 20 kHz everything above this threshold is considered inaudible, implying the 24 kHz option should be selected. However, the 16 kHz PWM signal also does not produce an audible tone on this setup. A lower PWM frequency yields a higher efficiency and so the 16 kHZ is selected. Therefore the resulting PWM resolution is 12 bits and the system clock is set at 33 MHz.

C gCSP models

Chapter 4 mentioned the use of gCSP as a code generation tool. This chapter shows the gCSP models of the rotation robot next to the diagrams used earlier in this report.

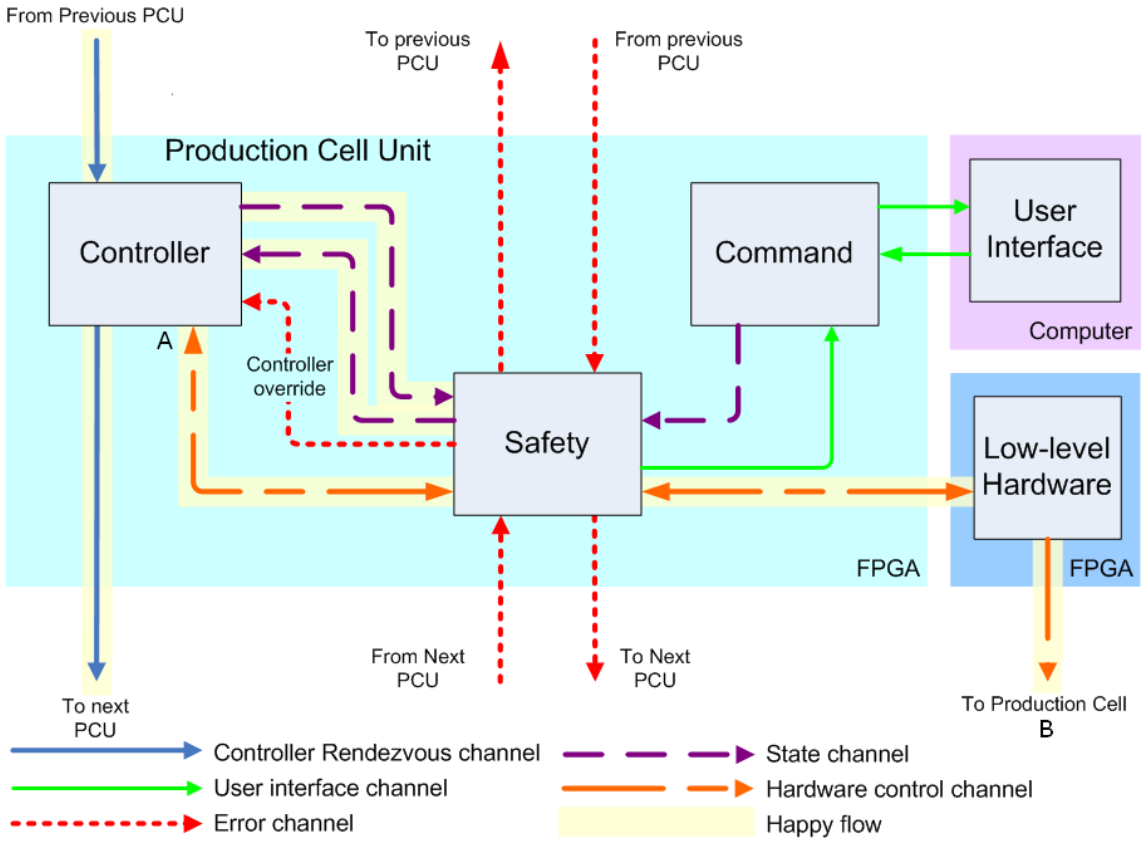


(a) Visio diagram

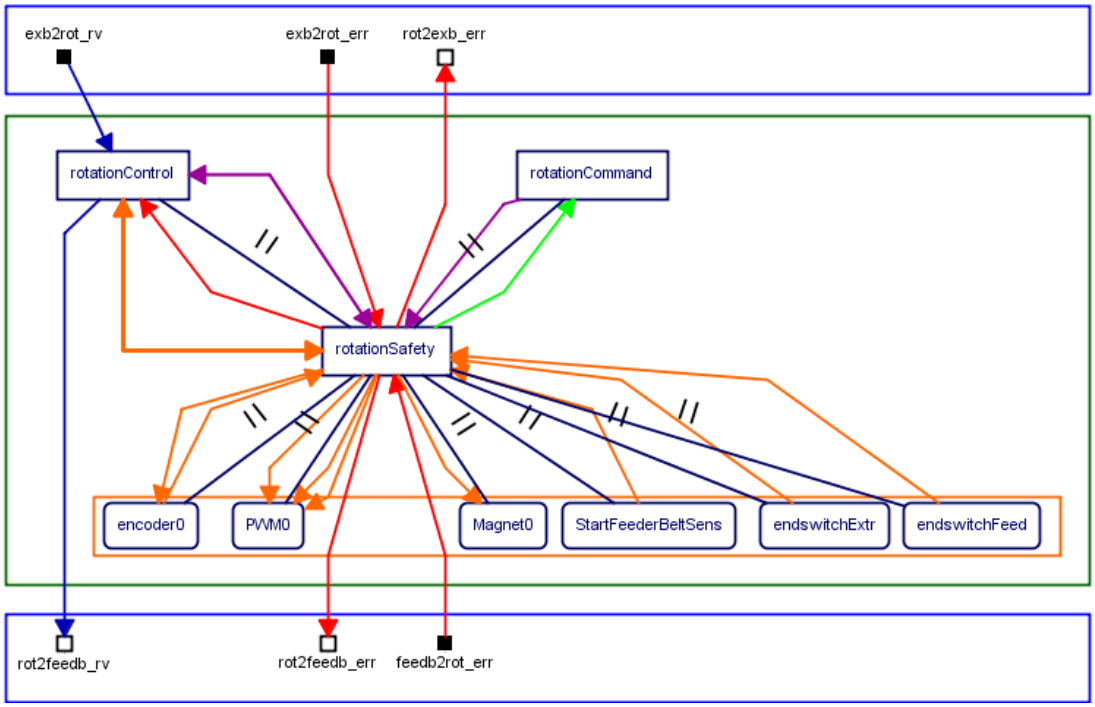


(b) gCSP model

FIGURE C.1 - Top level diagram

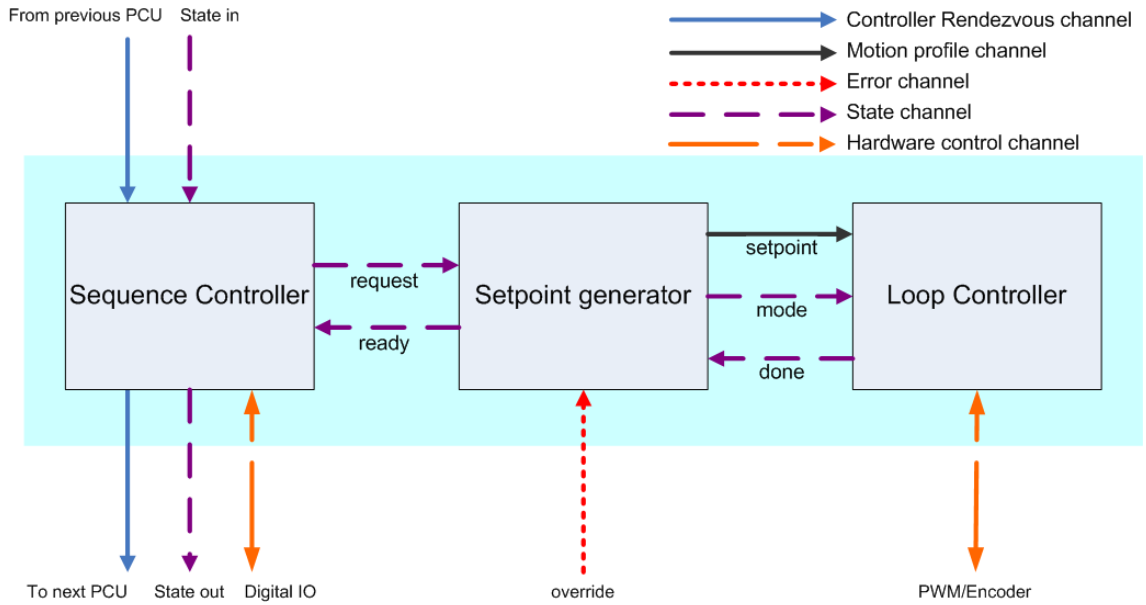


(a) Visio diagram

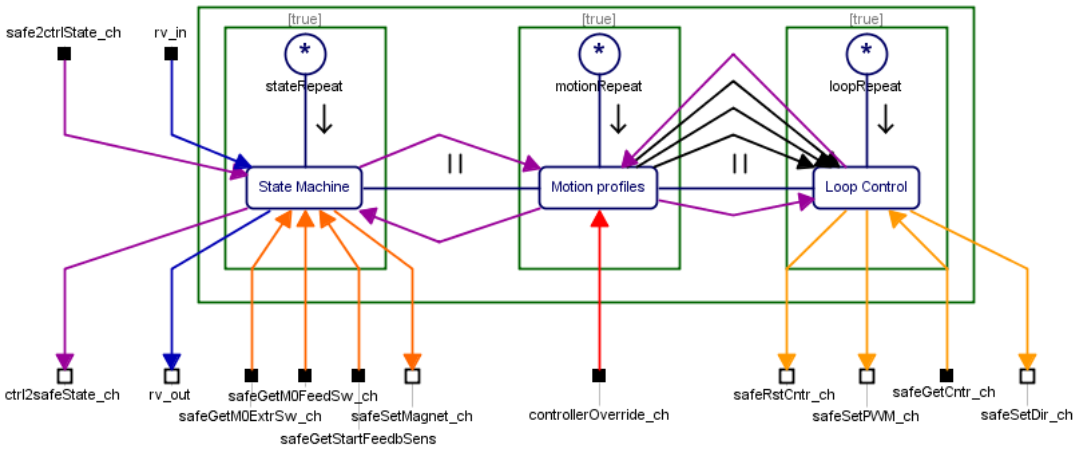


(b) gCSP model

FIGURE C.2 - PCU implementation

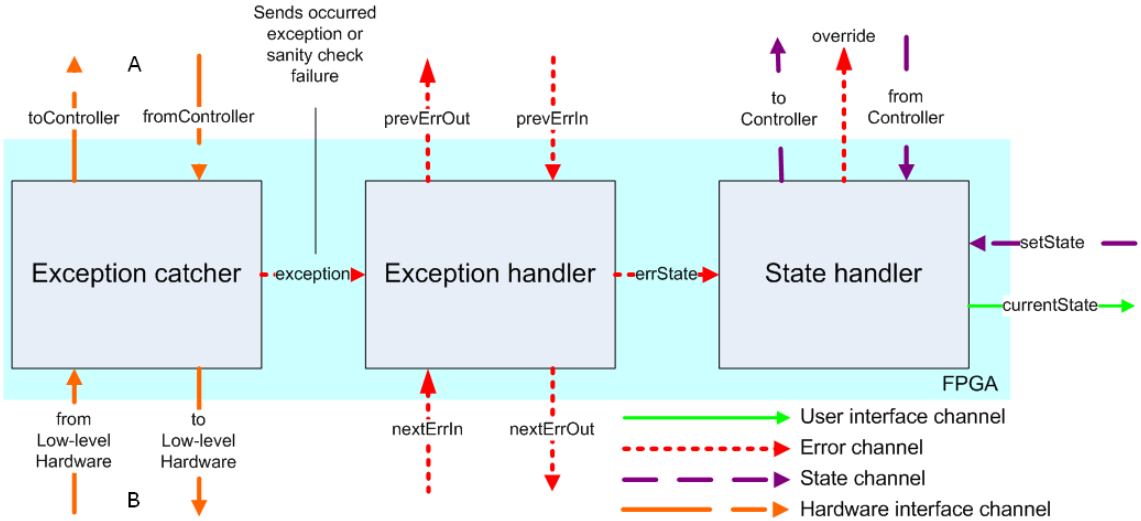


(a) Visio diagram

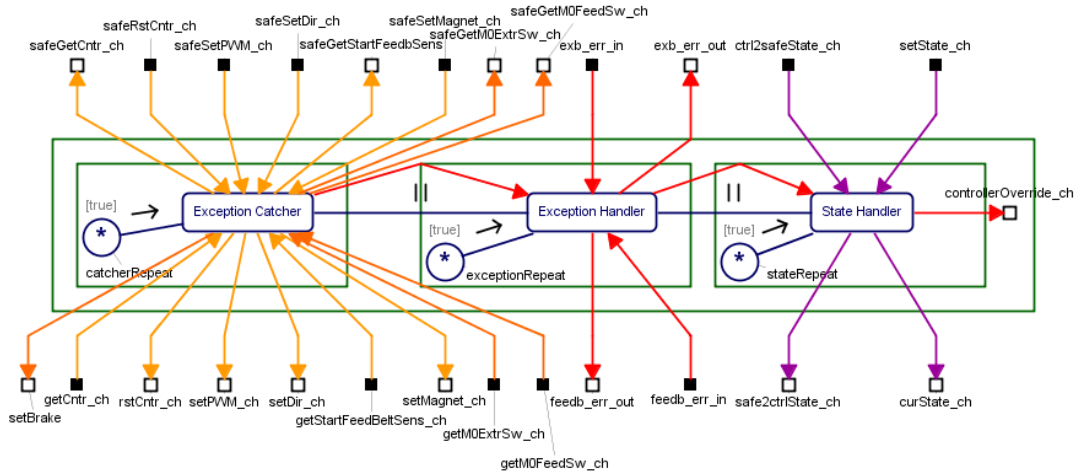


(b) gCSP model

FIGURE C.3 - Controller block



(a) Visio diagram

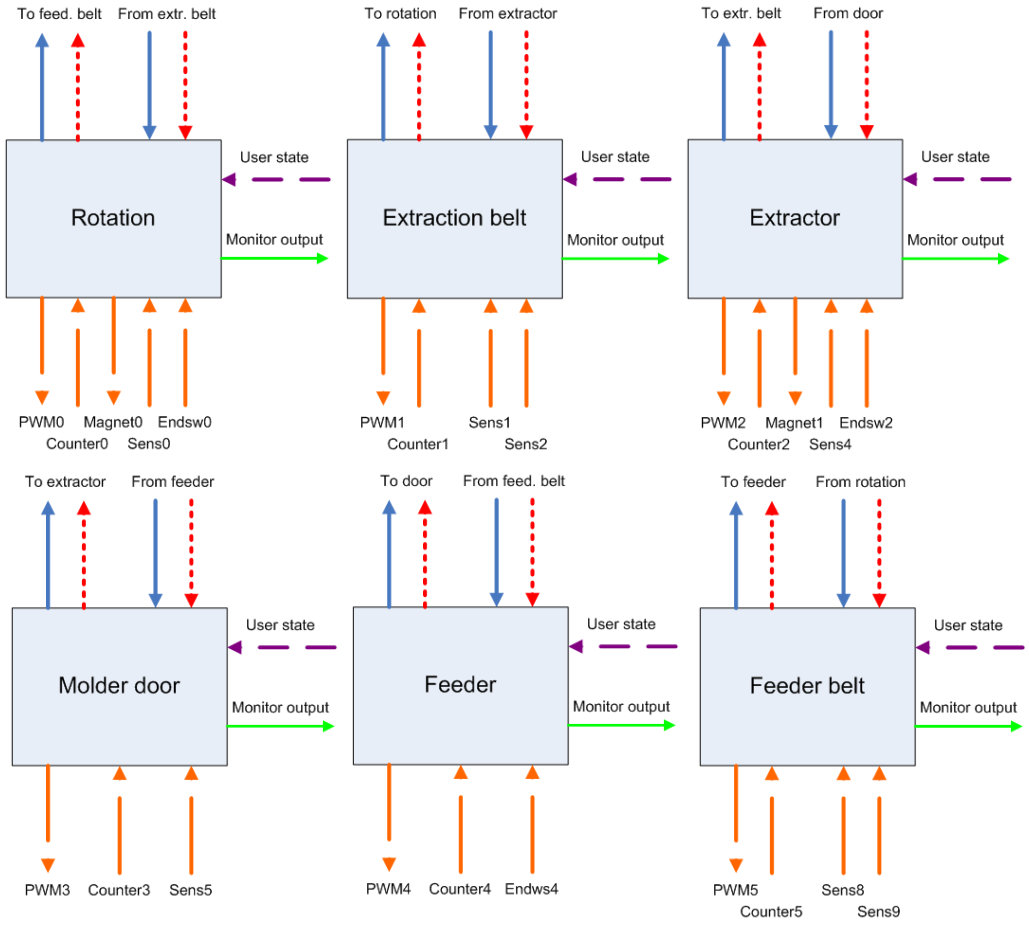


(b) gCSP model

FIGURE C.4 - Safety layer

D PCU modules

This appendix gives an overview of all the PCUs and shows their related inputs, outputs and communication channels.



(d) Molder door PCU (e) Feeder PCU (f) Feeder belt PCU

FIGURE D.1 - CPU IOs

The connections of the interfaces to the FPGA board are described on the following pages.

Description	Connector pin	AnyIO
M0_fault	1	0
M0_endsw_A	3	1
M0_endsw_B	5	2
M0_enc_A	7	3
M0_enc_B	9	4
M1_fault	11	5
UNUSED	13	6
UNUSED	15	7
M1_enc_A	17	8
M1_enc_B	19	9
rot_mag_sens	21	10
startFeederBeltSens	23	11
rotationPlatformSens	25	12
endExtractionBeltSens	27	13
UNUSED	29	14
M0_dir	31	15
M0_PWM	33	16
M0_Brake	35	17
M1_dir	37	18
M1_PWM	39	19
M1_Brake	41	20
rot_mag_out	43	21
UNUSED	45	22
UNUSED	47	23

TABLE D.1 - Connector 1

Description	**Connector pin**	**AnyIO**
M2_fault	1	24
M2_endsw_A	3	25
M2_endsw_B	5	26
M2_enc_A	7	27
M2_enc_B	9	28
M3_fault	11	29
M3_endsw_A	13	30
M3_endsw_B	15	31
M3_enc_A	17	32
M3_enc_B	19	33
extr_mag_sens	21	34
extractionSens	23	35
doorSens	25	36
UNUSED	27	37
UNUSED	29	38
M2_dir	31	39
M2_PWM	33	40
M2_Brake	35	41
M3_dir	37	42
M3_PWM	39	43
M3_Brake	41	44
extr_mag_out	43	45
UNUSED	45	46
UNUSED	47	47

TABLE D.2 - Connector 2

Description	Connector pin	AnyIO
M4_fault	1	48
M4_endsw_A	3	49
M4_endsw_B	5	50
M4_enc_A	7	51
M4_enc_B	9	52
M5_fault	11	53
UNUSED	13	54
UNUSED	15	55
M5_enc_A	17	56
M5_enc_B	19	57
UNUSED	21	58
feederSens	23	59
endFeederBeltSens	25	60
UNUSED	27	61
UNUSED	29	62
M4_dir	31	63
M4_PWM	33	64
M4_Brake	35	65
M5_dir	37	66
M5_PWM	39	67
M5_Brake	41	68
UNUSED	43	69
UNUSED	45	70
UNUSED	47	71

TABLE D.3 - Connector 3

E PCI mapping of the Production Cell with Handel-C

This chapter describes the relative addresses of the FPGA on the PCI bus.

E.1 Relative Read addresses

Address	Function	Width
0x00	0xBABECAFE	Long (32 bits)
0x04	Digital IO	Long (32 bits)
0x08	Counter 0	Long (32 bits)
0x0c	Counter 1	Long (32 bits)
0x10	Counter 2	Long (32 bits)
0x14	Counter 3	Long (32 bits)
0x18	Counter 4	Long (32 bits)
0x1c	Counter 5	Long (32 bits)
0x20	PWM 0	Long (32 bits)
0x24	PWM 1	Long (32 bits)
0x28	PWM 2	Long (32 bits)
0x2c	PWM 3	Long (32 bits)
0x30	PWM 4	Long (32 bits)
0x34	PWM 5	Long (32 bits)
0x38	Setpoint 0	Long (32 bits)
0x3c	Setpoint 1	Long (32 bits)
0x40	Setpoint 2	Long (32 bits)
0x44	Setpoint 3	Long (32 bits)
0x48	Setpoint 4	Long (32 bits)
0x4c	Setpoint 5	Long (32 bits)
0x50	State 0	Long (32 bits)
0x54	State 1	Long (32 bits)
0x58	State 2	Long (32 bits)
0x5c	State 3	Long (32 bits)
0x60	State 4	Long (32 bits)
0x64	State 5	Long (32 bits)

E.2 Relative Write addresses (not implemented)

Address	Function	Width
0x00	State 0	Byte (8 bits)
0x01	State 1	Byte (8 bits)
0x02	State 2	Byte (8 bits)
0x03	State 3	Byte (8 bits)
0x04	State 4	Byte (8 bits)
0x05	State 5	Byte (8 bits)

F FPGA information

The selected FPGA for this assignment is the Xilinx Spartan 3 with 1.500k gates. It is a general purpose FPGA, so no hard-core functionality is available.

F.1 CLB usage

The Spartan 3 offers a way of viewing the usage of the FPGA. An exert of these data is shown in listing F.1.

```
---------- MAP ----------
  Release 9.2i - Map J.36
  Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.
  Using target part "3s1500fg320-4".
  Mapping design into LUTs...
  Writing file ctrl_lib.ngm...
  Running directed packing...
  Running delay-based LUT packing...
  Running related packing...
  Writing design file "ctrl_lib.ncd"...

  Design Summary:
  Number of errors:      0
  Number of warnings:    2
  Logic Utilization:
  Number of Slice Flip Flops:        3,263 out of  26,624   12%
  Number of 4 input LUTs:            7,997 out of  26,624   30%
  Logic Distribution:
  Number of occupied Slices:                      5,790 out of 13,312  43%
  Number of Slices containing only related logic:5,790 out of  5,790 100%
  Number of Slices containing unrelated logic:       0 out of  5,790   0%
  *See NOTES below for an explanation of the effects of unrelated logic
  Total Number of 4 input LUTs:         10,055 out of  26,624   37%
  Number used as logic:                7,997
  Number used as a route-thru:         1,650
  Number used for Dual Port RAMs:        364
  (Two LUTs used per Dual Port RAM)
  Number used as Shift registers:         44
  Number of bonded IOBs:                  95 out of     221   42%
  IOB Flip Flops:                      22
  Number of GCLKs:                         1 out of       8   12%

  Total equivalent gate count for design:  122,503
  Additional JTAG gate count for IOBs:  4,560

---------- PAR ----------

  Device speed data version:  "PRODUCTION 1.39 2007-04-13".


  Device Utilization Summary:

  Number of BUFGMUXs                        1 out of 8      12%
  Number of External IOBs                  95 out of 221    42%
```

```
Number of LOCed IOBs                    95 out of 95       100%

Number of Slices                      5790 out of 13312    43%
Number of SLICEMs                     226 out of 6656       3%



Overall effort level (-ol):   High
Placer effort level (-pl):    High
Placer cost table entry (-t): 1
Router effort level (-rl):    High
```

LISTING F.1 - Exert of Xilinx output

Listing F.1 shows that only 43% of the available slices is occupied. More then half of the FPGA is still available.

F.2 CLB location

Most of the FPGA's CLB's are used by the PD algorithm. Table F.1 shows the usage of the PD controller.

LUTs	FFs	
290	1	uD1 = ((factor*extend(uD_prev,23))/tauDbeta);
184	1	uD2 = ((factor*(er-er_prev))/tauDkP);
81	1	uD3 = (er/(kpSampletime/factor));
32	1	uD = (uD1<-*16)* + *uD2* + *uD3;*

TABLE F.1 - FPGA occupation of the PD algorithm

Keep in mind that this algorithm is applied 6 times, each for every PCU.

Bibliography

Åström, K. J. and T. Hagglund (1995), *PID Controllers: Theory, Design and Tuning*, ISA, second edition, ISBN 978-1556175169.

Bennet, S. (1988), *Real-Time computer control: An introduction*, Prentice-Hall, New York, NY.

Bennet, S. and D. Linkins (1984), *Real-Time computer control*, Peregrinus, London, UK.

van den Berg, L. (2006), Design of a Production Cell Setup, MSc Thesis 016CE2006, University of Twente.

Broenink, J. and G. Hilderink (2001), A structured approach to embedded control systems implementation, in *2001 IEEE International Conference on Control Applications*, M. W.Spong, D. Repperger and J. M. I. Zannatha (Eds.), México City, México, pp. 761–766, ISBN 0-7803-6735-9.

Bruyninckx, H. (2000), Project Orocos, Technical report, Katholieke Universiteit Leuven.

Catalytic (2008), URL `http://www.catalytic.com/`.

Celoxica (2005), Handel-C Language Reference Manual.

Celoxica (2008), URL `http://www.celoxica.com/`.

Cooling, J. E. (2003), Software Engineering for Realtie Systems, Addison Wesley, pp. 686–699, ISBN 0-201-59620-2.

Damstra, A. (2008), Virtual prototyping through co-simulation in hardware/software and mechatronics co-design, MSc Thesis 005CE2008, University of Twente.

Groothuis, M. (2004), Distributed HIL simulation for BodeRC, MSc Thesis 020CE2004, University of Twente.

Groothuis, M. (2008), ViewCorrect, URL `http://www.ce.utwente.nl/viewcorrect/`.

Hoare, C. (1985), *Communicating sequential Processes'*, Prentice-Hall, ISBN 0131532715.

Huang, J., J. Voeten, M. Groothuis, J. Broenink and H. Corporaal (2007), A model-driven approach for mechatronic systems, in *Seventh International Conference on Application of Concurrency to System Design*, Bratislava, Slovakia, pp. 127–136, ISBN 0-7695-2902-X.

Jovanovic, D. S., B. Orlic, G. Liet and J. Broenink (2004), gCSP: A Graphical Tool for Designing CSP systems, in *Communicating Process Architectures*, IOS press, Oxford, UK, pp. 233–251, ISBN 1586034588.

Karapetian, A. R. (2006), PID controller in FPGA, Pre-doctoral assignment 035CE2006, University of Twente.

Kernighan, B. and D. Ritchie (1988), *The C programming language*, Prentice Hall, second edition, ISBN 0131103628.

Kuppeveld, T. v. (2007), Model-based redesign of a self-balancing scooter, MSc Thesis 022CE2007, University of Twente.

Lee, P. A. and T. Anderson (1990), *Fault tolerance, principles and practice*, Springer-Verlag, New York, NY.

Maljaars, P. (2006), Control of the Production Cell Setup, MSc Thesis 039CE2006, University of Twente.

Mrochuk, J. and B. Carson (2008), Manticore - Open Source 3D Graphics Accelerator Project, URL http://www.icculus.org/manticore/.

Orlic, B. (2007), *SystemCSP: A graphical language for designing concurrent component-based embedded control systems*, Control Engineering, University of Twente, Enschede, ISBN 978-90-365-2573-2.

Page, I. (1998), Hardware Compilation Research Group, URL http://archive.comlab.ox.ac.uk/hwcomp/index.html.

Page, I. and W. Luk (1991), Compiling Occam into field-programmable gate arrays, in *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Eds. W. Moore and W. Luk, Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK, pp. 271–283.

Silberschatz, A., P. B. Galvin and G. Gagne (2005), Operting System Concepts, John Wiley & Sons, Inc., Hoboken, NJ, pp. 8–10, ISBN 0471694665.

Smit, G., L. Smit, P. Heysters, M. Rosien and T. Krol (2008), Chameleon: reconfigurable computing, URL http://chameleon.ctit.utwente.nl/Projects/?project=Chameleon.

Spivey, M. and I. Page (1993), How to program in Handel, Technical report, Oxford University Computing Laboratory.

Tanenbaum, A. S. and A. S. Woodhull (1987), Operating systems, design and implementation, Prentice Hall, pp. 82–92, second edition, ISBN 0136386776.

Theelen, B., O. Florescu, M. Geilen, J. Huang, P. van der Putten and J. Voeten (2007), Software/Hardware Engineering with the Parallel Object-Oriented Specification Language, in *5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, IEEE, pp. 139 – 148.

Trolltech (2008), Qt: Cross-Platform Rich Client Development Framework, URL http://trolltech.com/products/qt.

Valentine, R. (1998), Motor control electronic handbook, McGraw-Hill, pp. 24–30, ISBN 0-07-066810-8.

Valk, W. d. (1997), Leerboek ASIC's (Dutch), Addison Wesley, pp. 47–58, second edition, ISBN 9055743674.

Visser, P., M. Groothuis and J. Broenink (2007), Multi-purpose toolchain for embedded control system code on a variety of targets, Proceedings, University of Twente.

Vranesic, Z., S. Brown and R. Francis (1992), Field-Programmable Gate Arrays, Kluwer Academic, pp. 1–11, ISBN 0792392485.

Wijbrans (1993), *Twente Hierarchical Embedded Systems Implementation by Simulation (THESIS)*, Universiteit van Twente.

Xilinx (2008), URL http://www.xilinx.com.

van Zuijlen, J. (2007), Feasibility study on Handel C for Embedded Control, Pre-doctoral assignment 014CE2007, University of Twente.