NTNU
Norwegian University of
Science and Technology

Self Reconfiguration of Clock Networks on FPGA

Methodology for partial reconfiguration of synchronous modules at run-time

Sindre Hansen

Master of Science in Electronics
Submission date: June 2011
Supervisor: Kjetil Svarstad, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

With the project as basis, the objectives are:
- To further develop the methodology for clocked, dynamic, reconfigurable modules.
- To integrate the methodology for clocking with the HWOS-methodologies previously developed.

The methods should be tested by implementing a chosen system on an FPGA. The results should be compared with published results if such exist.

Assignment given: 17. January 2011
Supervisor: Kjetil Svarstad, IET

Summary

In this thesis, methodology for partial self-reconfiguration of synchronous modules has been developed. A simple software-based scheduler has been built for scheduling synchronous modules on the FPGA. The motivation behind this was that partial reconfiguration of synchronous modules at run-time had not been performed earlier in the AHEAD-project. Also, the project report written by the same author as this thesis has shown that a synchronous module can be replaced in a bitfile. However, the project report did not perform this reconfiguration at run-time.

Based on the project report, the problem has been decomposed and simple tests using clocked flip-flop designs have been performed on the FPGA. These tests forms a proof-of-concept for partial self-reconfiguration of synchronous modules on the Virtex-4 FPGA. However, the tests also showed that the reconfiguration time was quite high. It took several seconds to write one partial bitstream to the configuration memory.

Vegard Endresen has previously made a backend module for data transfer between the HWOS and a reconfigurable module. Experiments were performed in this thesis to see if the clocking methodology could be integrated into this backend module. The module could be built with the methodology, but a running solution on the FPGA was not shown.

The software part of the HWOS was rewritten from scratch as the previous version was not thoroughly analyzed. A round-robin scheduler using priority queues has been implemented. A test-driven development technique has been used for development, hopefully making the system more robust. The scheduler is a part of a daemon running on the embedded system, where a message server handles requests for new processes and a placer places new tasks on the FPGA. The complete system was initially based on ideas and code developed by Sverre Hamre and Vegard Endresen in previous AHEAD-projects.

Foreword

This is the final report on my master thesis in the programme option *Design of Digital Systems* under the study programme *Electronics* at the Norwegian University of Science and Technology (NTNU). The work has been done during 20 weeks in the spring semester of 2011.

For me personally the work has been challenging, but the learning outcome has been large. It is exciting to walk through all the phases of building an embedded system, from planning and researching to the final implementation. I definitely think this experience will help me as an engineer out in the industry.

The challenges when building a complex system like this are many. It is a lot of work to read up on the existing work in the field, especially since partial self-reconfiguration on FPGA is a rather new concept. The practical solutions may be few and there may not be a "de facto standard" way of doing things. There's not tons of learning books on the subject, but articles and research projects with suggested implementations. Another challenge is working with an embedded system with parts in both hardware and software. In such a complex system, there are many parts that will have to work together and the use of several tools have to be learnt to accomplish the final solution.

I see the learning outcome of this thesis as threefold. Firstly, I have gained a lot of practical experience with the tools and the technology. I have learned a lot on C-programming, automated testing and working with the Xilinx tools. The second is all the new knowledge I've gained on partial self-reconfiguration and how a hardware operative system can be built in practise. The last is how to carry out a quite large project. It is interesting to see how important a good working methodology is (methodology is presented in section 1.2) and how I should relate to work done by others. Likewise important is how I can document and structure my work so *others* can relate to *my work*.

I would like to thank professor Kjetil Svarstad for guidance and help during the semester. I also would like to thank Magnus Namork, who has been writing a master thesis on the Network On Chip-part of the AHEAD-project, for tool-specific discussions.

Trondheim, june 2011
Sindre Hansen

Acronyms and expressions

FPGA
: Field Programmable Gate Array. An array of programmable logic designed to be reconfigured several times by the designer or end-user.

BRAM
: Block Random Access Memory. Small memory blocks integrated on the FPGA.

LUT
: Lookup-table. In digital logic, a LUT is typically implemented using multiplexers.

CLB
: Configurable Logic Block. One unit of the programmable logic on the FPGA. Typically has LUTs, multiplexers, flip-flops, latches and more.

Slice
: The CLB on Xilinx Virtex FPGAs.

FF
: Flip-flop. One bit storage element (register) on the FPGA. There are typically one or more of these in each CLB.

Bitfile
: In this thesis, this is typically a file containing a complete or partial configuration for the programmable logic on the FPGA.

HWOS
: Hardware Operative System. In this thesis, HWOS is a operative system for controlling execution of tasks on an FPGA.

AHEAD
: Ambient Hardware, Embedded Architectures on Demand.

Contents

Chapter 1

Introduction

1.1 What has been done

For partial reconfiguration of synchronous modules, several experiments have been performed on the FPGA using the Suzaku development board. A proof of concept has been developed and it has been shown that synchronous modules can be placed on the FPGA by writing them to ICAP (Internal Configuration Access Port).

For integration with the previously developed HWOS-methodologies, several experiments has been done with the instruction- and data cache backend made in [End10]. The software part of the HWOS-daemon has also been rewritten from scratch.

For a real-world system, a software-based scheduler has been integrated in the HWOS-daemon. This scheduler is working, but it has not been performed scheduling using FPGA-based modules.

1.2 Methodology

The following are the methodology and working philosophy adopted in this thesis. Much of the work done in this thesis is experimental and based on an existing framework. Parts of the system has been designed for or somehow depends on the propertiary, Xilinx-based FPGA platform. This is in contrast to a completely constructive approach where you build your own system from scratch. Some of the listed points are extra precautions that should be taken because of this fact.

Isolating the problem should be done at an early stage. The reconfigurable framework consists of many parts that will have to work together and if a simple run time reconfigurable module should be tested, a complete system would have to be set up. As discussed in [Han10], this process takes quite

some time and the documentation of the FPGA-platform from Xilinx does not always contain the level of detail needed.

Making good assumptions is another aspect that should be given extra thought in this kind of thesis. The framework is built during several master thesises and the experimental approach is central in most of them. Because of this and because the field is rather new and unexplored, one should be especially critical to claims and hypothesis in earlier work and available articles on the subject. Vague formulations must of course be avoided. It should be made clear if some parts are uncertain or not thoroughfully tested.

A broad litterature search should be done. It is really helpful to see how other people have solved the same or a similiar problem. Their experiences and thoughts can be important guidelines when choosing an implementation or making design decisions. If other implementations are easily accessible, they can be integrated into the project and with that "reinventing the wheel" can be avoided.

The portability of the framework should be taken into consideration. Targeting a completely abstract FPGA is probably to limiting, but the technology and the components on the test-platform should be compared to available technology on other FPGAs. This is especially important for the work done on the given framework, as one of the goals is to be more independent of propertiary tools. The embedded software in this thesis has been written in ANSI-C. It can easily be compiled on a standard PC (x86) architecture and for the embedded PowerPC-processor that is present on the Suzaku development board. This can be valuable if the code must be ported to another microprocessor.

The code should be well structured. The code in this thesis has been written in the low-level languages C and VHDL. In contrast to higher level languages like Java or Python, these languages does not contain the same amount of abstraction from implementation-specific details. Poorly written code can therefore be more difficult to read and isolating any problems may become hard if the code is not modularized good enough. Several object-oriented principles like data abstraction, modularity, polymorphism and encapsulation has been utilized when writing the C-code for this thesis. This has proven crucial when debugging and testing the system.

The developed systems should be well tested. This can be a weakness in some of the previous master thesises on self reconfiguration. Because proof of concepts in the field take so much time to develop, it must be taken into consideration that some of the previous work has not been rigorously tested.

Testing and documentation has been given large emphasizing in this thesis. Code has been documented using the documentation system Doxygen and the test framework Check has been used for testing the different parts of the embedded software.

An iterative approach has in general been used for developing the methodologies and systems in this thesis. Experiments has been set up in test suites

and test cases.

The implemented system, the decisions taken and the results should be discussed. All of these can also be compared to available litterature on the field. Even if it may be hard to do so, weaknesses in the implemented system and any bad decisions in the design should be revealed and discussed. These experiences are highly valuable to people that want to do further development or others doing research in the same field.

Most important of all, the implemented system should function deterministically. A product that has low power consumption, high performance and low space requirements may be completely useless if it fails now and then. This is especially true for embedded and real-time systems. For a hard real-time system, missing a task's deadline means the task has failed completely and there is no reason to finish it. For an embedded system, it might be impossible to do any further development or bug-fixing after it has been installed in the field. It's even worse if the system has been mass-produced. In any case, if the system is non-deterministic and suddenly fails, this could become very costly, ruin the company's reputation or even threaten human lifes.

1.3 Contribution from this thesis

The most important contributions from this thesis are:

- A more robust software-part of the HWOS and a well-tested library for code used in this HWOS.
- A simple round-robin scheduler made specifically for partial reconfiguration of FPGA-based tasks.
- A proof of concept showing that partial self-reconfiguration of synchronous modules can be performed on an FPGA.
- A methodology for constraining and making a static interface for clock signals.

1.4 Outline of this report

The next chapters are divided into *Related work*, *Theory*, *Implementation* and *Verification and results*.

Related work presents some of the earlier work done in the AHEAD-project and related work done worldwide.

Theory is a chapter for introducing the reader to the most relevant background theory for this thesis. Some of the theory, like parts of the scheduling

theory, is quite general, but it should be discussed how these parts can be related to the problem in this thesis. Most of the references to the bibliography will from the *Theory* chapter.

Implementation shows how the final system has been implemented, what choices has been taken and why they were taken. This chapter will make heavy use of information and discussions from the *Theory* chapter.

Verification and results will document which parts of the implemented system was tested and results from the verification of the system.

Inside the *Theory*- and *Implementation*-sections there are also *Objective*-sections. For the *Theory*, these small sections will answer the question "why is this information provided?", while the question "why has this been done?" is answered in the case of the *Implementation*-part. The *Objective*-sections are marked with the symbol in figure 1.1 to make it extra clear that they should answer these questions and nothing else.

Figure 1.1: Symbol for *Objective*-sections.

Chapter 2

Related work

The most relevant earlier and related work is presented in this chapter. The work is briefly evaluated and some critisism has been given.

This chapter has two purposes. The first is to give the reader a better overview of the work done in the AHEAD-project and discuss some of the challenges the earlier work reveals. The second purpose is to introduce the most relevant research done on partial self-reconfiguration and hardware operative systems for FPGA.

2.1 Work done in the AHEAD-project

2.1.1 Sverre Hamre's master thesis (june 2009): Framework for self reconfigurable system on Xilinx FPGA

The most important work done by Sverre Hamre [Ham09] (seen from the perspective of this thesis) is probably the software *icap_write*. This work has been integrated in the scheduler implemented in this thesis and makes it possible to replace parts of the logic on the FPGA (*icap_write* is further discussed in section 3.4.3).

Sverre Hamre seems to have tested *icap_write* only on a few different partial configurations. Because the program is such an important part of the HWOS, it should be much better tested.

2.1.2 Vegard Endresen's master thesis (june 2010): Hardware-software intercommunication in reconfigurable systems

Essentially, Vegard Endresen's work [End10] is an implementation of a backend/interrupter for FPGA tasks. The backend is built in VHDL and is able

to extract and load data from/into a task running on the FPGA. This functionality can be controlled through an instruction set that is used by software applications to communicate with the backend. A state loading/saving module of this kind is of great importance and a critical part of HWOS. The fact that it is built in hardware makes it even more interesting when it comes to performance and parallel execution with the rest of the system.

There were some challenges when trying to understand and integrate Vegard's system in the HWOS. Perhaps could this have been easier if there was more documentation on the backend's internal mode of operation. The software-part of his work could also have been more well-structured. Vegard Endresen himself says the following in [End10, page 57]:

> Implementation on a Suzaku-sz410 board has confirmed that the system and its parts are working. That said some components of the communication framework has been designed without a very thorough pre-analysis. Primarily this is the case for the HWOS where there might be much to gain by proper analysis and design.

2.1.3 Sindre Hansen's project report (december 2010): Self Reconfiguration of Clock Networks on FPGA

The work done in the project report [Han10] functions as a base for the work done in this thesis. The results in the project has shown that a clocked module in a FPGA-bitstream can be pulled out from a source bitstream and successfully put into a target bitstream. This operation was done on a PC and not while the FPGA was running. The target bitstream was later uploaded to the FPGA and it was verified that it worked as expected.

The project report showed that partial reconfiguration of clocked modules could be performed, but not why it was possible or how this could be done at run-time. These are problems that would have to be solved if a working solution on the FPGA should be realized.

2.2 Related work on partial self-reconfiguration and HWOS

2.2.1 Klaus Danne (2004): Memory Management to Support Multitasking on FPGA Based Systems

This article (found in [Dan04]) discusses some of the challenges when sharing an FPGA for multiple processes. A simple run-time system is introduced and special focus is on the Memory Management Unit (MMU), which use the

concept og Virtual Memory Management to share distributed memory between several hardware tasks.

Chapter 3

Theory

3.1 Development tools

The development tool for FPGA development was the Xilinx Design Suite 10.1. Included in this suite was:

- Embedded Development Kit 10.1 for building user logic base designs.
- ISE 10.1 for building standalone reconfigurable modules.
- FPGA-Editor for inspecting placed and routed designs. Was also used for rerouting wires for logic.
- Planahead. Was used for setting constraints in both the base design and for the reconfigurable module.

For developing the software in this thesis, the following was used:

- The GCC-compiler for development on the development computer and for cross-compiling on the Suzaku.
- Valgrind for debugging memory leakage problems during the development process.
- The Check testing framework (see section 3.8.5) for testing the system.

3.2 Base platform

The development platform in this thesis is a development board from the japanese company Atmark-Techno. The board has an FPGA from Xilinx and an embedded microprocessor for implementing the reconfigurable systems. The website for Atmark-Techno has a download section containing both the base

VHDL-design for the FPGA and the Linux-distribution for the processor. The board also has LAN-connection for connecting it to a standard network. This makes it easy to connect to the board from a development computer and upload new FPGA- or software-designs.

3.2.1 Objective

The objective of this section is to present and give a quick overview of the basic setup for the development platform. This should make it easier to understand the results and discussions in the thesis. A new developer in the AHEAD-project will need to understand how the different parts of the platform can be set up and what kind of tools should be used.

3.2.2 The development board

The development board used in this thesis has the specifications shown in table 3.1 and a similar model is shown in figure 3.1. The illustrated model (SZ310-U00) has some different specifications, among them is the type of the FPGA, the CPU and the configuration unit.

A photography of the Suzaku board is shown in figure 3.2.

Model	SZ410-U00
FPGA-device	Xilinx Virtex-4 FX XC4VFX12-SF363
CPU Core in FPGA	PowerPC 405 (32bit RISC core)
CPU Clock	350 Mhz
Crystal Oscillator	100Mhz
DRAM	2*32MB DDR2
Flash Memory	8MB (SPI)
Ethernet	10BASE-T/100BASE-TX (half-duplex not supported)
User I/O Pins	86
Serial Port	1ch (RS232C)
Configuration	SPI Flash
Board Size	72x47 [mm]
Power Input	DC3.3V
Linux kernel version	2.6

Table 3.1: Specifications of Suzaku-V (SZ410-U00)

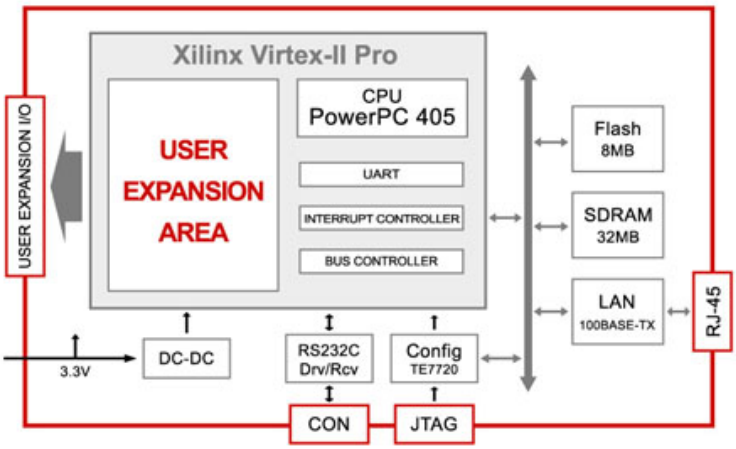

Figure 3.1: Illustration of the Suzaku-V (SZ310-U00). SZ410-U00 is used in this thesis. Picture taken from [AT11].

Figure 3.2: The Suzaku-V development board from Atmark-Techno.

3.2.3 The FPGA

A photography of the FPGA is shown in figure 3.3 and it's specifications are listed in table 3.2.

Manufacturer	Xilinx
Model	Virtex-4 FX XC4VFX12-SF363
Speed grade	-10
CLB Array: Row x Column	64 x 24
Number of slices	5 472
Number of LUTs	10 944
Maximum Distributed RAM or Shift Registers (Kb)	86
Number of flip-flops	10 944

Table 3.2: Specifications of the FPGA used in the thesis.

Figure 3.3: Photography of the FPGA on the development board.

3.2.4 Base VHDL-design for the Suzaku-V

The base design from Atmark-Techno contains the basic setup for communicating with the parts on the Suzaku-V board.

3.2.5 Internal Configuration Access Port (ICAP)

The Internal Configuration Access Port (ICAP) allows for internal access to the FPGA bitstream at runtime. The embedded processor can communicate with ICAP through the Processor Local Bus (PLB) (as described in [Han10] and [Xil08b]).

3.2.6 ATMARK-dist and uClinux-dist

ATMARK-dist is a Linux-distribution made specifically for the Microblaze- and PowerPC-processors on the Suzaku-boards. The distribution is based on uClinux-dist and is built around the standard Linux 2.6-kernel [Atm06].

The original uClinux was a derivative of the Linux 2.0 kernel and was intended for microcontrollers without Memory Management Units (MMU)

[Inc10]. It was later integrated in the main line Linux kernel sources, starting from Linux-2.5.46 [Ung02]. uClinux-dist is a collection of libraries, applications and tools, where the most important parts probably are the uClinux-kernel and the C standard library called uClibc.

A tutorial for setting up ATMARK-dist and NFS is provided in chapter A in the appendix.

3.2.7 uClibc

uClibc is a C library for the Linux platform. It is intended for embedded systems and is much smaller than glibc, which is the C library typically used when developing applications in Linux.

uClibc supports many of the same applications as the heavier glibc and often it is possible to just switch libraries and recompile the source code [ucl11]. This assumption has been important when writing the C code for this thesis. The code has been written and tested on the development computer, recompiled for the PowerPC-architecture and tested more extensively on the embedded platform.

3.3 Addressing the bitstream for Virtex-4

The addressing in the bitstream for the Virtex-4 has been discussed in [Ham09, page 19]. The term *frame* and what one such frame may contain can be rather confusing. This subsection makes the meaning of a frame clear by evaluating the existing framework, articles on the subject and the documentation from Xilinx.

3.3.1 Objective

The most important goal for this section is to define a frame and what meanings one frame or a series of frames can have in the bitstream or in the FPGA configuration memory. Another objective of this section is give a brief overview of how addressing of frames can be done in the bitstream file.

Understanding what a frame is and how frames can be addressed is very important if the programs *CLBRead* or *icap_write* should be utilized in an application or further developed.

3.3.2 Definitions

Bitstream file
A file made by the FPGA development tools. Does contain information to reconfigure the configuration memory on the FPGA, but also operations and overhead data for doing this.

Configuration memory
The configuration memory on the FPGA. Virtex-4 has SRAM-based configuration memory. These memory cells define the LUT equations, signal routing, IOB voltage standards and all other aspects of the user design [Xil09, page 87].

Frame
One frame is 41 32-bit words = 1312 bits [Xil09, page 92]. In the bitstream, 22 frames configures 8 CLBs + 1 HCLK + 8 CLBs. In the configuration memory, *one frame* constitutes 8 CLBs + 1 HCLK + 8 CLBs [TBC07].

Major column of CLBs
One full, vertical column of CLBs on the FPGA. Each column is one CLB wide.

Row of CLBs
A row spans the entire FPGA horizontally and it's vertical width is 8 CLBs + 1 HCLK + 8 CLBs.

Partial reconfiguration
Reconfigure only a part of the configuration memory.

Self reconfiguration
Defined here as that the embedded system performs the reconfiguration of the FPGA itself. A more strict definition could be that the FPGA reconfigures itself.

3.3.3 Addressing of frames

The FPGA used in this thesis is 16*4 = 64 CLBs in full height. A *major column of CLBs* is one full height of CLBs, one CLB wide. This FPGA has four *rows*, where each row spans the entire FPGA horizontally and is 16 CLBs in height. These definitions correspond with the definitions in [Xil09, page 92].

In [Ham09, page 10], Sverre Hamre lists up several articles that describe bitstream composure for Xilinx FPGAs. The article in [TBC07] is especially interesting as it describes a lot of the bitstream details for Virtex-4. One can imagine that these articles has been important when he wrote the program *icap_write*.

Figure 3.5 is from Sverre Hamre's master thesis [Ham09]. As shown in this figure, there should be 22 frames per column of CLBs on the FPGA. It is assumed that Sverre means there are 22 frames per *column of 16 CLBs*. Figure 3.6 is also from his master thesis and Sverre describes this as the organization of frames in the bitfile. It is assumed that this shows the organization of frames per row. Both assumptions correspond to the functionality of *CLBRead* and *icap_write*, as well as the article in [TBC07].

Figure 3.4) is from [Han10] and show what a frame may constitute in the *configuration memory*.

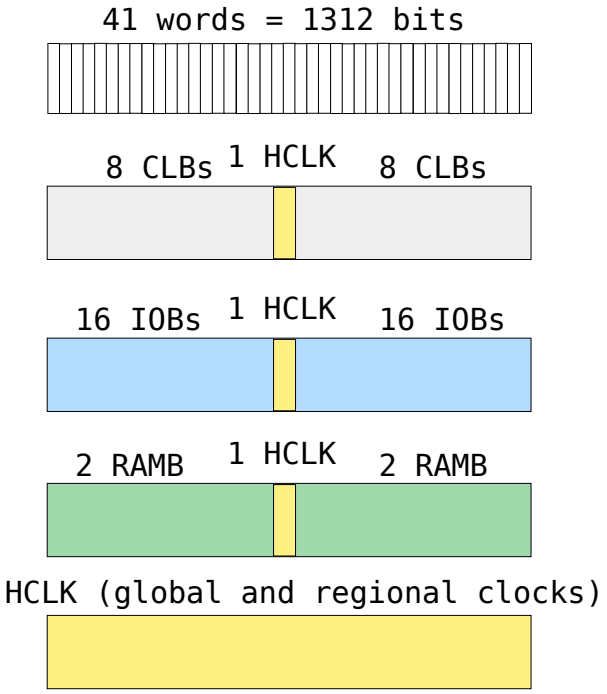

Figure 3.4: Number of words in a frame and example of what frames in the configuration memory may contain (picture taken from [Han10]).

In [Ham09, page 20], Sverre Hamre writes:

> In the bitstream the CLBs consist of 22 frames, 20 of these are for the routing and 2 are for the logic.

As discussed above, it is assumed that he means 22 frames per column of 16 CLBs. However, later on page 21 he says:

> A frame in the bitstream is 41 words, 1312 bits, this is equal to 16 CLBs plus one word in the center of the bitstream for global logic.

As the discussion in this section reveals, the last statement seems to be wrong as one frame in the bitstream does not alone configure so many CLBs. Sverre probably meant the configuration memory.

At last, figure 3.7 is added to show what the meaning of a frame may be in the bitstream or the configuration memory.

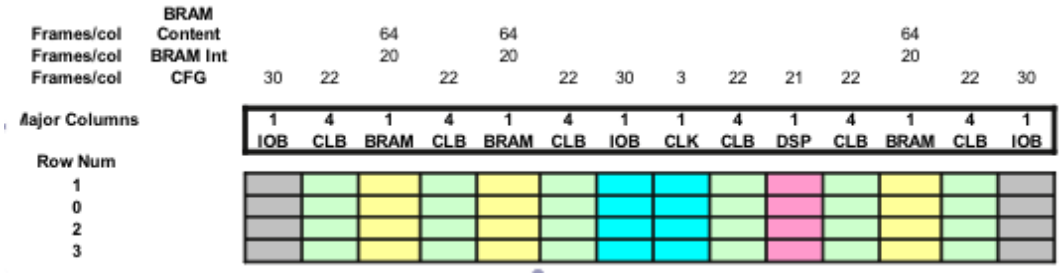

Figure 3.5: Virtex-4fx12 FPGA logic modules (picture taken from [Ham09, page 20]).

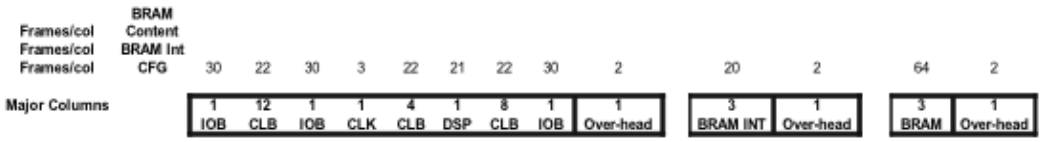

Figure 3.6: Virtex-4fx12 bitfile frame organization (picture taken from [Ham09, page 20]).

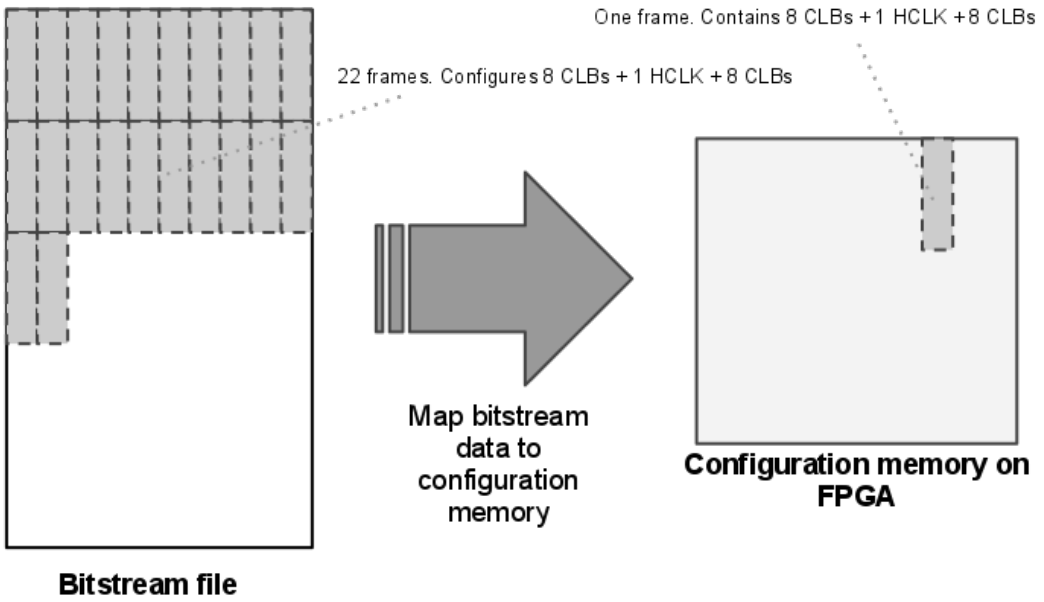

Figure 3.7: Conceptual drawing of the meaning of a frame on Virtex-4. Examples from the bistream and the configuration memory.

3.3.4 The term frame in CLBRead

A short extract from the defines in the CLBRead header file is shown in listing 3.1. The comments has been added here to show what role each define play in the actual program. The use of the term *frame* is quite confusing also in this file. However, if only the defines used for addressing are considered and `FRAMES_PR_CLB` is interpreted as *frames per column of 16 CLBs*, everything seems to be correct and matching the previous discussion in this section.

```
1  (...)
2  // Used for addressing in CLBRead.c
3  #define CLB_PR_ROW                    24
4  // Only used for debugging in CLBRead.c
5  #define CLB_PR_FRAME                  16
6
7  // Used for addressing in CLBRead.c
8  #define FRAMES_PR_CLB                 22
9  // Never used in CLBRead.c
10 #define BIT_PR_CLB                    80
11 // Only used for debugging in CLBRead.c
12 #define BYTES_PR_CLB                  BIT_PR_CLB/8
13 (...)
14 // Used for addressing in CLBRead.c
15 #define FRAME_LENGTH_32BIT_WORDS      41
16 // Used for addressing in CLBRead.c
17 #define FRAME_LENGTH_BYTES            FRAME_LENGTH_32BIT_WORDS*4
18 (...)
19 // Never used anywhere
20 typedef struct{
21    uint8_t iCLB[BYTES_PR_CLB*FRAMES_PR_CLB];
22    }clb_t;
23 (...)
```

Listing 3.1: Short extract from CLBRead.h. The omitted parts are marked (...).

3.3.5 The term frame in icap_write

The header file for *icap_write* has the line `#define WORDS_PR_FRAME 41` and one has to specify how many frames to write. The program takes 22 frames per *column of 16 CLBs*. This seems to be correct behaviour and has been verified by testing the program.

3.3.6 The term frame in the documentation from Xilinx

The Configuration User Guide in [Xil09, page 92] states the following:

> All Frames in Virtex-4 have a fixed, identical length of 1312 bits (41 32-bit words). One Frame configures one HCLK with either 4 block RAMS, 32 IOBs or 4 DSPs.

This presumably means that one frame in the *configuration memory* constitutes one HCLK with either 4 block RAMs, 32 IOBs or 4 DSPs. Note that the

documentation from Xilinx is rather sparse on information on the bitstream composure. It could for example not be found anywhere that 22 frames in the bitstream configures a column of 16 CLBs.

3.4 The existing framework for partial self-reconfiguration

In this section, the most important parts for the exisiting system for partial reconfiguration will be introduced. Some terms have been discussed in [Han10] and in earlier reports from the AHEAD-project. In that case, only a brief definition and a reference to the relevant report will be given.

3.4.1 Objective

The goal of this section is to give the reader an overview of the existing framework for self-reconfiguration on the FPGA, *before* the contributions from this thesis are introduced. This is important, because much of the work done in this thesis builds upon the earlier work in the AHEAD-project.

3.4.2 CLBRead

As discussed in [Han10, page 16], CLBRead is a program for reading out a partial CLB structure from a bitfile. It can also write a partial bitstream into another bitfile. In this case, the reconfiguration is done by modifying a bitstream that has not been uploaded to an FPGA. An important finding from [Han10] is that a partial bitstream containing routing information for both CLBs and clocks can be extracted and inserted into another bitstream. It was also shown that a flip-flop and it's corresponding clock signal can be relocated to another placement in the bitstream. This indicates that *synchronous modules* can be inserted into and relocated in a target bitstream file.

The greatest limitation with these results is that they have not been performed at run-time. One problem source that will have to be considered when performing reconfiguration at run-time are oscillating signals, especially when reconfiguring a high frequency clock signal. A clock buffer (see section 3.6.4) can be used for disabling the clock signal during reconfiguration and synchronous bus macros can be used for disabling normal signals into the module (see section 3.4.4). These considerations are important for practical designs, but are probably not necessary for showing a proof of concept for partial reconfiguration.

Another problem source is that propertiary protocols from Xilinx will have

to be used for writing the bitstream to the FPGA at run-time. The CLBRead program is open-source, thus making it easier to isolate problems when doing research on the area. For this matter, the analysis of the run-time reconfiguration process done by Sverre Hamre in [Ham09] acts as supporting documentation when doing any further development.

3.4.3 icap_write

Concept

CLBRead's ability to write a partial bitstream to another file has not been used in this thesis. Instead, the partial bitstream is written to a running configuration on a physical FPGA through the Internal Configuration Access Port (ICAP). Essentially, this means that the file containing the partial bitstream and a software program for writing to ICAP has to be present on the Suzaku platform. Sverre Hamre has in his thesis [Ham09] written the C-program *icap_write*. This program runs on the PowerPC-processor on the Suzaku, takes a bitfile description of a partial bitstream as input and writes it to the ICAP port. It is very important to note that this program has not been tested thorughfully, or as Sverre Hamre states (from [Ham09]):

> The icap_write and icap_test programs are written to utilize the Linux icap driver. These are just test programs, especially since they have hard coded the adressing and has a lot of printf() for debugging.

Despite that icap_write is a test program, it is well written and easy to understand. Sverre Hamre has provided good documentation of the process of writing to ICAP in his master thesis.

Practical use

```
1 ./icap_write -h
2 icap_write -i [filename] -f [frames]
3 This program has the address hardcoded inn.
4 Frames will be written to CLB 21 and on, on top row.
```

Listing 3.2: icap_write help

As seen is listing 3.2, the program takes two inputs. The first is the filename of the bitfile and the second is the number of frames to be written.

Because one vertical series of 16 CLBs is the smallest possible configurable unit on the Virtex-4 and because the original program by Sverre Hamre has the adressing hardcoded in the program, *icap_write* has been rewritten in 5.13. It is now a part of the HWOS-library and the library module is called *hicap*. Refer to that section to see how this functionality is used.

3.4.4 Bus macros

A bus macro is essentially just a static interface between the module and the static design (figure 3.8). One simple analogy to this is a standard power outlet in a house wall where different devices can be connected, always using the same type of plug.

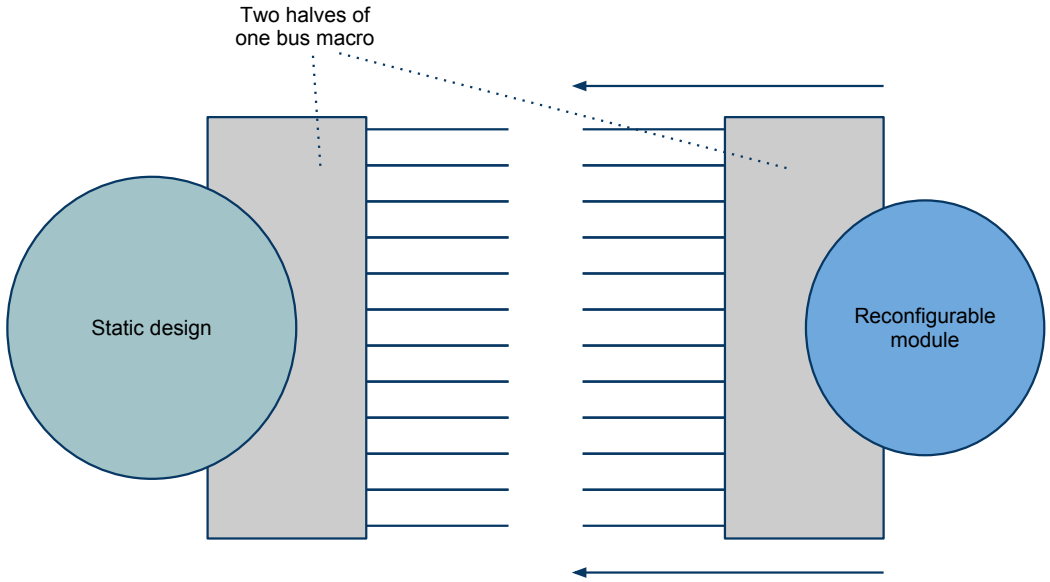

Figure 3.8: Conceptual drawing of a bus macro.

Normally the wiring between two adjacent CLBs are dynamically routed during the place-and-route phase of the implementation process. The problem is that the module and the static design are implemented separately and that the routing between them will not be consistent.

How a bus macro is made in FPGA-editor is described in a tutorial by Sverre Hamre [Ham08b]. The most simple bus macro is made of two adjacent CLBs and a static, predefined routing between them that does not change between implementations. The number of wires between the two CLBs will be technology dependent, but a wider bus macro can of course be made by setting up several pairs of CLBs.

As shown in figure 3.8, one half of the bus macro is integrated in the reconfigurable module and the other half is integrated in the static design. To achieve this, the reconfigurable module and one complete bus macro is built together in an empty FPGA-design. Using the program CLBRead (section 3.4.2), the reconfigurable module and one half of the bus macro is cut out and stored in a partial bit file.

3.5 Synchronous design

3.5.1 Objective

Because the thesis focus on methodology for partial reconfiguration of synchronous modules, the reader should understand what a synchronous module is and why synchronous design is important on an FPGA. Essentially, this section seeks to answer why it is so important that a clock signal can be routed to the reconfigurable module.

3.5.2 Definitions

Synchronous design
A clock signal triggers all events [Ste05, page 4].

Synchronous reconfigurable module
Is in this thesis defined as a module using the same or a derived clock signal from the static base design.

3.5.3 Motivation for using synchronous design

In [Ste05], Jennifer Stephenson from Altera has discussed some fundamental design practises for synchronous designs on FPGA. On page 4, she writes:

> The basic principle of synchronous design is that a clock signal triggers all events. As long as all of the registers' timing requirements are met, a synchronous design behaves in a predictable and reliable manner for all process, voltage, and temperature (PVT) conditions. Typically, designers can easily target synchronous designs to different device families or speed grades.

Her words says a lot about the motivation for using synchronous modules in a reconfigurable framework. Making the module synchronous to the clock signal in the base design can be seen as a way of letting the module "inherit" the timing of the framework. Because the interface for control signals and data transfer is the same for all the reconfigurable modules, it is important that each module has predictable timing behaviour.

3.5.4 Problems in asynchronous designs

There has been a lot of research on asynchronous design, both for ASICs and FPGAs. One of the largest problems is that the main-stream FPGAs and

their tools and not made for this type of design practise. The analysis tools for FPGA designs are tailored for synchronous designs and it is easy to verify that flip-flop setup and hold times are met under worst case conditions for such designs [Eri00]. For asynchronous designs, the designer would have to manually check the worst case timing of the signal paths, a process that would be time-consuming or maybe impossible for complex designs. Even worse is that this process must be redone for small changes in the design or if the design is migrated to another FPGA.

Another problem is handling asynchronous input values to flip-flops. The registers in FPGAs have a defined interval where the input value must be stable to ensure that a reliable output signal is achieved. Specifically, the input value must be stable for a minimum time of `t_su` (register setup time) before the positive clock edge and a minimum time `t_H` (register hold time) after the edge. The output value is then available after some time `t_co`. The problem is if the signal transition violates the `t_su` or the `t_H` requirements. In this case, the output may go into a *metastable state* and the output of the register may be delayed and not be available withing the required time `t_co` [Alt09]. For synchronous designs, the input signal to the registers must always meet the requirements, so the problem with *metastability* does not occur [Alt09].

3.5.5 Timing requirements for a reconfigurable module

The types of communication between the framework and the reconfigurable module can be divided into *control signals* and *data transfers*.

Control signals

Signalling that the module should start or that a data transfer is finished are typical control operations. If it is impossible to know when the operation has been performed (such as when the module has been started), hand-shake signals should be used. Signals for initiating operations and for signalling that operations has been done must probably be present for both synchronous and asynchronous modules. As discuessed earlier, the problem for asynchronous modules would be performing the actual operation without a clock signal to relate to.

Data transfers

Data transfers are more complex. The framework would typically implement some sort of serial communication to transfer data to and from the module. The state of the module is an example of data that must be transferred this way. In the implementation by Vegard Endresen [End10], one or more parallel shift-registers are used for this transfer. In this case, the framework assumes that the module shifts out one bit of data each clock cycle. This is a safe

and deterministic way of doing high-performance data transfers if the clock signal in the module has low skew and is synchronous to the clock signal in the framework.

On the other hand, if the module was asynchronous, some sort of hand-shaking mechanism would have to be used. An example implementation would function this way:

1. The framework signals to the module that one bit should be shifted out by setting `SHIFT` high.
2. The module shifts out one bit and sets a hand-shake signal, `SHIFT_DONE`, high.
3. The framework sets `SHIFT` low and goes back to the first step.

The problem for this is that many control signals would have to be used for the hand-shaking. Because of this, naive implementations would probabably have poor performance.

3.6 Defining an interface for clock signals

3.6.1 Objective

The goal of this section is to provide background theory for the development of a methodology for creating partially reconfigurable, clocked modules. This will discuss how to make ready an initial base configuration to put on the FPGA and how to make reconfigurable modules.

3.6.2 Definitions

Synchronous module
: In this thesis, this is defined as a reconfigurable module with the same or a derived version of the clock signal from the base design.

3.6.3 Concept

In this thesis, Directed Routing (see section 3.6.7) is a key concept for making it possible to perform partial reconfiguration of synchronous modules. The method can be used to put constraints on the routed clock signal for the modules. To do this, a *dummy design* is first built in ISE. This design has one global clock buffer and one flip-flop as shown in figure 3.9. Note that

the use of the global clock buffer (BUFGCE) must be clearly expressed in the VHDL-code. This is to make sure that at least one separate clock buffer is used to route the clock signal into one or more of the reconfigurable regions. If the same global clock buffer was to be used for both the static and the reconfigurable part of the design, this would mean the clock signal would be routed to a lot of different logic and the DIRT would be harder to maintain as explained in section 3.6.7.

The simple dummy design is synthesized, routed, placed and the resulting NCD-file is opened in FPGA-editor. Now this tool can extract a short and concise user constraint for the global clock buffer, the clock signal and the flip-flop.

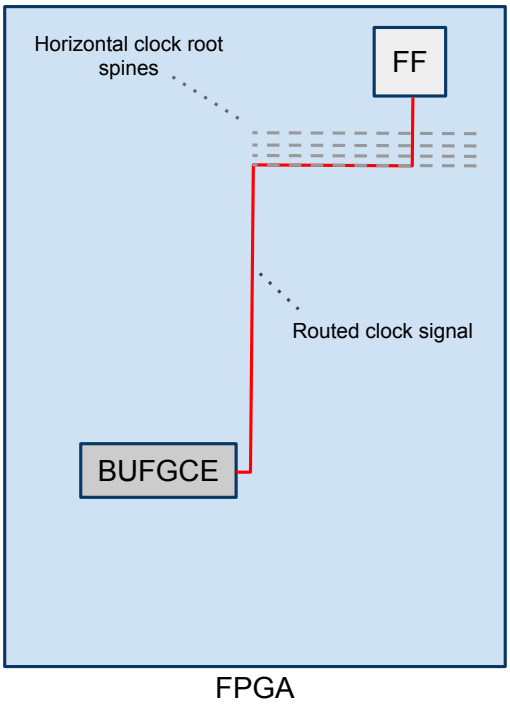

Figure 3.9: Dummy design containing one flip-flip and one global clock buffer (BUFGCE)

A few of the global clock root spines for the specific region is also drawn for illustration. The purpose is to show that the clock signal will be routed along the lowest and first available global clock wire. The routing information from this dummy design is put into the complete, static base-design, thereby guaranteeing that the same clock buffer and, most importantly, the same clock wire always will be used for any reconfigurable module. The dummy design actually defines *an interface for clock signals* into the reconfigurable module. In this simple example it is defined that any reconfigurable module placed in the upper, right corner of the FPGA is able to use the global clock buffer specified in the dummy design and that this clock signal will be routed along the lowest global clock wire. Any reconfigurable module built in ISE will always use the

lowest clock wire as this is the first one that is available. Because the base-design has been built with the clock-routing-information from the dummy-design, any synchronous, reconfigurable module will be compatible with this design.

3.6.4 Global clock buffer, BUFGCE

The BUFGCE primitive is one of the available global clock buffers on the Virtex-4 FPGA. It has one input and one output port for the clock signal and the input signal `CE` for enabling/disabling the output clock signal. All the global clock buffers are derived configurations from the BUFGCTRL primitive as described in [Han10].

3.6.5 Clock root spines

The different clock resources on the Virtex-4 FPGA was introduced and discussed in [Han10]. One important point was the distribution of the clock signals to the different clock regions on the FPGA. Both global and regional clock signals are routed along *horizontal root spines* as shown figure 3.10. Note that the Virtex-4 has 8 horizontal root spines, while Virtex-5 has 10.

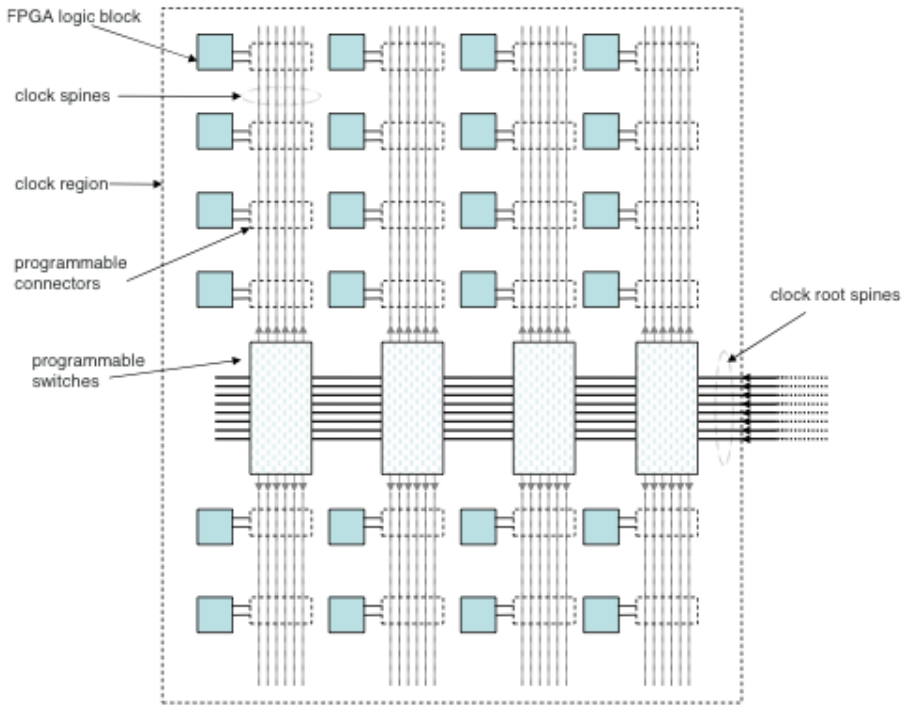

Figure 3.10: Distribution of global clock signals into a clock region (Virtex-5). Picture taken from [QWA09].

The connection of global clock buffers to the global clock lines on the Virtex-4 FPGA are shown in figure 3.11. The picture shows one global clock buffer

configured as a `BUFGCE` (see section 3.6.4).

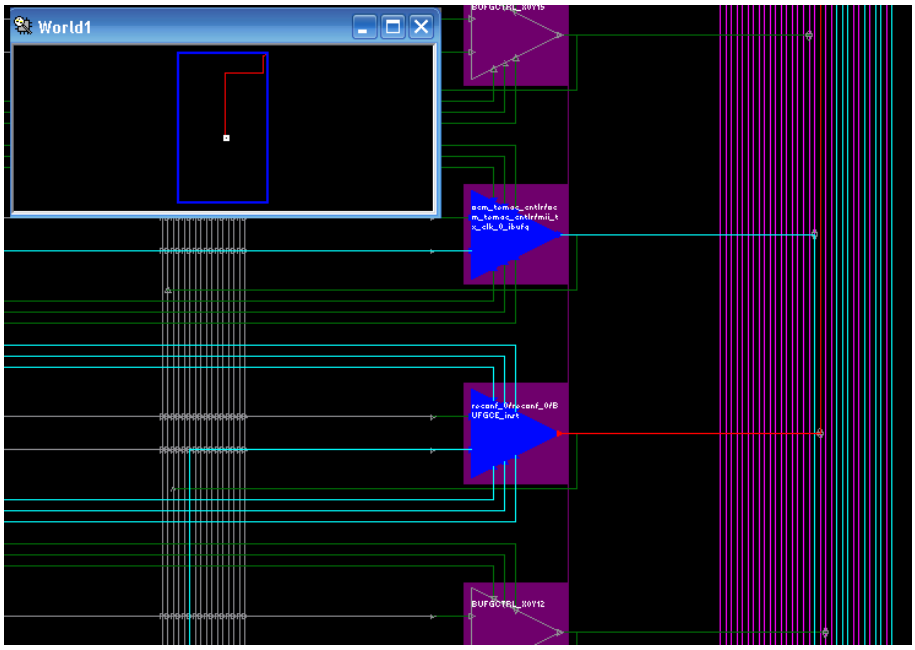

Figure 3.11: Connection of global clock buffers to global clock lines on the Virtex-4. Screenshot from FPGA-editor.

The clock signals are routed along *vertical root spines* as shown in figure 3.12. There are several switchboxes along the vertical spine, making it possible for the routed clock signal to turn left or right into a specific clock region. In this figure, the clock signal is routed into the upper right region of the FPGA.

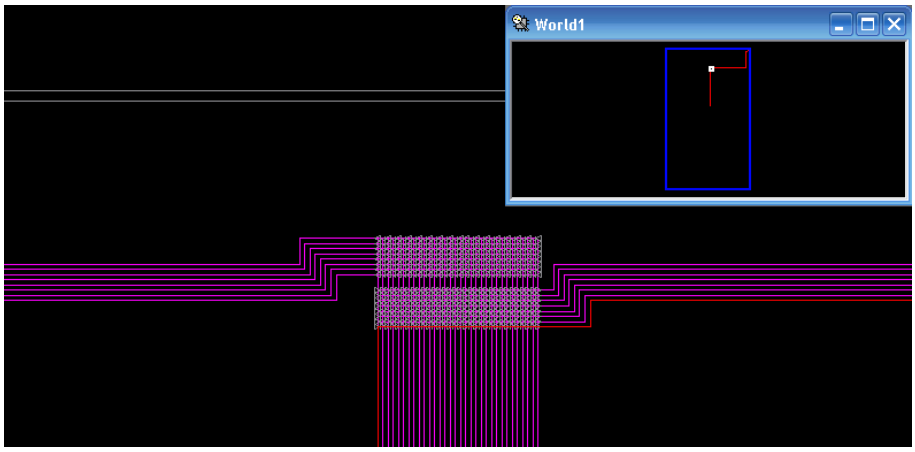

Figure 3.12: Distribution of global clock signals in the middle of the Virtex-4. The routed clock signal is marked red. Screenshot from FPGA-editor.

Figure 3.13 shows a screenshot of the horizontal root spine on the Virtex-4 FPGA. The clock signal is routed horizontally from the middle of the FPGA and then switched up to the CLBs. Note that there are 8 global clock wires. The two lower wires in figure 3.13 come from regional clock buffers (see [Han10] for more information on these).

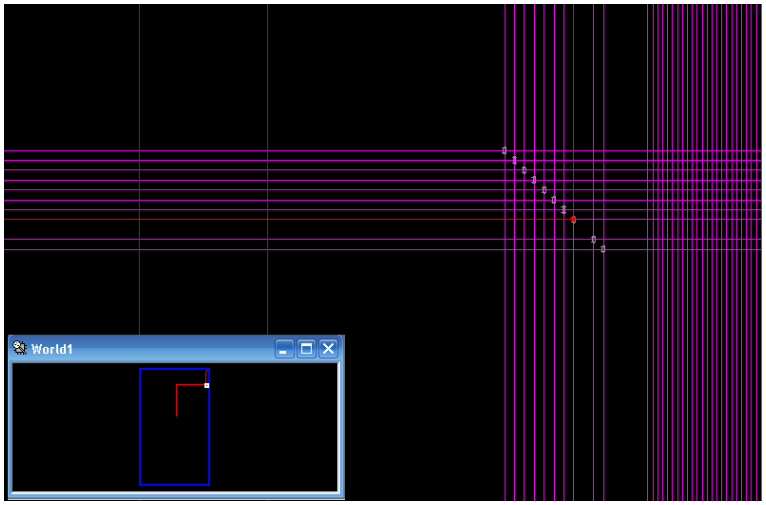

Figure 3.13: Distribution of global clock signals in the upper right corner of the Virtex-4. The routed clock signal is marked red. Screenshot from FPGA-editor.

3.6.6 User Constraints File (UCF)

The User Constraints File (UCF) are used during place and route for the FPGA logic. Examples of constraints are placement of logic blocks, usage of resources like BRAM and timing constraints. Constraints can be written directly into the UCF-file, but this is hard for complex constraints. The tool Planahead are typically used for location constraints.

An interesting type of constraint for this thesis is Directed Routing as discussed in section 3.6.7.

3.6.7 Directed Routing (DIRT)

Directed Routing (DIRT) is meant to be a method for defining repeatable, locked routing functionality for a limited number of critical signals in a design via UCF constraints [Xil08a]. Generally, this can be useful for maintaining determinable timing of nets in the design or as a potential workaround for routing limitations introduced by the tools. The routing information for one routed signal can be extracted and reused in the same or in another design.

This kind of constraint is typically extracted from a design that already has been placed and routed. This can be done by opening the NCD-file in FPGA-editor and choosing the menus `Tools` → `Directed Routing Constraints`. This will open a dialogbox with several configuration options, which essentially will export a Directed Routing constraint to raw text that can be put in a User Constraints File (UCF). Directed Routing specifies the exact routing between two or more components on the FPGA. This means that constraints for these relevant components also must be extracted if the DIRT should func-

tion correctly. This can impose a problem when the same signal is routed to many components as the DIRT will be both large and have strong dependence on the original design. Also, the extracted text constraint are not actually human-readable, which means it is not easy to maintain manually. In practise one have to use the tools from Xilinx to generate such a constraint, but it might be possible to reverse-engineer the format to understand how they are made.

3.6.8 Setting up a base design

Before tests on swapping reconfigurable modules can be performed, a VHDL-design must be set up on the FPGA. A VHDL-project containing a minimum configuration for the FPGA can be downloaded from Atmark-Techno's web pages (elaborated in 3.2.4).

The base design can be built with an initial reconfigurable module or at least open room for a module. The reconfigurable module must be connected to the rest of the design through so-called bus macros (described earlier in section 3.4.4).

3.6.9 Setting up a reconfigurable module

There are two ways of creating a reconfigurable module in VHDL. The first one is to include the reconfigurable module in the process when designing the base design. This is typically done in EDK. The other way is to design only the reconfigurable module in ISE (description of tools is in section 3.1).

Either way, it is important to put constraints in the design to make sure the bus macros and the reconfigurable module are placed on deterministic locations. Constraints are put in a User Constraints File (UCF), which are a plain text file that can be altered manually or by the program Planahead (see section 3.6.6). How to put constraints on logic is discussed in [Han10] and in [End09a] and will not be described in detail here. The most important logic constraints for a reconfigurable module are summarized in the following subsections. Figure 3.14 works as a reference for these sections.

Constraints in the base design

For the base design it is important to define a reconfigurable region where one or more reconfigurable modules can be placed.

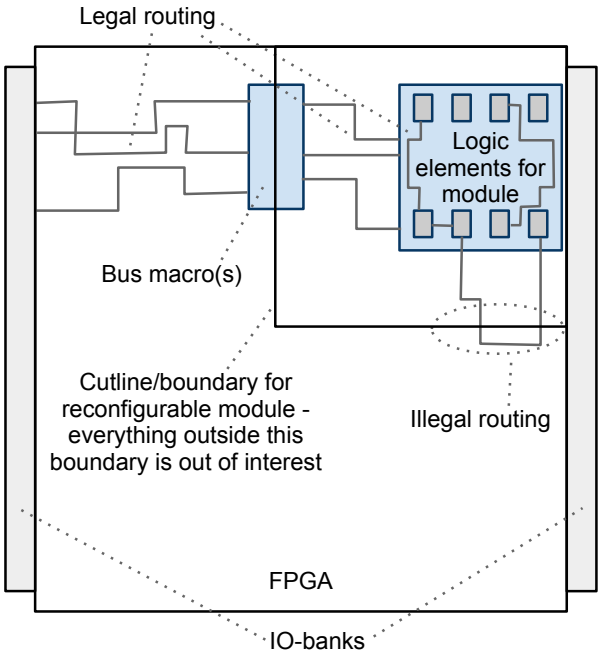

Figure 3.14: Conceptual drawing of a reconfigurable module built on an empty FPGA in ISE.

Constrain logic blocks for a reconfigurable module

In most reconfigurable systems, it is convenient to group the logic blocks for a module as close together as possible. The idea is that communication will be faster when the logic is in the same group. In this thesis, the internal communication for a module goes through direct wiring. This should work as a quite general solution, but it is worth mentioning that some systems use a dedicated Network on Chip (NoC) both for communication between modules and for the internal communication in a module.

Constrain routing for a reconfigurable module

When building a standalone module in ISE, the internal wiring in the module should of course not extend outside the region defined for that module (see "Illegal routing" in figure 3.14). It makes sense to keep the boundaries of this region as tight and close to the logic as possible, but this is also dependent on the routing resources available inside the module.

One significant problem is that routing can not be easily constrained in the implementation process. In general, this means that the Xilinx tools will consider the whole FPGA when making routing choices and manual rerouting of wires must be performed to make sure that routing is held inside the module's region. The rerouting process is done in FPGA-editor before generating a bitfile and can be quite time consuming for complex modules. If the module is changed and must be synthesized one more time, the rerouting process must

be redone and even more time is wasted. Not that this is not the case for clock signals. A methodology for constraining the clock will be presented in 3.6.10.

One possible workaround for normal signals could be to create a hard macro or use Directed Routing (section 3.6.7) to occupy all or most of the routing resources on the FPGA, except the resources in the reconfigurable region. This method has not been tried out, but could potentially force ISE to only use routing resources in that region.

Another smaller problem is the routing from the bus macros to the IO-banks (see figure 3.14).

3.6.10 Making base design compatible with synchronous modules

As described in *Concept* (section 3.6.3), some steps must be done to make sure the base design are compatible with reconfigurable modules.

A dummy module can be set up using just one clocked flip-flop and one global clock buffer. A VHDL design for this is shown in listing C.2. As described earlier, this design is opened in FPGA-editor and choosing the menus `Tools` → `Directed Routing Constraints`. As shown in figure 3.15, a window is opened and all the routing resources in the module are listed. In this example, the output signal from the clock buffer, `clk_buffer_out`, has been selected.

Note that both a relative and an absolute constraint can be generated. To make sure the routing is followed exactly, the last option is chosen. After clicking "OK", the UCF-constraint shown in listing 3.3 will be written to a file. As can be seen, this code put constraints on the global clock buffer, the flip-flop and the routing between them.

Listing 3.3: DIRT constraint for dummy module built in ISE

```
1 NET "clk_buffer_out"
2 ROUTE="{3;1;4vfx12sf363;fd3a8469!-1;1112;-11800;S
      !0;376;11925!1;31897;"
3 "82119!2;23663;-787!3;2853;26551!4;683;-496;L!}";
4 INST "BUFGCE_inst" LOC=BUFGCTRL_X0Y13;
5 INST "R/out_16_0" LOC=SLICE_X46Y127;
6 INST "R/out_16_0" BEL="FFY";
```

This constraint is put in the UCF-file of the base design. It should not matter what kind of static design this is as long as the initial reconfigurable module is the module with one flip-flop. It must be the exact same one as described here.

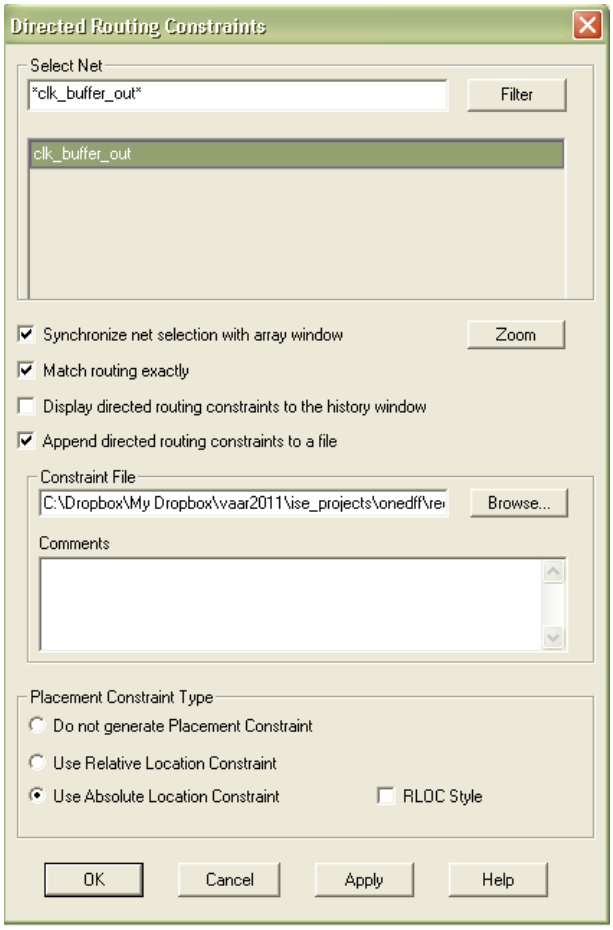

Figure 3.15: Screenshot from Directed Ruting Constraints menu in FPGA-editor.

3.6.11 Making a scalable solution

It should be possible to use several clock buffers in the complete design. It could for example be useful to have different clock frequencies on different modules. The general problem is to route a set of clock buffers `BUFFER0`, `BUFFER1`, `BUFFER2` and so on to a set of modules `MODULE0`, `MODULE1`, `MODULE2` and so on. A brief description of methodology for the reconfigurable module and the static design follows.

The static design

For the static design, this is quite easy. Just build a dummy module with all the clock buffers and one flip-flop per clock buffer (`FF0`, `FF1`, `FF2` and so on). Iterate the design flow as follows:

1. Build a dummy module with only `BUFFER0` and `FF0` in ISE.
2. Extract the DIRT for this clock signal and place the constraint in the

UCF-file of the same ISE-project.

3. Rebuild the dummy module with the same buffer and flip-flop as before, but also `BUFFER1` and `FF1`. `FF1` is placed in a different reconfigurable region than `FF0`. The two regions will now represent different clock-domains. A module placed in the first region will use the output of `BUFFER0` and a module placed in the second region will use the output of `BUFFER1`.

4. Extract the DIRT for the clock output of `BUFFER1` and place it in the same project.

5. Rebuild the project again and do the same operations for `BUFFER2` and any other buffers.

6. At last: Place all the DIRT-constraints in the UCF-file of the static design. Build the static design with the same set of intitial, reconfigurable modules.

After this is done, the static design should be compatible with the rest of the design. The FPGA should be divided into several reconfigurable regions, each region potentially using the output from a different clock buffer.

The reconfigurable module

It is assumed that each reconfigurable module use only one clock signal each. In this case there is no need to constrain the clock signal, as the first clock wire in to the module always will be used.

3.7 Scheduling on FPGA

3.7.1 Objective

The objective is to provide background information on how scheduling is done for computer systems in general, for embedded systems and for partial reconfigurable systems on FPGA. In this thesis, this theory forms a background for the choices done when implementing the scheduler in section 5.

3.7.2 Motivation

In this thesis, an FPGA is used as a reconfigurable hardware accelerator and the overall goal is to increase the overall speed of the system. Processes that run on the FPGA can be interruptible, meaning that after they have been

placed on the FPGA, they can be stopped. The process state can be saved to for example a file on disk or to some random access memory. The FPGA area can then be utilized by some other process that needs run-time.

[Wol05] discusses hardware accelerators and scheduling on hardware. As said in this article, moving functionality from software to a hardware accelerator must of course only be done if the task can run faster on the hardware than on the CPU. This can be a challenge for a high-performance CPU, but the fact that hardware modules can run in parallel with other modules and the CPU increases the chance of speed-up. Furthermore, the system should be analyzed and the bottlenecks should be identified. Speeding up one part of the design in hardware when in fact another part is the bottle-neck may not add any net-gain to the complete system. Yet another challenge is communication and synchronization between the accelerated module and the rest of the system. The communication cost between two processes on the same processing unit is in general less than the cost between processes on different units. Even if a module runs faster in hardware than in software, the communication costs may be to large to gain any real speed-up.

3.7.3 Type of scheduling decisions for reconfigurable hardware

For a software programs running on a single CPU, a scheduler will be in charge of determining *when* each process should run on the CPU. This will be more complex if there are multiple CPUs where the process can run. It depends on the definition of *a scheduler* whether or not it is the scheduler's responsibility to also decide *where* the process should be executed. In this thesis, the following are defined.

Partitioning module

The partitioning module is responsible for partitioning the application between hardware and software.

Scheduler

In this thesis, scheduling is defined as deciding *when* a process should run.

Placer

The placer is defined as the part of the HWOS that decides *where* the process should be placed on the FPGA. This can be done in an 1-dimensional or 2-dimensional manner. For 1D-placers, many of the same concepts as for virtual memory can be used.

If a reconfigurable system, like an FPGA and a coprocessor, has maximum flexibility, a process can run in both hardware and software, on an arbitrary place on the FPGA and in any given time. For such a system, the decision taken by the scheduler may of course depend on the results by the partitioning module and the placer. For example, it may not be a good idea to schedule a process for run-time on the FPGA if it must transfer a lot of data to another module that is running in software. Also, it may not be a good idea to place a module on the FPGA if it is to large to fit within the exisiting configuration.

3.7.4 Process

In this thesis, a process is defined as an entity owned by some module, either a software or a hardware module, that is running on the FPGA for hardware acceleration. For example can a software application ask the hardware scheduler to perform multiplication operations on the FPGA. If a multiplicator is present on the FPGA, the process can utilize the multiplicator logic to perform it's operations in cooperation with the larger software application. In a certain way, one can see the multiplicator logic on the FPGA as a class and the process as an object of that class. A process has state data and input and output values bound to it, while a reconfigurable module is logic acting upon these input and state values.

Whether a process is I/O-bound or processor-bound is an important classification for any type of scheduler.

I/O-bound processes

This type of process makes heavy use of I/O-resources. This can for example be a process that is awaiting keystrokes from a user. Such a process will spend relatively long time waiting for input, and will therefore often be preempted. However, it is important that the process is waken up as fast as possibly, else the user will experience the system as unresponsive. In a typical UNIX-scheduler, this kind of process will be given large priority so that it can be resumed quickly [BC00].

In the AHEAD-project, the FPGA are meant to function as a hardware accelerator on a separate platform than the client. Because the communication cost between the tag and the client will be relatively large, it is probably not a good idea to place user-interactive processes on the FPGA. It is more appropriate that such processes are separated from the rest of the application and placed in software. In the case of a smartphone client, the process can run on the mobile OS.

However, there will be processes running on the FPGA that will have more need for communication than others. For example could the bottle-neck of a MPEG-decoder be running in hardware, but still need new input data quite

often.

Processor-bound processes

These kind of processes are typically spending much time performing heavy calculations that demands a lot of the processor. A process can for example be busy doing a cryptography algorithm that takes a few inputs, but where the actual algorithm takes a lot of time do to on the processor. Because these processes will spend a lot time on the CPU and not be preempted often by external events, the scheduler should penalize them so they do not starve other processes. In Linux, or any other UNIX-based kernel, such processes will be assigned a less favorable priority [BC00].

When building the scheduler for this thesis, it was assumed that all processes in the system are mostly processor-bound. It is assumed that the decision between which processes should run in software and which processes should be accelerated on the FPGA had already been taken by a partitioning module.

In any case, there will be some processes on the FPGA that will have less need for communication with the client or the other modules on the FPGA. Ideally, these processes should be given less priority so that they do not occupy the FPGA too much. A valid question to ask is how should the scheduler know how much the process is bound to I/O. This information can, if possible, be given by the module that partitions the application between hardware and software and should in fact be a major consideration when performing this partitioning.

Static priority

In Linux, static priority range from 1 to 99 and can be given to real-time processes by the users [BC00]. Real-time processes will always have higher priority than convential processes. This concept is interesting when considering scheduling in the AHEAD-project. For example can a pool of hardware modules be available for the AHEAD-tag, each assigned with a static priority relatively to the other modules. It would then be quite easy for the scheduler to assign priorites to the processes running on the FPGA.

3.7.5 Queue structure

The scheduler developed in this thesis makes heavy use of queue-structures for the processes. The queue structure is central in many schedulers, including the Linux scheduler [BC00]. A general queuing diagram for scheduling can be seen in picture 3.16 and was found in [Sta05, page 396].

As seen in the figure, the scheduler typically has an incoming queue for jobs and a READY-queue for jobs that is ready to be placed on the processor.

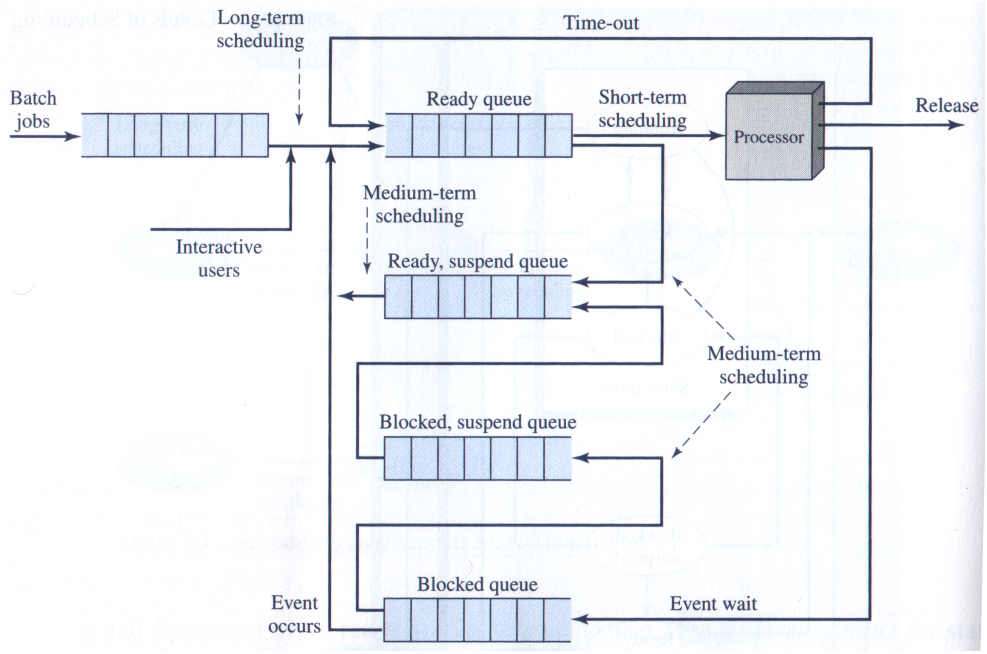

Figure 3.16: A generic overview of queues used in scheduling. Picture taken from [Sta05, page 396].

A process that has to wait for I/O is typically interrupted and placed in the BLOCKED-queue. When the I/O-resource is ready this process moves to the READY-queue as seen in the figure. The two suspend-queues are used when a process is swapped out from main memory to disk. The concept of swapping is complex and is not studied in this thesis.

3.7.6 Interrupter

Vegard Endresen has in his thesis worked on an interrupter built in VHDL and implemented on the FPGA [End09b]. The purpose of this module is to ask the running process to stop and then act as a buffer mechanism between the process and the CPU when the process sends it's state data to the CPU. The interrupter takes commands from the CPU, which in turn is interpreted as actions like shift in/out data to/from the module or start/stop the module.

The interrupter is a very critical part of the scheduler as much depends on if the state of the module can be extracted and reinserted fast enough. Equation 3.1 sums this up pretty good (taken from [Wol05]). In the worst case scenario for a partial reconfigurable system like in this thesis, t_{in} is the time it takes to place the process on the FPGA and insert the state data, while t_{out} is the time it takes to save the state data. It is assumed that the module's functionality is static and saved in a bitfile on disk and that it is never any need to read back the logic of a module from the FPGA.

The functional description of a process does not always need to be removed

from the FPGA. In a larger system, there may be several reconfigurable modules on the FPGA at the same time. For example could an adder and a multiplier both be present on the FPGA the whole time and be reused by several processes. In this case would t_{in} and t_{out} only be the time to stream data in and out of the module.

$$t_{accel} = t_{in} + t_x + t_{out} \tag{3.1}$$

Another way of saving and restoring state data is using the Internal Configuration Access Port (ICAP) (see section 3.2.5). This method has been briefly discussed as possible in Roman Plessl's master thesis in [Ple04, page 72]. The downside with this approach is that one would have to read back and save both the state data and the logic of the module. Most of the data would then describe the functionality of the module and not the state.

Round Robing scheduling policy and time slices

A time slice is defined as how long a process is allowed to run before it is preempted by the system. For a scheduling policy like Round-Robin (RR), each process is allowed to run for a certain time and is then preempted by a clock interrupt at periodic intervals [Sta05, page 405]. When the currently running process is interrupted, it is placed in the READY-queue.

3.7.7 Types of scheduling

The goal of a scheduler is to meet system objectives, such as response time, throughput and processor efficiency [Sta05, page 393]. The different types of scheduling can be summarized in table 3.7.7 and is taken from the book in [Sta05].

Type	Description
Long-term scheduling	The decision to add to the pool of processes to be executed.
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory.
Short-term scheduling	The decision as to which available process will be executed by the processor.
IO scheduling	The decision as to which process' pending I/O request shall be handled by an available I/O device.

Table 3.3: Types of scheduling (from [Sta05]).

3.8 Automated testing

3.8.1 Objective

The objective of this section is to introduce the reader to methodology for testing embedded systems. Since partial reconfigurable systems have conceptual similarities to software based operative systems, it seems natural to look at some of the most promising testing methodologies from software development. Test-driven development (TDD) has been gained a lot of popularity the recent years and may also be applicable for many embedded systems.

The theory in this section is used as background for the testing of the implemented systems in this thesis.

3.8.2 Definitions

Test-driven development (TDD)
Software development process.

Unit test
Smallest testable unit in a system. Could for example be a function of a module.

3.8.3 Motivation

The embedded software developed in this thesis runs on the PowerPC-processor on the Suzaku-development board. The code was first developed for the x86-architecure using Linux and the glibc-library and then recompiled to run on the PowerPC using the uClibc-library. As explained in section 3.2.7, porting a C-application from glibc to uClibc is often no harder than writing proper Makefiles and recompiling the software. This fact makes it interesting to look into more sophisticated methods to perform testing. Test-driven development, along with automatic testing, is an interesting approach to performing HW/SW-coverification of embedded systems.

Verification of HW/SW-systems is in general harder than for pure software systems. A typical methodology is to divide the development into phases. First the hardware is developed in one phase, then the software is built on top of the hardware in the next phase. This means the software-developers has to wait for the hardware-developers to finish or vica-versa. It also means that the interfaces and structure of the HW and SW has to be built independently, when they in fact depend heavily on each other. More time can then be spent redefining the interfaces or restructuring the system in one phase. If the software phase detects any bugs in the hardware, this can be very costly

to repair and the reiterations in the development cycle may become large. Variations of this methodology will of course exist, but the general problem is that time-to-market is crucial and it may be hard to build robust systems this way.

3.8.4 Test-driven development (TDD)

Test-driven development is a rather old technique (originated from the 80's), but has later gained more attention as development processes such as eXtreme Programming (XP), SCRUM and the Unified Process (UP) has become popular [CS10].

Figure 3.17 is from the article in [CS10] from January 2010. As discussed in that article, TDD-driven development is based on the red-bar/green-bar mantra, where the colors describe different steps in the development process.

The TDD-approach, as most other development approaches, starts with a list of requirements. One specific requirement is then converted to an executable test as shown in top of the figure. This could for example be a small unit test. To at least make the test compile successfully, some minimal system code is implemented, thereby moving down to the *red bar* as depicted in the illustration. The test is run and reports that the given system does not respond as expected. Now the developer knows two things:

- The test is working. It is reporting failure as expected.
- The system contains unimplemented features or features not working.

Now the developer starts working with the system to make the test pass successfully. When this is done, the process has moved to green bar (see figure 3.17). When making the test pass, the developer's focus could have been quite narrow and in the worst case some poorly written code was developed just to make the test pass. Some reasons for this are listed below.

- The developer did not see the tested unit as part of a complete system and did not consider the reuse potential.
- The development involved a lot of low-level or system-dependent considerations. The language used could for example involve a lot of implementation-specific details. For an embedded system, low-level optimizations can be a necessary part of the requirements.

To make the system better, the tested implementation is *refactored* several times as seen in the bottom of figure 3.17. Ideally, the refactoring should not introduce any new functionality. The smallest iteration makes the system

implementation more well structured, robust and reusable. The largest iteration is needed to improve the structure of the tests and improve the interface to the implemented system. For each change, the test is rerun to make sure everything is working as expected.

A large gain from TDD is that focus is shifted from implementation to the interface and observable behaviour of the system [CS10]. Also, focusing on verification early in the development process helps reveal misunderstandings in the specifications and poorly designed interfaces. The motivation is simply: Errors that are revealed later in the development process are more costly to repair. Errors that are revealed after production can be *very* expensive to repair, as the system may be mass-produced and/or in use by customers.

Another advantage is that TDD focus on automated testing. These tests can be run several times and the final results are also easily reproducable. For some systems, a small modification of one part of the system may cause another part to malfunction. The malfunctioning part may have been tested earlier, but without rerunning the test the error may not be revealed before much later. Well structured automatic tests, for example written using a test framework, may also function as documentation of the system's behaviour.

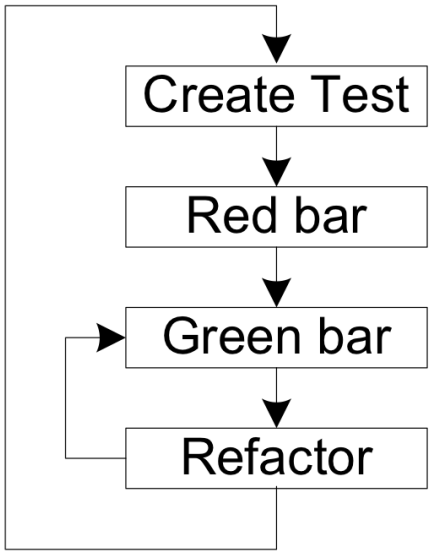

Figure 3.17: Test-driven development cycles. Picture taken from [CS10].

3.8.5 Check: A unit testing framework for C

Most of the implemented parts in this thesis was tested using the test-framework Check [che09]. Check is simple, written in ANSI-C and used by several open-source projects (see their website for more information). It is easy to compile the framework for both the x86- and the PowerPC-architecture, meaning many of the same tests can be run on both the development computer and the Suzaku board. For this thesis, the tests involving an FPGA could obviously not be

run on the development computer, but other than that all tests run on this system also.

Using a test framework should also make it easier for future developers to reuse the work. The tests are pretty easy to read and it is quite clear which parts of the system have been tested and which has not.

Chapter 4

Implementation: Partial reconfiguration of synchronous modules at run-time

4.1 Objective

The purpose was to create a methodology for making reconfigurable modules with an incoming clock signal. This is an important part of the AHEAD-project and a key for making modules interruptible in a HWOS. A module typically make use of several flip-flops for saving it's state and clocked shift-registers are used in Vegard Endresen's master thesis [End10] for shifting this data in and out.

Performing partial reconfiguration of asynchronous modules has been studied and performed at run-time in [Ham09]. However, reconfiguring synchronous modules has not been done earlier in the AHEAD-project and has also shown to be problematic in [End10].

4.2 Definitions

DIRT
: Directed Routing (see section 3.6.7). Used to constrain the clock routing for the reconfigurable framework.

EDK
: Embedded Development Kit from Xilinx (see section 3.1). Used to build complete FPGA configurations.

ISE
: ISE Design Suite from Xilinx (see section 3.1). Used to build partial FPGA modules.

Base design in VHDL from Atmark Techno
Used as basis for the FPGA on the Suzaku-V development board (presented in section 3.2.4). Built in EDK.

User logic base design in VHDL
Built by the developer on top of the base design from Atmark Techno. This is the framework and the interface for the reconfigurable modules. Must contain bus macros and is built in EDK.

Reconfigurable module in VHDL
The module to be swapped in and out of the FPGA. It is built in ISE and cropped by CLBRead afterwards. Must contain bus macros for signal interface.

BUFGCE
Global clock buffer configured with Clock Enable (`CE`) port (see section 3.6.4).

NFS
Network File Server (NFS). Described in chapter A.

4.3 Concept

In [Han10], reconfiguration of synchronous modules were done. The experiments and results from the project report are important and show that the existing framework supports reconfiguration of the routing information for a clock signal. The problem is that these experiments were not performed while the FPGA was running. The partial reconfiguration of the bitfile was done on the development computer and this file was afterwards verified by uploading it to an FPGA. Also, it was not done a thorough analysis on why partial reconfiguration of synchronous modules worked in this case.

The simple clocked flip-flop designs were found in [Han10]. A drawing of the base design and the reconfigurable modules are shown in figure 4.1. The setup contains one base design that is initially uploaded to the FPGA and four different reconfigurable modules. Each module had different placement of the clocked flip-flops.

The ultimate goal was to perform partial reconfiguration of these synchronous modules by writing them to ICAP at run-time.

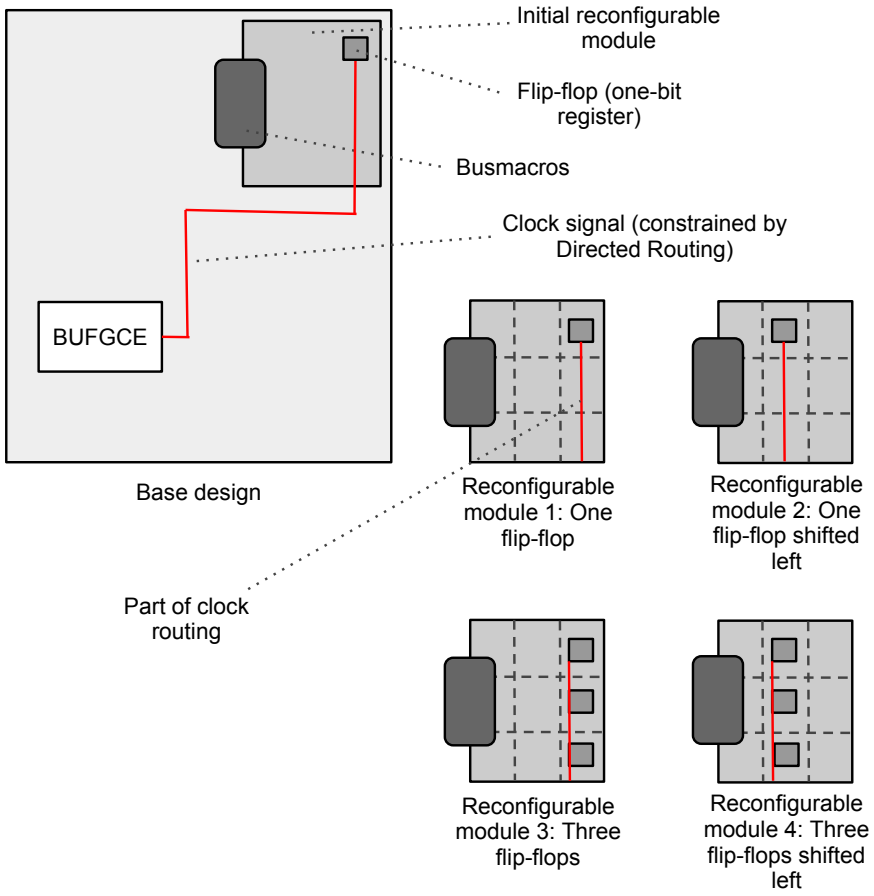

Figure 4.1: Base design and reconfigurable modules for testing reconfiguration of flip-flops.

4.4 Requirements and design

4.4.1 Setting up the base designs on the FPGA

The base VHDL-design for the Suzaku-V from Atmark-Techno was set up in EDK.

Several different user logic base designs were set up for the tests. These are briefly discussed here.

Base design 1: One flip-flop

This user logic base design is appended in listing C.1 and was found in [Han10]. A global clock buffer (BUFGCE) was added to enable/disable the clock signal to the reconfigurable module. The Clock Enable (`CE`) input port of this buffer is connected to one of the software accessible registers, thereby making it possible

to enable/disable the clock signal in the reconfigurable module from software. As described in section 3.4.4, bus macros must be included in the base design to make it compatible with reconfigurable modules.

This base design was built with the initial module shown in listing C.3 found in [Han10]. The values of the flip-flops can be read by using the kernel module and the small C-program found in [Han10]. How this design is placed and routed on the FPGA is shown in figure 4.2.

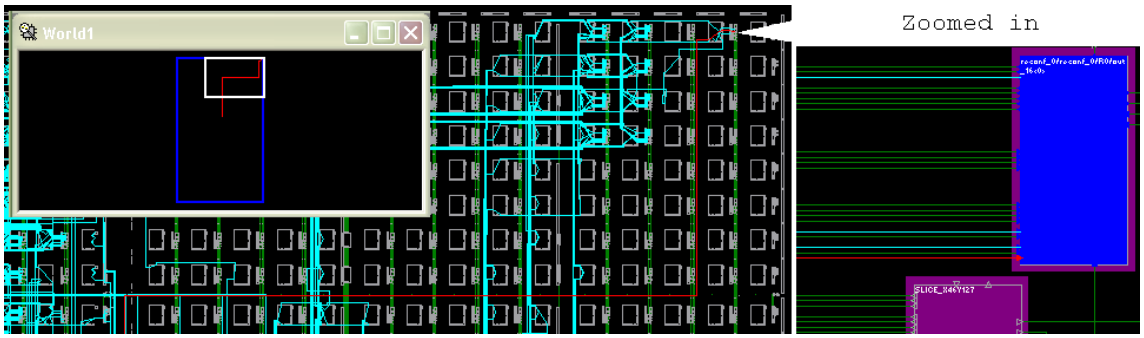

Figure 4.2: Base design with one flip-flop (from FPGA-editor).

Base design 2: Three flip-flops

This design is the same as base design 1, but the initial reconfigurable module has three flip-flops as shown in listing C.4 (from [Han10]). A picture of the design is shown in figure 4.3.

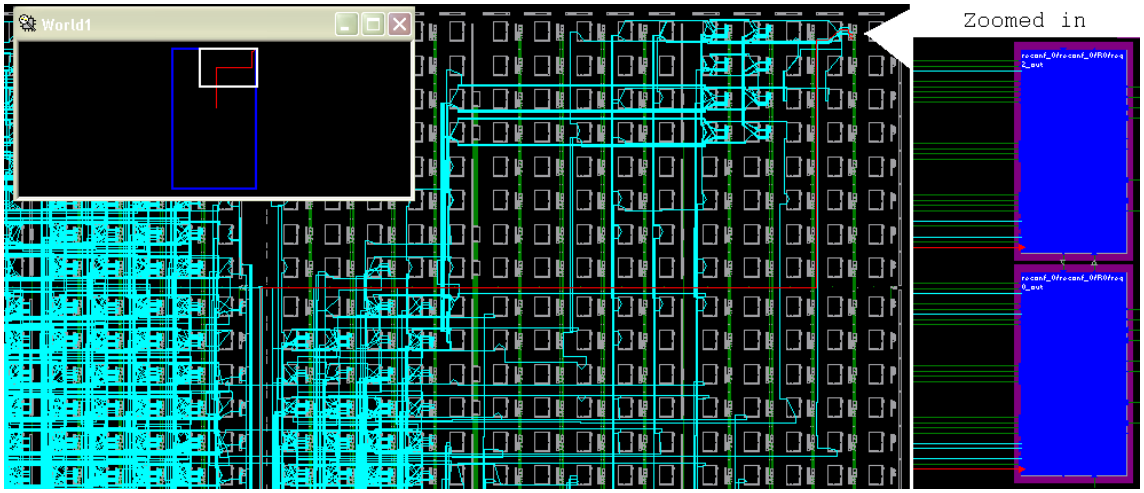

Figure 4.3: Base design with one flip-flop (from FPGA-editor).

Base design 3: One flip-flop with DIRT constraint on clock

This design is built using base design 1 as basis, but the routed clock signal was constrained by a DIRT constraint. How this is done is discussed in chapter 3.6.

Base design 4: Vegard Endresen's instruction- and data cache back-end

As described in [End10] and in section 3.7.6, this is the most promising design from this thesis for handling state loading/saving. The design is built with a sequential multiplicator as the initial reconfigurable module. As all the other base designs, bus macros are used for communicating with the reconfigurable module.

4.4.2 Building reconfigurable modules in ISE

All the reconfigurable modules shown in figure 4.1 were built without the base design. This was done in Xilinx ISE. These modules and how they were built are described briefly here.

Reconfigurable module 1: One flip-flop

The simple code for this module is shown in listing C.3. By generating a User Constraints file (see section 3.6.6 in Planahead, the module is forced to be placed at the upper left corner of the FPGA. This is important to make sure the module is placed in the reconfigurable area when writing it to the base configuration on the FPGA. The module will be look like the module in figure 4.2 without the static design.

Reconfigurable module 2: One flip-flop shifted left

This is the same module as the first module, except that a constraint in the UCF-file is used to move the flip-flop left.

Reconfigurable module 3: Three flip-flops

The code for this module is shown in listing C.4. The same type of constraints is used as for the first module.

Reconfigurable module 4: Three flip-flops shifted left

This is the same module as the third module, except that a constraint in the UCF-file is used to move the flip-flops left.

Reconfigurable module 5: A multiplicator

The multiplicator from base design 4 was also built as it's own module in ISE.

4.5 Development of methodology through test suites

The methodology for self-reconfiguration of synchronous modules were done by performing a series of test suites. These test suites, their objectives, results and summaries are described in chapter 6.

4.5.1 Analysis of first test suite: Simple flip-flop designs

The first test suite is described in section 6.2. The results from this suite verified that the experiments from [Han10] could be performed at run-time by writing the reconfigurable module to ICAP, but does not explain why Vegard Endresen could not perform run-time reconfiguration in [End10].

Vegard Endresen only used EDK for building the base design, while the reconfigurable modules were built without the base design in ISE. To find if there were any difference when building the modules alone in ISE compared to building them together with the static design in EDK, Vegards system where rebuilt.

Figure 4.4 shows a close-up screenshot of the clock routing in Vegards static backend module. The system was built with a multiplicator module as the initial reconfigurable module and a global clock buffer for controlling the clock signal into the module. As seen in this picture, the clock signal is routed along the second clock wire.

For the simple flip-flop design (base design 1), the picture in 4.5 shows that the clock signal is routed incorrectly along the second clock wire. All the reconfigurable modules built in ISE had the clock routed along the first clock wire.

The mismatch of the clock wires is seen as a likely reason as to why his experiments and the experiments with the simple flip-flop designs did not succeed.

Different ways of solving this problem was investigated. One potential solu-

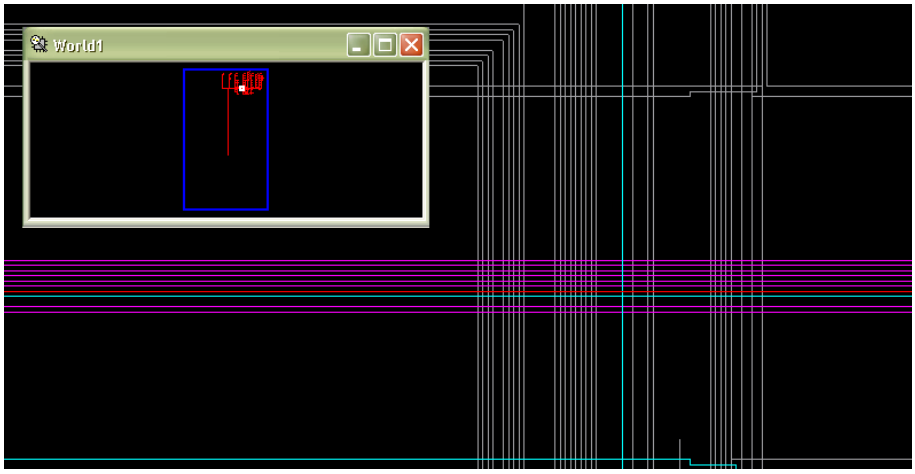

Figure 4.4: Screenshot of clock routing for Vegard's backend module (from FPGA-editor).

tion was to use a bus macro solution for the clock signal. Bus macros has been used for the data signals into the module (see section 3.4.4). Sverre Hamres project report [Ham08a] and the attached tutorial has shown how a bus macro can be built. However, because the high amount of work for creating one such bus macro and because it was not sure whether or not this solution would work for a clock signal, this idea was discarded.

A more promising solution was the use of Directed Routing. Section 3.6.3 in the theory discusses how this concept works and how it can be applied to a complete static design. This was done in the third test case 6.2.3. This test case is a proof-of-concept showing that a static interface for the clock routing can be defined, much in the same way as bus macros define a static interface for data and control signals.

4.5.2 Analysis of second test suite: Instruction- and data cache backend

The test cases in the second test suite (see section 6.3) was performed to check if the clocking methodology could be used for the backend module written by Vegard Endresen. As can be seen from the results, both tests failed. It was possible to build the backend and constrain the clock routing if the backend was built with a simple flip-flop as it's initial reconfigurable module.

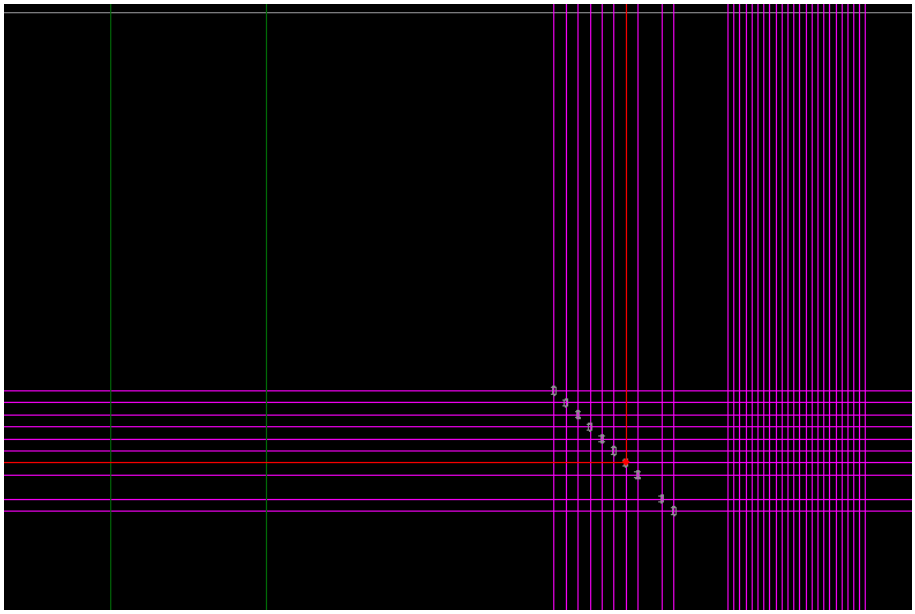

Figure 4.5: Screenshot of clock routing for base design 1 (from FPGA-editor).

Chapter 5

Implementation: Scheduler and HWOS

5.1 Objective

The goal was to develop and implement a simple scheduler for FPGA-based tasks. This scheduler should be able to schedule processes in time. A good structure should be realized to make room for future optimizations on the critical parts of the scheduler.

5.2 Structure of code and compiling

The HWOS can be compiled for the standard Linux x86-architecture or the embedded uClinux using the PowerPC microprocessor. A description of the directory structure for the HWOS and how to compile the code is found in B.

5.3 Documentation of HWOS in Doxygen

All the C-code for the HWOS was documented in the documentation system Doxygen. A HTML-version of this documentation is included in the appended ZIP-file. The HTML-document is probably the best place to start when trying to understand the source code.

The documentation can be found in the directory `hwos_sw/docs`.

The code is also in the appendix (see section D) for convenience.

5.4 Work done by Vegard Endresen

The software part of the HWOS was started from the implementation made by Vegard Endresen in [End10]. However, as stated by Vegard Endresen in section 2.1.2, this implementation was rather rough and done without a thoroughly preanalysis. Most of the implementation has been replaced or completely rewritten. The parts that are based on Vegard's implementation is hdev (not used in this thesis) and `hmqueue` (queue of System V message queues). Also the concept of having a message server and a daemon has been taken from him.

It should be clearly defined what is original work from this thesis in the .h-files or in the Doxygen-documentation.

5.5 General structure of the HWOS

The HWOS has been implemented as a daemon running as a background process in uClinux on the Suzaku. As shown in figure 5.1, the daemon was implemented as four threads: The scheduler thread, the placer thread, the message server thread and the timer thread.

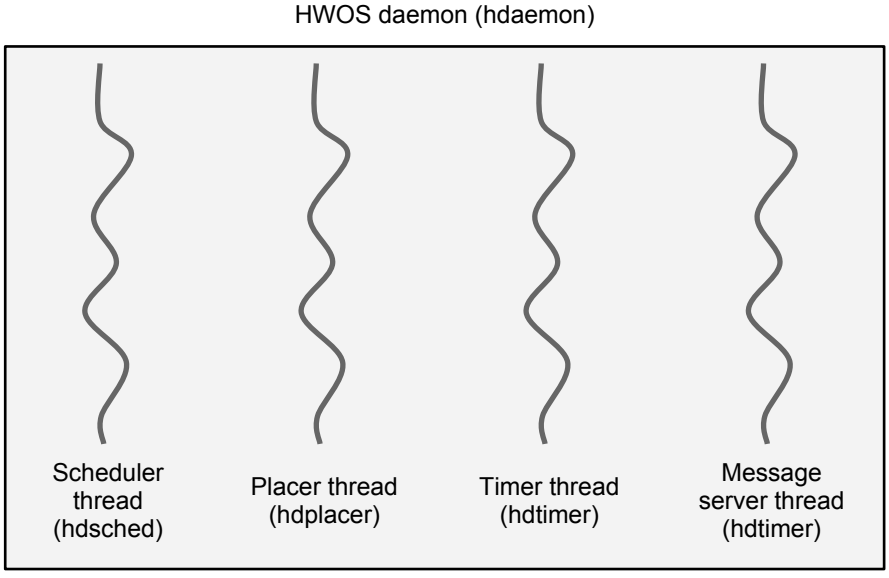

Figure 5.1: General structure of the HWOS-daemon.

5.6 General structure of the message server

The message server wait for new events from clients. The module makes heavy use of the `hmsg` message interface module in the HWOS-library. When a client

registers a new process, this is signalled to the scheduler thread.

5.7 General structure of the placer

The placer thread works really simple. It waits for a timer interrupt from the timer thread. When this is received, the placer replaces the currently running process on the FPGA and puts it in the queue for rescheduling. This is signalled to the scheduler thread.

5.8 General structure of the timer

The timer simply waits for one predefined time slice. Each time a time slice has been reached, the timer interrupts the placer.

5.9 General structure of the scheduler

The concept of processes in priority queues (as discussed in 3.7.5) was implemented. This means that each process of the same priority and state belongs to the same queue. There is for example a list of queues in the READY-state. Each queue in this list has processes with the same priority and the scheduler will always choose the process with the highest priority (see figure 5.2 and implementation below).

The scheduler use the module `hevent` for handling events from the message server and for sending messages to the `hplacer` module. The scheduler performs long-term scheduling, short-term scheduling or rescheduling of a process depending on what message it receives. This are listed below:

Long-term scheduling The message server has added a new process to the queue of new processes. The new process is fetched by the scheduler. If the process is admitted, it will be added to the correct READY-queue or a new READY-queue will be created for the process.

Short-term scheduling A new process has been added to the READY-queue. The scheduler gets the process and makes it ready for the placer module for placement on the FPGA.

Rescheduling The time slice for the running process has elapsed. The process is added to the queue for rescheduled processes and is fetched by the scheduler for rescheduling.

5.10 List of scheduler queues: hsqlist

The structure for `hsqlist` is shown in listing 5.1. Each instance of this structure keeps a list of scheduler queues. Each queue in this list is supposed to keep processes of the same state. Because of this, the system only contains a finite number of queues kept in an array as shown in listing 5.2.

Listing 5.1: Structure for list of scheduler queues (from libhwos/hsqlist.c)

```
1  /*! A simple structure for a list of queues.
2  *  One instance of this structure points to one list of
       queues.
3  *
4  * There are only as many instances of this structure as
       there are
5  * different process states (defined by the constant
       HPS_NUMBER).
6  */
7  struct hsqlist {
8          //! Number of queues in this list.
9          int queues_num;
10         //! First queue element in this list.
11         struct hsqueue* first_queue;
12         //! Last queue element in this list.
13         struct hsqueue* last_queue;
14 };
```

Listing 5.2: Array of queue lists (from libhwos/hsqlist.c)

```
1  /*! Array of queue lists.
2  *  There is one list for each possible state in the
       scheduler.
3  *  Warning: This will result in a compiler error if the
       number of NULL's on the right
4  *  side of the expression does not match HPS_NUMBER.
5  */
6  static struct hsqlist* queue_lists[HPS_NUMBER] = {NULL, NULL
       , NULL, NULL, NULL};
```

An conceptual drawing of the structure of an hsqlist is shown in figure 5.2. An UML-class diagram of the hsqlist, hsqueue and hprocess is shown in figure 5.3.

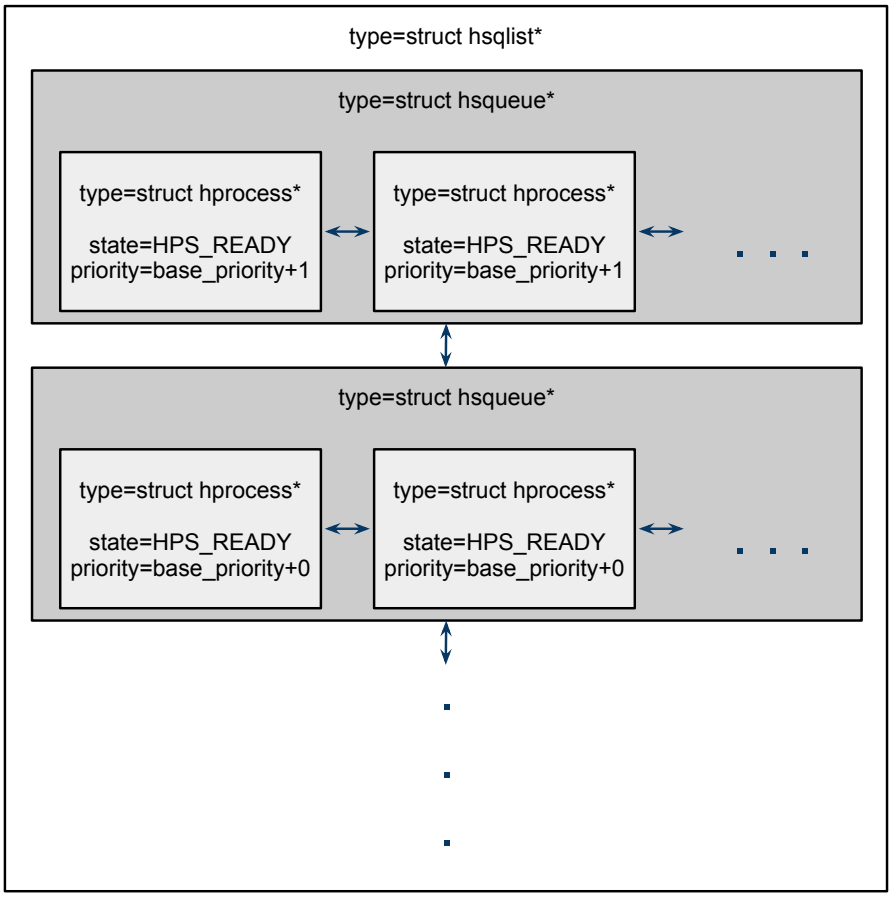

Figure 5.2: Conceptual drawing of hsqlist, hsqueue and hprocess.

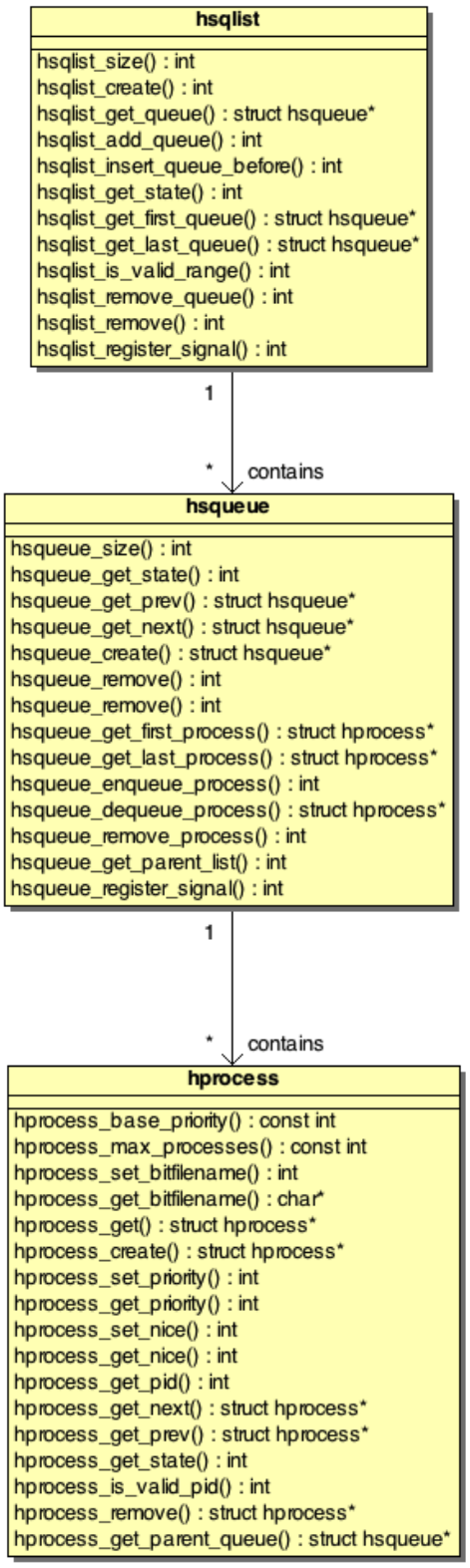

Figure 5.3: UML 2 Class Diagram of hsqlist, hsqueue and hprocess. Arguments to functions are omitted.

5.11 Process structure: hprocess

A structure for a process was defined as in listing 5.3. This struct is not visible to the rest of the code, but the interface to the module is shown in D.19.

Listing 5.3: Structure for process (from libhwos/hprocess.c)

```
1  struct hprocess {
2          //! The process id.
3          int pid;
4          //! The state of the process.
5          int state;
6          //! Value to manipulate priority (set by user
               application).
7          int nice;
8          //! Actual priority for process.
9          int priority;
10         //! File where the FPGA-bitstream for the process is
               .
11         char* bitfilename;
12         //! File where the state data for the process is.
13         char* statefilename;
14         struct hprocess* next;
15         struct hprocess* prev;
16         struct hsqueue* parent_queue;
17 };
```

This structure holds the basic information about the process such as ID, state, priority and filenames for the bitfile and the state data. The fields `next`, `prev` and `parent_queue` are for keeping the process in a double linked list (as explained in section 5.12).

5.12 Scheduler queue structure: hsqueue

As described in section 3.7.5, each process belongs to a queue. Each `hsqueue` contains a queue of processes of the same state and priority.

Listing 5.4: Structure for process queue (libhwos/hsqueue.c)

```
1  //! A double linked list structure for a process queue.
2  //! One instance of this structure points to one queue of
       processes.
3  //!
4  //! Several instances of this structure forms a list of
       queues.
5  //! All queues in the same list will have processes with the
        same state.
6  struct hsqueue {
7          //! Number of processes in the queue.
8          int processes_num;
9          //! Points to the parent list, if any.
10         int parent_list;
11         //! First element in the queue.
12         struct hprocess* first_process;
13         //! Last element in the queue.
14         struct hprocess* last_process;
15         //! Points to the next queue in this list.
16         struct hsqueue* next;
17         //! Points to the previous queue in this list.
```

```
18          struct hsqueue* prev;
19  };
```

5.13 Rewritten version of icap_write: hicap

The implementation of *icap_write* by Sverre Hamre (see section 3.4.3) was rewritten to a library module in the HWOS library. The interface was changed and documented in Doxygen. Whereas the previous version of *icap_write* had the adressing of frames hard coded in the program, the new implementation does this conversion automatically.

The newest `hicap` was not tested thoroughly. This should be done before using it in real systems. The header file defining the interface for *hicap* can be found in D.13.

Chapter 6

Verification and results: Partial reconfiguration of synchronous modules at run-time

The verification of the clocking methodology is divided into test suites. Each suite contains a set of test cases for a particular system.

Many of the tests were done manually: Some system were set up and testing was by manually running a program and providing input on the command line. For the most important test suites, the tests were written as automated tests in the Check framework 3.8.5.

6.1 Definitions

The definitions for this chapter is the same as for chapter 4.

6.2 First test suite: Simple flip-flop designs

6.2.1 Test case 1: Cut the reconfigurable modules from the base designs

Setup

In this test case, base design 1 was initially built and placed on the FPGA. Base design 2 was also built and bitfiles was generated for both designs. Building the modules this way takes a lot of time (over 30 minutes for each complete bifile). CLBRead from section 3.4.2 was used to cut out the modules from both designs. Listing 6.1 shows how this was done for both the base designs. The same syntax was used for both bitfiles.

The two resulting partial bitfiles corresponds to reconfigurable module 1 and 3 in figure 4.1. These bitfiles were uploaded to the FPGA using NFS.

```
$ ./CLBRead -i onedff_base.bit -o onedff_part.bit -fmR -sc
    21 -ec 23 -verbose
Input file name:        onedff_base.bit
Output file name:       onedff_part.bit
Frame mode, reading out frames
read_frame: acessing
read_frame: Memory alocated
locateFrameStart: Synch word = aa995566
locateFrameStart: FDRI word = 30004000
locateFrameStart: word = 50024090
locateFrameStart: Type2 header detected
locateFrameStart: Word count: 147600
locateFrameStart: Number of configuration bits: 4723200
locateFrameStart: Configuration bytes = 4801
locateFrameFromCLBnr: frame: 644
locateFrameFromCLBnr: CLBcol: 21
locateFrameFromCLBnr: frame: 1190
read_frame: frames to read out: 66
read_frame: frameStart: 199961
read_frame: Read out CLBs startCLB: 21 endCLB: 23
read_frame: startCLB >= 16 startCLB < 24
warning BRAM/DSP connections read out not implemented
read_frame: Closing files
```

Listing 6.1: Using CLBRead to cut out partial reconfigurable modules from base design 1 and 2.

The program *icap_write* (as described in section 3.4.3) was modified (around line 237) as in listing 6.2.

```
// Address hardcoded in, row 2(=1) from center, CLB 21(=25),
    frame 0.
      if(write_header(handlemem, 1, 25, 0, frames) < 0){
```

Listing 6.2: Modification of icap_write for flip-flops tests

Because the old version of *icap_write* had hardcoded the addressing of frames, this program had to be recompiled after the change. The new version of the *icap_write*, called *hicap*, does include an easier interface for adressing (see section 5.13).

A kernel driver and a software program (from [Han10]) were used to write/read to the flip-flops on the FPGA. This program could also control the clock enable port (`CE`) of the global clock buffer.

Objective

A setup similar to the one in this test case was set up in [Han10]. In that project, a partial bitfile was written to another bitfile containing the complete FPGA-configuration, but the experiment was not done while the FPGA was running. The resulting bitfile was later verified by uploading it to the FPGA.

In this test case, the partial bitfiles is written to a running FPGA using `icap_write`. The running configuration is verified before and after the writing.

The objective is to verify that the experiments from [Han10] can be performed at run-time also.

Testing and results

The partial bitfiles is in this case written to the running configuration by issuing the command on the form depicted in 6.3.

```
1 $ ./icap_write -i partial_bitfile.bit -f 22
```

Listing 6.3: Usage of icap_write for DFF-tests.

The running configuration were tested by setting each flip-flops input value HIGH one at a time. The same tests were performed when the clock enable (`CE`) signal to the clock buffer was turned off.

All flip-flops did respond to the input values when `CE` was high. No change happened when it was low.

Summary

The tests showed that clocked, reconfigurable modules can be extracted from one bitfile and inserted into another at run-time. This verifies the results from [Han10]. Note that this test suite did not test if the flip-flops and the clock routing could be moved left when performing reconfiguration. It should also be noted that building the reconfigurable module together with the base design takes a lot of time.

6.2.2 Test case 2: Build the base design and the reconfigurable module separately

Setup

In this test case, base design 1 was initially placed on the FPGA. Reconfigurable module 1 and 3 were built separately in ISE as described in section 4.4.2.

For cutting out the modules from the ISE-designs, CLBRead were used as in listing 6.1. This produced partial bitfiles for each module.

icap_write was modified as in listing 6.2 and used as in listing 6.3.

A kernel driver and software program (from [Han10]) was used for writing/reading to the flip-flops.

Objective

The objective was to verify that modules built separately in ISE may not be compatible with a static design built in EDK.

Testing and results

The partial bitfile was written to the configuration memory using *icap_write*. Several test values was written to the registers, but no response were seen.

Summary

The test case showed that building the base design in EDK and the reconfigurable modules in ISE made the reconfigurable modules not compatible with the base design. After writing the partial bitfile through ICAP, the resulting configuration on the FPGA did not work as expected and the flip-flops were not updated when the input signals were changed.

6.2.3 Test case 3: Build the base design with DIRT

Setup

In this test case, base design 3 was initially placed on the FPGA. Reconfigurable module 1, 2, 3 and 4 were built separately in ISE as described in section 4.4.2.

For cutting out the modules from the ISE-designs, CLBRead was used as in listing 6.1. This produced partial bitfiles for each module.

Because these tests are so important, it was written automated tests for them in the Check framework. These tests use the HWOS-library and the new *hicap* module for writing each partial bitfile to ICAP.

The directory `dff_tests` in the appendix contains the following subdirectories:

bitfiles
: Pregenerated bitfiles for the base design and the reconfigurable modules.

check-powerpc
: Precompiled version of the Check library for the PowerPC-processor on the Suzaku board.

drivers
: Precompiled kernel drivers for ICAP-access and access to the HW/SW-accessible registers.

swreg_driver
: Source code for the HW/SW-access-driver. This driver is used to read-/write to the flip-flops on the FPGA.

tests
: Source code for actual tests.

The test program performs several tests on all the reconfigurable modules. The input of all the flip-flops are automatically toggled and the response is registered both when the Clock Enable signal (`CE`) is HIGH and when it is LOW.

The tests are compiled the same way as the HWOS as described in section B. Some testing was also done manually to make sure the automated tests worked correctly.

Objective

The objective was to show that partial run-time reconfiguration of clocked modules could be performed if the method using DIRT constraint (see section 3.6.3) was used.

Testing and results

The output from the tests are shown in listing 6.4. The tests were conducted on two different Suzaku development boards.

Listing 6.4: Automatic tests for DFFs with DIRT

```
1  Self-reconfiguration of synchronous modules on FPGA
2  Prerequisites:
3  Development board: Suzaku-V.
4  Processor: Power-PC.
5  FPGA: Virtex-4 XC4VFX12.
6  OS: atmark-dist-20090318 (linux-2.6.18-at11).
7  Base FPGA-design: bitfiles/dff_with_bufgce_with_dirt.bit (
       must be uploaded with netflash).
8
9  These tests will perform partial reconfiguration of
       synchronous modules on the FPGA.
10 For each time the FPGA is reconfigured, the new design will
       be thoroughly tested.
11 ===
12 Inserting module: One flip-flop.
13 ===
14 Running suite(s): onedff
15 100%: Checks: 3, Failures: 0, Errors: 0
16 test_onedff.c:45:P:Core:test_create:0: Passed
17 test_onedff.c:91:P:Core:test_clock_enabled:0: Passed
18 test_onedff.c:143:P:Core:test_clock_disabled:0: Passed
19 ===
20 Inserting module: Three flip-flops.
21 ===
22 Running suite(s): threedffs
23 100%: Checks: 3, Failures: 0, Errors: 0
24 test_threedffs.c:44:P:Core:test_create:0: Passed
```

```
test_threedffs.c:120:P:Core:test_clock_enabled:0: Passed
test_threedffs.c:191:P:Core:test_clock_disabled:0: Passed
===
Inserting module: One flip-flop (shifted one column left).
===
Running suite(s): onedff
100%: Checks: 3, Failures: 0, Errors: 0
test_onedff.c:45:P:Core:test_create:0: Passed
test_onedff.c:91:P:Core:test_clock_enabled:0: Passed
test_onedff.c:143:P:Core:test_clock_disabled:0: Passed
===
Inserting module: Three flip-flops (shifted one column left)
    .
===
Running suite(s): threedffs
100%: Checks: 3, Failures: 0, Errors: 0
test_threedffs.c:44:P:Core:test_create:0: Passed
test_threedffs.c:120:P:Core:test_clock_enabled:0: Passed
test_threedffs.c:191:P:Core:test_clock_disabled:0: Passed

Summary:
All tests passed!
```

One significant problem was that it took several seconds between each new partial reconfiguration.

Summary

All tests worked as expected. These tests showed that a static interface could be defined for routing of clock signals. This is done using Directed Routing and makes it possible to perform partial self-reconfiguration of synchronous modules on the FPGA.

One problem was large reconfiguration-time.

6.3 Second test suite: Instruction- and data cache backend

6.3.1 Test case 1: Make the backend compatible with synchronous modules

Setup

In this test, base design 4 (backend) was built in EDK. Reconfigurable module 5 (the multiplicator) was built in ISE.

After this was done, the DIRT constraint for the ISE-design was extracted similar to the last test case in test suite 1. Because all flip-flops must be defined for the UCF, this constraint was really long. It was also seen that the relative paths to the components were different in the base design and in the reconfigurable module. This had to be changed before the constraint were put

in the base design.

The last thing that was done in the setup was placing the DIRT-constraint in the UCF-file of the base design. This is the same methodology as for the last successfull test case in test suite 1.

Objective

The objective of this test was to show that constraints could be put on the clock signal for the backend module created by Vegard Endresen in [End10]. The backend was built with the same module as in his thesis.

Testing and results

This test could not be performed on the FPGA, because the Directed Routing Constraint were reported by the Xilinx tools to not beeing followed during the build process. The output from Xilinx EDK is shown in listing 6.5.

Listing 6.5: Output from EDK after adding DIRT to Vegards backend with multiplier

```
1 INFO:Route - One or more directed routing (DIRT) constraints
      generated for a specific device have been found. Note
    that
2    DIRT strings are guaranteed to work only on the same
       device they were created for. If the DIRT constraints
       fail,
3       verify that the same connectivity is available in the
          target device for this implementation.
4
5       # of EXACT MODE DIRECTED ROUTING found:1, SUCCESS:0,
          FAILED:1
```

This was the only information provided on why it failed.

Summary

The test did not succeed. Xilinx EDK refused to follow the extracted DIRT constraint.

6.3.2 Test case 2: Make the backend compatible with synchronous modules using dummy module

Setup

In this test, base design 4 (backend) was modified so it used reconfigurable module 1 (one flip-flop) as the initial reconfigurable module. The complete design was built in EDK. Reconfigurable module 1 (one flip-flop) was modified to make use of the same bus macros as base design 4 (backend).

Similar to the previous test case, the DIRT constraint for the ISE-design was extracted and placed into the UCF-file of the base design. Similar to the test cases in test suite 1, the extracted DIRT was really short and only specifed the global clock buffer, the flip-flop and the routing between them.

Objective

The objective of this test was to show that constraints could be put on the clock signal for the backend module created by Vegard Endresen in [End10]. By building his backend with a very simple flip-flop-design as initial reconfigurable module, Xilinx EDK would hopefully follow the constraint.

Testing and results

When building this design, Xilinx EDK reported the constraint to be followed successfully. The output from Xilinx EDK is shown in listing 6.6.

Listing 6.6: Output from EDK after adding DIRT to Vegards backend with a dummy module

```
1 INFO:Route - One or more directed routing (DIRT) constraints
      generated for a specific device have been found. Note
    that
2    DIRT strings are guaranteed to work only on the same
       device they were created for. If the DIRT constraints
       fail,
3       verify that the same connectivity is available in the
          target device for this implementation.
4
5       # of EXACT MODE DIRECTED ROUTING found:1, SUCCESS:1,
          FAILED:0
```

However, after uploading the base design to the FPGA, the Suzaku would not boot anymore. This happened despite to the fact that the design was built with CRC-checking.

Summary

The test did not succeed. Xilinx EDK did follow the DIRT constraint, but the Suzaku deadlocked when rebooting the new base design.

Chapter 7

Verification and results: Scheduler

The verification of the HWOS and the scheduler is divided into test suites. Each suite contains a set of test cases for a particular module.

All tests were written as automated tests in the Check framework (see section 3.8.5). This should make it easy for a developer to see in detail what has been tested and quickly reproduce the test results. The actual source files for the tests are pretty large and are therefore only included in the appended ZIP-file.

7.1 Portability

The tests and the Check framework are written in ANSI-C. They can be compiled for both the x86-architecture on the development computer and the PowerPC-architecture on the Suzaku. How to build the tests for each architecture is described in section B.

7.2 Test strategy

The actual tests are written in C and is divided into *test suites*, *test cases* and *unit tests*. Each *test suite* tests a particular module in the system. Each *test case* tests a typical scenario that the module should handle through it's interface. For each test case, there is a number of *unit tests*. Some initial conditions and setup must be done for each test case, but the test cases was meant to run independently of each other and that each unit test should run independently of other unit tests. The Check framework helps to structure the tests this way.

7.3 Description of test suites

hprocess
Tests that the process module can be created, removed and that fields in the process image can be set and read.

hsqueue
Tests basic functionality for a scheduler queue. Tests that a large amount of processes can be enqueued and dequeued.

hsqlist
Tests basic functionality for a list of scheduler queues. Tests that a large amount of scheduler queues can be inserted and removed from the list.

hmqueue
Tests basic functionality for a queue of System V message queues. Tests that message queues can be added, removed, enqueued and dequeued.

hdev
Tests basic functionality for a list of kernel devices. Tests that a devices can be added, removed, enqueued and dequeued.

hlist
Tests basic functionality for a general list structure. Test that list elements can be added, removed, enqueued and deququed.

hvmem
Tests allocation and deallocation of virtual memory. Note that this functionality was just started and is not done.

hwos_daemon
Tests basic message passing between a client and the HWOS message server. Also tests that a process can be registered and scheduled by a client.

7.4 Test results

7.4.1 The HWOS-daemon

It was verified that the message server worked as expected. It was also verified that the scheduler could receive new processes and reschedule processes in a round-robin manner. The placer worked as expected and `printf` statements showed that it potentially could place FPGA-tasks. However, because the instruction- and data cache backend was not working, it was not performed placement of real FPGA-tasks.

7.4.2 The HWOS-library

All the tests were run successfully both on the development computer and the Suzaku board. A output trace from the test program is shown in listing 7.1.

```
1=yes, 0=no
Perform test on daemon (daemon must be started)?
1
===
Running suite(s): hprocess
100%: Checks: 1, Failures: 0, Errors: 0
test_hprocess.c:55:P:Core:test_process_create:0: Passed
===
Running suite(s): hsqueue
100%: Checks: 3, Failures: 0, Errors: 0
test_hsqueue.c:83:P:Core:test_queue_create:0: Passed
test_hsqueue.c:110:P:Core:test_enqueue:0: Passed
test_hsqueue.c:148:P:Queue:test_dequeue:0: Passed
===
Running suite(s): hsqlist
100%: Checks: 4, Failures: 0, Errors: 0
test_hsqlist.c:78:P:Core:test_list_create:0: Passed
test_hsqlist.c:105:P:Core:test_add_queues:0: Passed
test_hsqlist.c:136:P:Core:test_insert_queues_before:0:
    Passed
test_hsqlist.c:175:P:Remove:test_remove_queues:0: Passed
===
Running suite(s): hmqueue
100%: Checks: 3, Failures: 0, Errors: 0
test_hmqueue.c:54:P:Core:test_queue_create:0: Passed
test_hmqueue.c:81:P:Enqueue:test_enqueue:0: Passed
test_hmqueue.c:119:P:Dequeue:test_dequeue:0: Passed
===
Running suite(s): hdev
100%: Checks: 3, Failures: 0, Errors: 0
test_hdev.c:58:P:Core:test_queue_create:0: Passed
test_hdev.c:89:P:Enqueue:test_enqueue:0: Passed
test_hdev.c:127:P:Dequeue:test_dequeue:0: Passed
===
Running suite(s): hlist
100%: Checks: 4, Failures: 0, Errors: 0
test_hlist.c:77:P:Core:test_queue_create:0: Passed
test_hlist.c:106:P:Core:test_enqueue:0: Passed
test_hlist.c:169:P:Core:test_insert_inbetween:0: Passed
test_hlist.c:203:P:Queue:test_dequeue:0: Passed
===
Running suite(s): hvmem
100%: Checks: 1, Failures: 0, Errors: 0
test_hvmem.c:56:P:Core:test_hvmem_allocate:0: Passed
===
Running suite(s): hwos_daemon
100%: Checks: 4, Failures: 0, Errors: 0
test_hwos_daemon.c:70:P:Core:test_setup:0: Passed
test_hwos_daemon.c:70:P:Register message queue:test_setup:0:
     Passed
test_hwos_daemon.c:78:P:Register message queue:test_rmqueue
    :0: Passed
test_hwos_daemon.c:94:P:Process:test_register_process:0:
    Passed

Summary:
All tests passed!
```

Listing 7.1: Output trace from testing of the HWOS-library.

Chapter 8

Discussion

8.1 Partial self-reconfiguration of synchronous modules

The results from this thesis has shown that partial self-reconfiguration of clocked reconfigurable modules can be performed on a running FPGA. This was the most important goal of the thesis. All the tests with the simple, clocked flip-flop designs were working when constraining the clock signal with Directed Routing. This solution should in theory be compatible with any reconfigurable system using bus macros. This is because the clock signal is the only signal not piped through these macros and because the bus macros make it possible to build the static system with any kind of reconfigurable module. In theory, any such reconfigurable system can be built with a simple flip-flop and a corresponding global, clock buffer.

One significant problem did occur: The reconfiguration time for each partial design was several seconds long. This is occured both for the old and the new version of *icap_write*. This is in large contrast to the results by Vegard Endresen in his project report. These results show that partial reconfiguration of quite large, asynchronous modules can be performed in milliseconds. It is possible that the reconfiguration time get larger when using a clock signal, but several seconds seems to be quite high. The problem could be due to some hardware failure, but this is rather unlikely as the tests were conducted on two different Suzaku development boards.

The results shows some challenges when integrating the clocking methodology in the instruction- and data cache backend by Vegard Endresen [End10]. The backend can be built successfully with a constrained clock signal using a dummy module, but the Suzaku fails to boot after uploading the complete design to the FPGA. This is quite strange, as the dummy module and the backend both worked fine when not building them together. Since the reconfigurable dummy module is built together with the backend using bus macros, the backend should presumably be built the same way as when building it

together with the multiplier. There is no reason that signals would be optimized away by the tools, because the bus macros function as a black box in the design. Working with this system was quite challenging as a lot of time was spent on understanding the system made by Vegard Endresen. One problem was that it could not be found testbenches for each part of the system and that the documentation on the actual implementation was a bit sparse. There is also some uncertainty whether it works good with any reconfigurable module. It also takes a lot of time to perform these tests. Each time a small change is done in the base design, the complete system has to be built in EDK (a process that takes over 30 minutes). Each time the Suzaku deadlocks, it have to be reflashed with an inital configuration through JTAG using a dedicated development computer.

The results from [Han10] have been the largest motivation for finding a way to perform reconfiguration of synchronous modules at run-time. The general motivation for using synchronous modules in the framework has been disussed in section 3.5. This shows that synchronous design is the most reasonable design technique for an FPGA, at least if no special framework for asynchronous development is used.

When searching for available litterature on reconfiguration of synchronous modules, it could not be found any articles with a specific solution to the problem. One reason could be that many of them are using the partial reconfiguration flow from Xilinx and that this makes the reconfiguration of clock signals transparent to the user. For example does the article in [JIA06] ("The GAPLA: a globally asynchronous locally synchronous FPGA architecture") state that static conditions between modules must be done through bus macros and that global clock signals *must* be used for reconfigurable modules according to Xilinx. The author also writes that using asynchronous modules will remove the need for clock distribution.

8.2 Scheduler

A lot of time was spent on developing the scheduler for the HWOS. The ultimate goal was to perform scheduling of FPGA-based tasks. The state loading facility made by Vegard Endresen in [End10] could be used when interrupting modules. However, because the modified version of his backend did not work on the FPGA, this was not done.

However, the scheduler did work when registering processes from a test client. It was shown that the multithreaded software program could perform scheduling of several processes in a round-robin manner. This scheduler can be further optimized by placing some parts of it in hardware. Typically would the placement process be very complex as modules can be placed at many different places on the FPGA. The article in [QDG] states the following:

> The on-line scheduling of HW tasks on PRTR FPGA is much more complicated than SW scheduling. SW tasks only share computing resources in the time dimension, while HW tasks share computing resources in not only the time but also the space dimension. The on-line HW task scheduling algo- rithms are usually very time-consuming.

The focus for the scheduler in this thesis has been to schedule processes in time. The article in [HSM] describes multitasking on the FPGA both through paralell execution on the FPGA and through interrupting processes and replacing them on the FPGA. The main advantage that is described is that sharing of IO-resources becomes easier when only one process is running on the FPGA. One could also imagine that the system would benefit from removing processes waiting on IO-resources from the FPGA.

However, as described in section 3.7.6 and equation 3.1, a critical requirement for an interrupter is the time performed to interrupt a process, save the state and reconfigure the region. Obviously, if the reconfiguration time is very slow, the gain from performing partial reconfiguration would be less. In this case it could be easier to reconfigure the complete FPGA and perform parallel execution of a set of processes on the FPGA. To make the system in this thesis usable, the cause of the long reconfiguration time must be found.

Chapter 9

Conclusion

The results from this thesis has shown that partial self-reconfiguration of synchronous modules is possible on a running FPGA. It has been shown that clock routing for a reconfigurable module can be constrained using Directed Routing. This is similiar to using bus macros for normal signals. The proof of concept is an important result from this thesis. In theory, this solution should be compatible with any reconfigurable system built using bus macros.

A significant problem was the partial reconfiguration time when using this solution. The delay was several seconds and is in contrast to earlier studies in the AHEAD-project. These have shown that the partial reconfiguration time is as low as a few milliseconds.

Experiments on partial self-reconfiguration was performed on the instruction- and data cache backend made by Vegard Endresen. This backend is able to shift data in and out of a reconfigurable module. The results show that this static design can be built successfully with Directed Routing constraint on the clock signal, but when uploading the design to the FPGA, the Suzaku deadlocked and further testing could not be conducted.

The software parts of the HWOS have been restructured and rewritten from scratch. A software based solution for performing simple scheduling of reconfigurable modules has been made. This scheduler is working correct using a round-robin scheduling policy.

Chapter 10

Further work

The work with the complete reconfigurable framework reveals several challenges ahead. A brief discussion of some of them is discussed in this chapter. Some suggestions on how to solve them is also included.

10.1 Better testing of each part of the complete framework

Several parts of the reconfigurable framework could have been better tested. Especially since the complete framework has become quite large, testing each part could make it less unwieldy and easier to maintain.

CLBRead and *icap_write* is working in most cases, but it would be really useful to set up a larger amount of test cases for these programs. For example could some automated tests run on the Suzaku and reconfigure many parts of the entire FPGA. Naturally, to test the area where the static design resides, the static design would have to be moved by reconfiguring the complete configuration. This is a bit problematic since the Suzaku must reboot each time the static has been reconfigured.

Some more unit tests could have been set up on Vegards backend. This will probably make further development on the interrupt procedure of modules much easier.

Some more testing can be done on the scheduler and the version of the HWOS developed in this thesis, especially on the top-level functionality.

It is higly recommended to make use of the Check framework and a test-driven development strategy (see section 3.8.4). It has been set up for the Suzaku-V and is working good for the development platform.

10.2 Further development

10.2.1 Partial self-reconfiguration

The program *CLBRead* can't read out all possible sections of the FPGA. This should be further developed.

When performing partial reconfiguration on the FPGA, it is a bit like "working in the dark". If something fails it is hard to determine what went wrong. An interesting idea is to read back the modified configuration from the FPGA and use the work done by Ingar Hauge [Hau06] to analyze it graphically. Reading back configurations from the FPGA should be possible according to [Xil09]. However, Ingar Hauges analysis software must first be rewritten to support the Virtex-4 bitstream architecture. It is unsure how much work this is and if it is possible at all, but it could potentially increase the knowledge on the partial reconfiguration process done by *icap_write* and make the debug process much easier.

Bibliography

[Alt09] Altera. Understanding metastability in fpgas, 2009.

[AT11] Atmark-Techno. Suzaku developers site. `http://suzaku-en.atmark-techno.com` (Retrieved 6. June 2011), 2011.

[Atm06] Atmark-Techno. *Suzaku Software Manual*, 1.3.1 edition, August 2006.

[Atm07] Atmark-Techno. *Suzaku-V SZ310-U00 Hardware Manual*, 1.0.0.1 edition, June 2007.

[Atm08] Atmark-Techno. *Using NFS for the Root File System*, 2008. `http://suzaku-en.atmark-techno.com/dev/howtos/nfs` (Retrieved 6. February 2011).

[BC00] Daniel P. Bovet and Marco Cesati. Understanding the linux kernel - chapter 10: Process scheduling. `http://oreilly.com/catalog/linuxkernel/chapter/ch10.html` (Retrieved 19. May 2011), 2000.

[che09] Check: A unit testing framework for c. `http://check.sourceforge.net` (Retrieved 22. May 2011), 2009.

[CS10] Jeroen Boydensand Piet Cordemans and Eric Steegmans. Test-driven development of embedded software, January 2010.

[Dan04] Klaus Danne. Memory management to support multitasking on fpga based systems. In *In Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReCon)*, page 21, 2004.

[End09a] Vegard Endresen. Creating a reconfigurable fpga system. NTNU, 2009. Tutorial.

[End09b] Vegard Endresen. Hardware task execution in reconfigurable systems. NTNU, 2009. Project report.

[End10] Vegard Endresen. Hardware-software intercommunication in reconfigurable systems. Master's thesis, NTNU, 2010.

[Eri00] Ken Erickson. Asynchronous fpga risks. In *2000 MALPD International Conference*, 2000.

[Eto07] Emi Eto. Difference-based partial reconfiguration (xilinx, xapp290), 2007.

[Ham08a] Sverre Hamre. Self reconfigurable system on a xilinx spartan3 fpga by using bus macros, 2008. Project report.

[Ham08b] Sverre Hamre. Tutorial for creating a hard/bus macro to the spartan3 fpga, December 2008. Tutorial.

[Ham08c] Sverre Hamre. Tutorial for using atmark techno fpga development environment, 2008. Tutorial.

[Ham09] Sverre Hamre. Framework for self reconfigurable system on a xilinx fpga. Master's thesis, NTNU, 2009.

[Han10] Sindre Hansen. Self reconfiguration of clock networks on fpga. NTNU, 2010. Project report.

[Hau06] Ingar Hauge. Analyse, dekomponering og rekonstruksjon av fpga-konfigurasjoner for ahead. Master's thesis, NTNU, 2006.

[HSM] L. Levinson H. Simmler and R. Manner. Multitasking on fpga co-processors.

[Inc10] Arcturus Networks Inc. uClinux^{TM} – embedded linux microcontroller project – home page. `http://www.uclinux.org/` (Retrieved 14. February 2011), 2010.

[JIA06] XIN JIA. Gapla: A globally asynchronous locally synchronous fpga architecture, 2006.

[Joh06] Mikael Johansson. How to use NFS on SUZAKU-V. `http://staff.aist.go.jp/hirano-s/mikael/Notes/nfs.htm` (Retrieved 9. February 2011), 2006.

[KR06] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. FPGA'06, February 22–24, 2006, Monterey, California, USA, 2006.

[PJC09] Steve J.E. Wilton Peter Jamieson, Wayne Luk and George A. Constantinides. An energy and power consumption analysis of fpga routing architectures. Field-Programmable Technology, 2009.

[Ple04] Roman Plessl. Embedded machine on fpga. Master's thesis, Swiss Federal Institute of Technology Zurich, 2004.

[QDG] Nan Guan Qingxu Deng, Yi Zhang and Zonghua Gu. A unified hw/sw operating system for partially runtime reconfigurable fpga based computer systems. Northeastern University, Shenyang, China.

[QWA09] Subodh Gupta Qiang Wang and Jason Anderson. Clock power reduction for virtex-5 fpgas. FPGA'09, February 22–24, Monterey, California, USA, 2009.

[Sal07] Peter Jay Salzman. The linux kernel module programming guide. `http://www.tldp.org/LDP/lkmpg/2.6/html/index.html` (Retrieved 9. February 2011), May 2007.

[SH09] Jason H. Anderson Safeen Huda, Muntasir Mallick. Clock gating architectures for fpga power reduction. IEEE International Conference on Field-Programmable Logic and Applications (FPL), pp. 112-118, Prague, Czech Republic, 2009.

[SKB02] Li Shang, Alireza Kaviani, and Kusuma Bathala. Dynamic power consumption in virtex[tm]-ii fpga family. In *FPGA*, pages 157–164, 2002.

[Sta05] William Stallings. *Operating systems Internals and Design principles*. Pearson Prentice Hall, 5 edition, 2005.

[Ste05] Jennifer Stephenson. Design guidelines for optimal results in fpgas (altera), 2005.

[suz06] Suzaku-V tutorials. `http://ramsites.net/~wcsleeman/` (Retrieved 10. February 2011), February 2006.

[TBC07] Wayne Luk Tobias Becker and Peter Y.K. Cheung. Enhancing relocatability of partial bitstreams for run-time reconfiguration, 2007.

[ucl11] uclibc. `http://uclibc.org` (Retrieved 22. May 2011), 2011.

[Ung02] Greg Ungerer. uCdot | uClinux merged into main line linux kernel sources. `http://www.ucdot.org/article.pl?sid=02/11/05/0324207` (Retrieved 14. February 2011), November 2002.

[Wol05] W. Wolf. Computers as components : principles of embedded computing system design, chapter 7 hardware accelerators. Elsevier, 2005.

[Xil05] Xilinx. Power management solution guide, July 2005.

[Xil08a] Xilinx. Constraints guide (10.1), 2008.

[Xil08b] Xilinx. Virtex-4 fpga user guide (ug070), 2008.

[Xil09] Xilinx. Virtex-4 fpga configuration user guide (ug071), 2009.

Appendix A

Tutorial for uClinux

This tutorial describes the process of compiling and configuring ATMARK-dist. ATMARK-dist is operating system for the Suzaku boards (using a PowerPC- or Microblaze-processor) and is built upon uClinux. uClinux is a lightweight operating system based on the Linux kernel.

Compiling and configuring ATMARK-dist is a relatively easy task once the correct cross-development tools are installed. This tutorial aims to give a quick overview and help for setting up ATMARK-dist on the Suzaku-V sz410 board. This board has a PowerPC-processor and a Virtex-4 FPGA. The operating system used for development in this tutorial was Debian Squeeze (6.0).

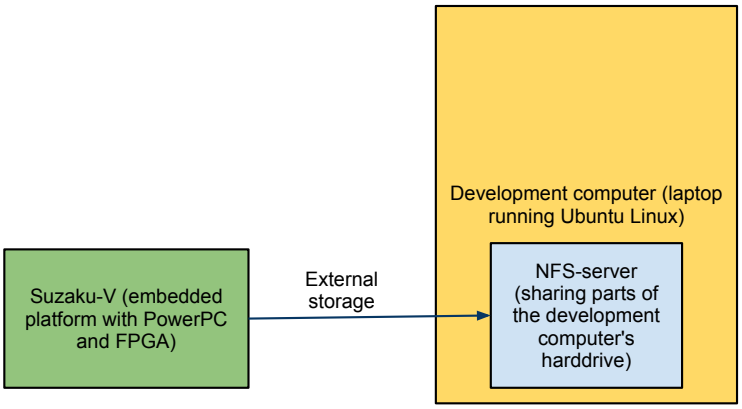

Figure A.1: Concept for using NFS on the Suzaku-V.

A brief description of how to set up the Network File System (NFS) on the Suzaku will be given. This will make it possible to share files between the development computer and the Suzaku-board. The Suzaku-V has 8 MB of non-volatile flash-memory, but the problem is that files can only be permanently added by adding them to the system image. This means that the complete filesystem has to be made ready on the development computer and later uploaded to the Suzaku. A much better alternative is setting up the uClinux-distribution for using NFS as a external storage device as shown in

picture A.1.

A.1 Objective

The most important goal of this tutorial is to make the reader able to set up the cross-development tools needed for compiling the source code in this thesis. A developer in the AHEAD-project should also be able to recompile and configure the Linux-distribution for the embedded platform, for example for including tools or kernel modules. To be effective in the development, NFS can be used as an external storage device for the Suzaku.

Note that one of the objectives is to show how the development enviroment was set up in practise on the given platform, not to document every single part of the process. If there are any problems, please refer to the documentation from Atmark-Techno.

A.2 Prerequisites

The prerequisites for this tutorial are listed in table A.2.

OS on development computer	Debian Squeese 6.0 (kernel: linux-2.6.32-5-686)
OS on embedded platform	atmark-dist-20090318 (kernel: linux-2.6.18-at11)
Embedded platform	Suzaku sz410
Embedded processor	PowerPC
Embedded software manual	Suzaku Software Manual 1.3.1 [Atm06].
Embedded hardware manual	Suzaku-V SZ310-U00 Hardware Manual 1.0.0.1 [Atm07].

Table A.1: Prerequisites for this tutorial.

A.3 Download and compile ATMARK-dist

It is assumed that the reader is using a Linux based operating system for compiling ATMARK-dist. It may be possible to do the compiling on newer versions of for example Ubuntu, but this can cause problems as Ubuntu contains libraries and configurations that may not be compatible with the cross-development tools used in this tutorial. The safest choice is to use the virtual

machine from Atmark Techno, which is based on a stable version of Debian Linux. The newest version of the virtual machine can be downloaded from:
`http://download.atmark-techno.com/suzaku/atde/`
Sverre Hamre has written a tutorial on how to change the default language and add extra hard-drive storage in [Ham08c].

After this is done, download an ISO-image containing the cross-development tools, ATMARK-dist and uCLinux. This ISO-image can be downloaded at:
`http://download.atmark-techno.com/suzaku/iso/`
`sv_20100924.iso` was used and was the latest version when this tutorial was written. As shown in A.1, the file `sv_20100924.iso` should be mounted and the distribution files must be unpacked.

Listing A.1: Downloading and unpacking ATMARK-dist

```
# Download the iso-file.
wget http://download.atmark-techno.com/suzaku/iso/
    sv_20100924.iso

# Make a directory and mount the iso-file to it.
mkdir isomount
chmod 777 isomount
sudo mount -o loop sv_20100924.iso isomount/

# Change directory into suzaku/dist and copy the files.
cd isomount/suzaku/dist
cp atmark-dist-20090318.tar.gz ../../../
cp linux-2.6.18-at11.tar.gz ../../../

# Go back to root directory and unpack the files.
cd ../../../
tar xvf atmark-dist-20090318.tar.gz
tar xvf linux-2.6.18-at11.tar.gz
```

The cross-development tools are in the folder
`isomount/suzaku/cross-dev/powerpc/deb`. Use the command
`sudo dpkg -i [deb-file]` to install the Debian package files. It is probably enough to install the package `atde-essential-powerpc_9_all.deb` and its many dependencies. If two packages depend on each other, install both and leave them unconfigured. Then issue `sudo aptitude upgrade` to resolve dependencies. After this is done, run the code in A.2.

Listing A.2: Compile ATMARK-dist

```
# Link the Linux kernel to atmark-dist and configure the
    kernel.
cd atmark-dist-20090318
ln -s ../linux-2.6.18-at11 ./linux-2.6.x
make menuconfig

# Make choices here as described in the tutorial
# ...

# Run make when finished
make
```

It is possible to configure the kernel after executing `make menuconfig`. A minimal configuration is described below.

- Choose "Vendor/Product Selection".
- Choose "AtmarkTechno" as Vendor and "SUZAKU-V.SZ410" as Product.
- Go to "Exit".
- Choose "Kernel/Library/Defaults Selection".
- Choose "powerpc" as Cross-dev and "uClibc" as Libc Version.
- Mark "Customize Kernel Settings (NEW)" and "Customize Vendor/User Settings (NEW)".
- Go to "Exit" and "Exit".

This is actually the minimum configuration needed for ATMARK-dist to run on the Suzaku-V. If the cross-development tools are installed, it should be possible to just run `make` after the rest of the configuration is done. It is recommended to keep the default network settings if facilities as telnet/FTP and Network File System (NFS) are needed. You could also choose other programs you find useful (for example `dmesg` among the Busybox-programs).

The image resides in `images/image.bin`. If adding files to the file system is needed, put them in one of the folders in romfs/ and run the command `make image` in the root directory afterwards.

It should be possible to login into the Suzaku by running `telnet [IP-address]`. If the Suzaku is connected to the local network, the IP-address can be found by running the script in A.3. This will only work if your router assign IP-addresses on the form 192.168.0.x. If not, change the script accordingly.

Listing A.3: Get all IP-addresses on LAN

```
1 for ip in $(perl -e '$,="\n"; print 1 .. 254;') ; do ping -t
      1 -c 1 192.168.0.$ip>/dev/null; [ $? -eq 0 ] && echo "
    192.168.0.$ip UP" || : ; done
```

Upload the new image to a local or external server and run the commands in A.4 to reflash the uClinux-image.

Listing A.4: Telnet into the Suzaku and upload a new uClinux image

```
1 telnet [IP-address]
2 netflash -r image http://local.server.name/suzaku/image.bin
```

A.4 Setting up Network File System (NFS) on Suzaku and development machine

Since the Suzaku has flash memory, everything that is created by user in the file system will be lost after reboot of the device. It is possible to upload

them every time using FTP and/or automatic scripts, but this can be quite tedious. Also, the amount of memory can be limiting. Suzaku-V has 8MB flash memory and a huge amount of the memory is needed for ATMARK-dist. To solve these problems, the development computer can be set up as a NFS-server. That means that the Suzaku will have access to a shared folder on the development computer and the files will only be loaded into the Suzaku's memory whenever they're executed or opened on the Suzaku.

Note that there are two ways of using NFS on the Suzaku. These are:

- Sharing a single folder between the Suzaku and the development computer.
- Mount the complete Suzaku-filesystem in a folder on the development computer. Then configure the Suzaku to boot from this folder across the network.

The first option is discussed in this tutorial. The last option is probably a bit more complex and is discussed in the tutorial from Atmark-Techno [Atm08]. Refer to this tutorial if you need to mount the complete filesystem across the network.

The process of setting up NFS for a single folder is briefly discussed here. The steps included are listed below and will be discussed in the subsequent sections.

- Set up the development-PC to share a folder on the network.
- Set up the Suzaku-V as a client. This essentially means mounting a folder.

On Debian based systems, the NFS server can be installed using the code in A.5.

Listing A.5: Install and configure NFS in Debian

```
1  # Install the NFS server (run as root)
2  apt-get install nfs-kernel-server
3
4  # Create a shared folder
5  mkdir -p  /path-of-your-home-folder/suzaku_shared
6
7  # Specify the folder as shared.
8  # As root, open the file /etc/exports and add:
9  /path-of-your-home-folder/suzaku_shared
       192.168.0.0/255.255.255.0(rw,sync)
10 # The IP-address 192.168.0.0 must be adjusted for your
       router.
11
12 # After you have done this, run the following command to
       restart NFS-server (as root):
13 /etc/init.d/nfs-kernel-server restart
```

Once the server is configured, make sure ATMARK-dist is compiled with support for NFS (it should be enabled by default). To mount the shared folder on the Suzaku, type the code in A.6. Without the option `nolock` the mount-point will be locked the first 5 minutes. The options `rsize` and `wsize` specifies the buffer size and makes it possible to load/copy larger files [Joh06].

Listing A.6: Mount a NFS folder on the Suzaku

```
# Change to a directory where you have write-access
cd /var

# Create the directory where you want to mount the NFS
    folder
mkdir -p suzaku_shared
chmod 777 suzaku_shared

# Mount the shared NFS folder
# Replace 192.168.0.133 with the IP-address to your
    development computer
mount -o nolock,rsize=4096,wsize=4096 -t nfs 192.168.0.133:/
    address-to-your-home-folder/suzaku_shared /var/
    suzaku_shared
```

This could be done more permanent by creating a folder and a file in the `romfs`-folder of the root-directory where ATMARK-dist is compiled (see A.7).

Listing A.7: Adding an entry to fstab to permanently mount shared folder

```
# Current directory is where ATMARK-dist is compiled (atmark
    -dist-20090318 in this example).
# Create a directory in romfs/mnt.
cd romfs/mnt
mkdir -p suzaku_shared
chmod 777 suzaku_shared

# Add the following text to romfs/etc/fstab (create the file
    )
# Replace 192.168.0.165 with the IP-address to the
    development computer
192.168.0.165:/path-to-your-home-folder/suzaku_shared    /
    var/suzaku_shared    nfs    nolock,rsize=4096,wsize=4096
        0 0

# At the end of the file romfs/etc/init.d/rc.local, add:
mount -a
# This will mount all filesystems when starting uClinux.

# Change directory to the root directory (atmark-dist
    -20090318 in this example).
# Run the following command to update the file system.
make image
```

This will automount the NFS-folder each time uClinux starts up.

A.5 Creating kernel modules

There are numereous sources documenting the act of writing your own kernel module. For generic Linux modules, a guide called "The Linux Kernel Module Programming Guide" from "The Linux Documentation Project" [Sal07] gives

a quick introduction and is a good place to start. Some extra consideration must be taken when writing kernel modules for the Suzaku-V. It is important that the uClinux-kernel must be compiled with support for loadable modules.

A.6 Compiling HWICAP-driver for uClinux

The process of compiling the Hardware Internal Configuration Access Port (HWICAP) is described in [Ham09, page 27]. Be careful when reading the source code on these pages, as the output from the command `diff` can be misunderstood.

Appendix B

Compiling the HWOS-code

The software code of the HWOS are in the directory *hwos_sw*. The following are the directory structure of this source code and strongly related parts.

libhwos/
: Forms a library (libhwos.a) to be used when building the top-level entities of the HWOS. Functionality that (potentially) has to be reused are typically in this library.

hwos_daemon/
: Code for the top-level daemon (server) to run in the background on the Suzaku-board. Contains top-level entities for the message server, the scheduler, a simple placer and a timer. These run in parallel threads.

tests/
: Contains automated tests for the library and the daemon. Because the library will be reused a lot, the most elaborate and important tests are for the library.

docs/
: Documentation for the library and the daemon. The interfaces (.h-files) are the parts that have been documented the most.

There are several other directories that are not really a part of the HWOS-source. These are listed below.

check-0.9.8/
: Original source code for the Check framework (found in [che09]). Installation instructions for the HWOS-framework can be found in *INSTALL_CHECK.txt*, but there has also been made precompiled versions as described under. Check was compiled for the x86-architecture using glibc (standard C-library in desktop-Linux) and for the PowerPC-architecture using uClibc (see section 3.2.7).

check-x86/
A precompiled version of the Check library for the x86-architecture and glibc-version of the HWOS.

check-powerpc/
A precompiled version of the Check library for the PowerPC-architecture and uClibc-version of the HWOS.

bitfiles/
Files containing the FPGA-bitstream description for the processes in the system.

logs/
Log files for the daemon and the tests.

The HWOS can be compiled for the standard Linux x86-architecture or the embedded uClinux using the PowerPC microprocessor. To configure the makefiles for the x86-architecture using the standard C-library glibc, just do the following:

- Go to the root directory *hwos_sw*. Run `./configure`
  - The configuration files will now be automatically written to compile for the x86-architecture using the C-library glibc.

To compile it for uClinux and the PowerPC using uClibc, do the following:

- Go to the root directory *hwos_sw*. Run `./configure uclinux`
  - The configuration files will now be automatically written to compile for the PowerPC-architecture using the C-library uclibc.

If all the standard build tools for the Linux-distribution are installed, this should be successfull. The configuration process for the uClinux-distribution assumes that a cross-development enviroment has been set up.

Now, for both architectures:

- Run `make`
  - The library, the daemon and the tests will now be compiled. If you do not want to make all of them, run `make <entity>`, where `<entity>` can be `library`, `daemon` or `test`. This will compile the library, the daemon or the tests respectively. Run `make clean` to remove all executables and other files generated by `make`.
- The daemon can be started by changing directory to *hwos_daemon/* and running `./hwos`

  - This will start the daemon in the background. The message server will be ready to receive messages from a client application.

- The tests can be started by changing directory to *tests/* and running `./hwos_test`

  - First of all this will test the HWOS-library, but the program will also provide options for running tests on the daemon and the FPGA-dependent parts of the library.

Appendix C

VHDL-code

Listing C.1: Top-level entity for base design (one flip-flop) in EDK

```
1  -- Reconfigurable top-level module for the tests with one
       flip-flop.
2  -- From my project report.
3  -- Some (not very relevant) parts are omitted and marked
       (...).
4  -- See the appended ZIP-file for complete files.
5  --
6  -- Sindre Hansen (2010/2011).
7
8  (...)
9
10 signal bm_l2r_in : std_logic_vector(0 to 15);
11 signal bm_l2r_out : std_logic_vector(0 to 15);
12
13 signal bm_r2l_in : std_logic_vector(0 to 15);
14 signal bm_r2l_out : std_logic_vector(0 to 15);
15
16 signal clk_buffer_out : std_logic;
17 signal clk_buffer_ce : std_logic;
18
19 (...)
20
21 -- The user logic basically maps the input and output of
22 -- the bus macros to HW/SW-accessible registers.
23 USER_LOGIC_I : entity reconf_v1_00_a.user_logic
24 (...)
25 port map
26 (
27   in_16 => bm_r2l_out,
28   out_16 => bm_l2r_in,
29 (...)
30 );
31
32 R : entity reconfigurable_module
33 port map
34 (
35   clk_in => clk_buffer_out,
36   in_16 => bm_l2r_out,
37   out_16 => bm_r2l_in
38 );
39
40
41 -- Connect one of the outputs from the HWSW-register to
       clk_buffer_ce.
42 -- This means the clock buffer's CE-input can be controlled
       by a Linux-app.
43 clk_buffer_ce <= bm_l2r_in(7);
```

```
44 |                                                                    |
45 |-- Instantiate global clock buffer explicitely to enable/ disable clock |
46 |BUFGCE_inst : BUFGCE                                                |
47 |port map (                                                          |
48 |  -- Clock buffer output (connected to reconf. module)              |
49 |  O => clk_buffer_out,                                              |
50 |  -- Clock enable input                                             |
51 |  CE => clk_buffer_ce,                                              |
52 |  -- Clock buffer input (Using same as to user_logic/backend module |
53 |  I => ipif_Bus2IP_Clk                                              |
54 |);                                                                  |
55 |                                                                    |
56 |                                                                    |
57 |bm_l2r_0 : component busmacro_xc4v_l2r_async_narrow                 |
58 |port map(                                                           |
59 |  input0 =>bm_l2r_in(0),                                            |
60 |  input1 =>bm_l2r_in(1),                                            |
61 |  input2 =>bm_l2r_in(2),                                            |
62 |  input3 =>bm_l2r_in(3),                                            |
63 |  input4 =>bm_l2r_in(4),                                            |
64 |  input5 =>bm_l2r_in(5),                                            |
65 |  input6 =>bm_l2r_in(6),                                            |
66 |  input7 =>bm_l2r_in(7),                                            |
67 |  output0 =>bm_l2r_out(0),                                          |
68 |  output1 =>bm_l2r_out(1),                                          |
69 |  output2 =>bm_l2r_out(2),                                          |
70 |  output3 =>bm_l2r_out(3),                                          |
71 |  output4 =>bm_l2r_out(4),                                          |
72 |  output5 =>bm_l2r_out(5),                                          |
73 |  output6 =>bm_l2r_out(6),                                          |
74 |  output7 =>bm_l2r_out(7)                                           |
75 |);                                                                  |
76 |                                                                    |
77 |bm_l2r_1 : component busmacro_xc4v_l2r_async_narrow                 |
78 |port map(                                                           |
79 |  input0 =>bm_l2r_in(8),                                            |
80 |  input1 =>bm_l2r_in(9),                                            |
81 |  input2 =>bm_l2r_in(10),                                           |
82 |  input3 =>bm_l2r_in(11),                                           |
83 |  input4 =>bm_l2r_in(12),                                           |
84 |  input5 =>bm_l2r_in(13),                                           |
85 |  input6 =>bm_l2r_in(14),                                           |
86 |  input7 =>bm_l2r_in(15),                                           |
87 |  output0 =>bm_l2r_out(8),                                          |
88 |  output1 =>bm_l2r_out(9),                                          |
89 |  output2 =>bm_l2r_out(10),                                         |
90 |  output3 =>bm_l2r_out(11),                                         |
91 |  output4 =>bm_l2r_out(12),                                         |
92 |  output5 =>bm_l2r_out(13),                                         |
93 |  output6 =>bm_l2r_out(14),                                         |
94 |  output7 =>bm_l2r_out(15)                                          |
95 |);                                                                  |
96 |                                                                    |
97 |bm_r2l_0 : component busmacro_xc4v_r2l_async_narrow                 |
98 |port map(                                                           |
99 |  input0 =>bm_r2l_in(0),                                            |
100|  input1 =>bm_r2l_in(1),                                            |
101|  input2 =>bm_r2l_in(2),                                            |
102|  input3 =>bm_r2l_in(3),                                            |
103|  input4 =>bm_r2l_in(4),                                            |
104|  input5 =>bm_r2l_in(5),                                            |
105|  input6 =>bm_r2l_in(6),                                            |
106|  input7 =>bm_r2l_in(7),                                            |
107|  output0 =>bm_r2l_out(0),                                          |
108|  output1 =>bm_r2l_out(1),                                          |
```

```
109 |   output2 =>bm_r2l_out(2),                                    |
110 |   output3 =>bm_r2l_out(3),                                    |
111 |   output4 =>bm_r2l_out(4),                                    |
112 |   output5 =>bm_r2l_out(5),                                    |
113 |   output6 =>bm_r2l_out(6),                                    |
114 |   output7 =>bm_r2l_out(7)                                     |
115 | );                                                            |
116 |                                                               |
117 | bm_r2l_1 : component busmacro_xc4v_r2l_async_narrow            |
118 | port map(                                                     |
119 |   input0 =>bm_r2l_in(8),                                      |
120 |   input1 =>bm_r2l_in(9),                                      |
121 |   input2 =>bm_r2l_in(10),                                     |
122 |   input3 =>bm_r2l_in(11),                                     |
123 |   input4 =>bm_r2l_in(12),                                     |
124 |   input5 =>bm_r2l_in(13),                                     |
125 |   input6 =>bm_r2l_in(14),                                     |
126 |   input7 =>bm_r2l_in(15),                                     |
127 |   output0 =>bm_r2l_out(8),                                    |
128 |   output1 =>bm_r2l_out(9),                                    |
129 |   output2 =>bm_r2l_out(10),                                   |
130 |   output3 =>bm_r2l_out(11),                                   |
131 |   output4 =>bm_r2l_out(12),                                   |
132 |   output5 =>bm_r2l_out(13),                                   |
133 |   output6 =>bm_r2l_out(14),                                   |
134 |   output7 =>bm_r2l_out(15)                                    |
135 | );                                                            |
136 |                                                               |
137 | (...)                                                         |
```

Listing C.2: Top-level entity for a reconfigurable module (one flip-flop) in ISE

```
1  |-- This is a top level entity for a reconfigurable            |
2  |-- module built in ISE.                                       |
3  |--                                                            |
4  |-- Sindre Hansen (2011).                                      |
5  |                                                              |
6  | library ieee;                                                |
7  | use ieee.std_logic_1164.all;                                 |
8  | use ieee.std_logic_arith.all;                                |
9  | use ieee.std_logic_unsigned.all;                             |
10 |                                                              |
11 | library unisim;                                              |
12 | use unisim.vcomponents.all;                                  |
13 |                                                              |
14 | entity reconf is                                             |
15 |     port (                                                   |
16 |    clk_in    : in   std_logic;                               |
17 |    in_16     : in   std_logic_vector(0 to 15);               |
18 |    out_16    : out  std_logic_vector(0 to 15)                |
19 |   );                                                         |
20 | end entity reconf;                                           |
21 |                                                              |
22 |                                                              |
23 | architecture IMP of reconf is                                |
24 |                                                              |
25 | signal bm_l2r_in : std_logic_vector(0 to 15);                |
26 | signal bm_l2r_out : std_logic_vector(0 to 15);               |
27 |                                                              |
28 | signal bm_r2l_in : std_logic_vector(0 to 15);                |
29 | signal bm_r2l_out : std_logic_vector(0 to 15);               |
30 |                                                              |
31 | signal clk_buffer_out : std_logic;                           |
32 | signal clk_buffer_ce : std_logic;                            |
33 |                                                              |
34 | component busmacro_xc4v_l2r_async_narrow is                  |
35 |       port (                                                 |
36 |         input0 : in std_logic;                               |
```

```
37          input1 : in std_logic;
38        input2 : in std_logic;
39          input3 : in std_logic;
40        input4 : in std_logic;
41          input5 : in std_logic;
42        input6 : in std_logic;
43          input7 : in std_logic;
44          output0 : out std_logic;
45          output1 : out std_logic;
46        output2 : out std_logic;
47          output3 : out std_logic;
48        output4 : out std_logic;
49          output5 : out std_logic;
50        output6 : out std_logic;
51          output7 : out std_logic
52        );
53    end component;
54
55
56  component busmacro_xc4v_r2l_async_narrow is
57        port (
58          input0 : in std_logic;
59          input1 : in std_logic;
60        input2 : in std_logic;
61          input3 : in std_logic;
62        input4 : in std_logic;
63          input5 : in std_logic;
64        input6 : in std_logic;
65          input7 : in std_logic;
66          output0 : out std_logic;
67          output1 : out std_logic;
68        output2 : out std_logic;
69          output3 : out std_logic;
70        output4 : out std_logic;
71          output5 : out std_logic;
72        output6 : out std_logic;
73          output7 : out std_logic
74        );
75    end component;
76
77
78
79  begin
80
81    -- Connect the outputs & inputs to the bus macro outputs &
          inputs
82    out_16 <= bm_r2l_out;
83    bm_l2r_in <= in_16;
84
85    R : entity reconfigurable_module
86        port map
87        (
88      clk_in => clk_buffer_out,
89      in_16 => bm_l2r_out,
90      out_16 => bm_r2l_in
91    );
92
93    -- Connect one of the outputs from the HWSW-register to
          clk_buffer_ce.
94    -- This means the clock buffer's CE-input can be
          controlled by a Linux-app.
95    clk_buffer_ce <= bm_l2r_in(2);
96
97    -- Instantiate global clock buffer explicitely to enable/
          disable clock
98    BUFGCE_inst : BUFGCE
99    port map (
100               O => clk_buffer_out,    -- Clock buffer output (
                      connected to reconf. module)
101               CE => clk_buffer_ce,    -- Clock enable input
```

```
102 |              I => clk_in              -- Clock buffer input |
103 |            );                                              |
104 |                                                            |
105 |                                                            |
106 |   bm_l2r_0 : component busmacro_xc4v_l2r_async_narrow      |
107 |     port map(                                              |
108 |         --                                                 |
109 |       input0 =>bm_l2r_in(0),                              |
110 |       input1 =>bm_l2r_in(1),                              |
111 |       input2 =>bm_l2r_in(2),                              |
112 |       input3 =>bm_l2r_in(3),                              |
113 |       input4 =>bm_l2r_in(4),                              |
114 |       input5 =>bm_l2r_in(5),                              |
115 |       input6 =>bm_l2r_in(6),                              |
116 |       input7 =>bm_l2r_in(7),                              |
117 |       output0 =>bm_l2r_out(0),                            |
118 |       output1 =>bm_l2r_out(1),                            |
119 |       output2 =>bm_l2r_out(2),                            |
120 |       output3 =>bm_l2r_out(3),                            |
121 |       output4 =>bm_l2r_out(4),                            |
122 |       output5 =>bm_l2r_out(5),                            |
123 |       output6 =>bm_l2r_out(6),                            |
124 |       output7 =>bm_l2r_out(7)                             |
125 |                                                            |
126 |     );                                                     |
127 |                                                            |
128 |     bm_l2r_1 : component busmacro_xc4v_l2r_async_narrow    |
129 |     port map(                                              |
130 |                                                            |
131 |       input0 =>bm_l2r_in(8),                              |
132 |       input1 =>bm_l2r_in(9),                              |
133 |       input2 =>bm_l2r_in(10),                             |
134 |       input3 =>bm_l2r_in(11),                             |
135 |       input4 =>bm_l2r_in(12),                             |
136 |       input5 =>bm_l2r_in(13),                             |
137 |       input6 =>bm_l2r_in(14),                             |
138 |       input7 =>bm_l2r_in(15),                             |
139 |       output0 =>bm_l2r_out(8),                            |
140 |       output1 =>bm_l2r_out(9),                            |
141 |       output2 =>bm_l2r_out(10),                           |
142 |       output3 =>bm_l2r_out(11),                           |
143 |       output4 =>bm_l2r_out(12),                           |
144 |       output5 =>bm_l2r_out(13),                           |
145 |       output6 =>bm_l2r_out(14),                           |
146 |       output7 =>bm_l2r_out(15)                            |
147 |                                                            |
148 |     );                                                     |
149 |                                                            |
150 |     bm_r2l_0 : component busmacro_xc4v_r2l_async_narrow    |
151 |     port map(                                              |
152 |                                                            |
153 |       input0 =>bm_r2l_in(0),                              |
154 |       input1 =>bm_r2l_in(1),                              |
155 |       input2 =>bm_r2l_in(2),                              |
156 |       input3 =>bm_r2l_in(3),                              |
157 |       input4 =>bm_r2l_in(4),                              |
158 |       input5 =>bm_r2l_in(5),                              |
159 |       input6 =>bm_r2l_in(6),                              |
160 |       input7 =>bm_r2l_in(7),                              |
161 |       output0 =>bm_r2l_out(0),                            |
162 |       output1 =>bm_r2l_out(1),                            |
163 |       output2 =>bm_r2l_out(2),                            |
164 |       output3 =>bm_r2l_out(3),                            |
165 |       output4 =>bm_r2l_out(4),                            |
166 |       output5 =>bm_r2l_out(5),                            |
167 |       output6 =>bm_r2l_out(6),                            |
168 |       output7 =>bm_r2l_out(7)                             |
```

```
169 |
170 |     );
171 |
172 |
173 |
174 |     bm_r2l_1 : component busmacro_xc4v_r2l_async_narrow
175 |     port map(
176 |
177 |       input0 =>bm_r2l_in(8),
178 |       input1 =>bm_r2l_in(9),
179 |       input2 =>bm_r2l_in(10),
180 |       input3 =>bm_r2l_in(11),
181 |       input4 =>bm_r2l_in(12),
182 |       input5 =>bm_r2l_in(13),
183 |       input6 =>bm_r2l_in(14),
184 |       input7 =>bm_r2l_in(15),
185 |       output0 =>bm_r2l_out(8),
186 |       output1 =>bm_r2l_out(9),
187 |       output2 =>bm_r2l_out(10),
188 |       output3 =>bm_r2l_out(11),
189 |       output4 =>bm_r2l_out(12),
190 |       output5 =>bm_r2l_out(13),
191 |       output6 =>bm_r2l_out(14),
192 |       output7 =>bm_r2l_out(15)
193 |
194 |     );
195 |
196 | end IMP;
```

Listing C.3: Reconfigurable module (one flip-flop)

```
1  | -- Reconfigurable module for the tests with one flip-flop.
2  | -- From my project report.
3  | -- Some (not very relevant) parts are omitted and marked
   |    (...).
4  | -- See the appended ZIP-file for complete files.
5  | --
6  | -- Sindre Hansen (2010/2011).
7  |
8  | library (...)
9  |
10 | entity reconfigurable_module is
11 |   port (
12 |    clk_in    : in   std_logic;
13 |    in_16     : in   std_logic_vector(0 to 15);
14 |    out_16    : out  std_logic_vector(0 to 15)
15 |   );
16 | end entity reconfigurable_module;
17 |
18 | architecture arch of reconfigurable_module is
19 | begin
20 |
21 |   -- D-flip-flop
22 |   -- in_16
23 |   --    bit 0: d
24 |   --    bit 1: rst
25 |   --    bit 2-15 : not used (but connected to output to avoid
   |         error messages)
26 |   -- out_16
27 |   --    bit 0: q
28 |   --    bit 1-14: not used (but connected to input bits 2-15
   |         anyway)
29 |   --    bit 15: not used (set to '0')
30 |
31 |   out_16(1 to 14) <= in_16(2 to 15);
32 |   out_16(15) <= '0';
33 |
34 |   OUT_PROC : process(clk_in, in_16(1))
```

```
35 |   begin
36 |     if (in_16(1)='1') then
37 |       out_16(0) <= '0';
38 |     elsif (clk_in'EVENT and clk_in='1') then
39 |       out_16(0) <= in_16(0);
40 |     end if;
41 |   end process OUT_PROC;
42 | end arch;
```

Listing C.4: Reconfigurable module (three flip-flops)

```
1  | -- Reconfigurable top-level module for the tests with one
   |    flip-flop.
2  | -- From my project report.
3  | -- Some (not very relevant) parts are omitted and marked
   |    (...).
4  | -- See the appended ZIP-file for complete files.
5  | --
6  | -- Sindre Hansen (2010/2011).
7  |
8  | library (...)
9  |
10 | entity reconfigurable_module is
11 |   port (
12 |    clk_in    : in   std_logic;
13 |    in_16     : in   std_logic_vector(0 to 15);
14 |    out_16    : out  std_logic_vector(0 to 15)
15 |   );
16 | end entity reconfigurable_module;
17 |
18 | architecture arch of reconfigurable_module is
19 |   signal clock_enable : std_logic;
20 |   signal rst_in : std_logic;    -- Reset DFF
21 |   signal reg0_in : std_logic;   -- DFF input
22 |   signal reg0_out : std_logic;  -- DFF output
23 |   signal reg1_in : std_logic;   -- DFF input
24 |   signal reg1_out : std_logic;  -- DFF output
25 |   signal reg2_in : std_logic;   -- DFF input
26 |   signal reg2_out : std_logic;  -- DFF output
27 | begin
28 |
29 |   -- D-flip-flop
30 |   -- in_16
31 |   --    bit 0-2: d0-d2
32 |   --    bit 3: rst
33 |   --    bit 4-15 : not used (but connected to output to avoid
   |         error messages)
34 |   -- out_16
35 |   --    bit 0-2: q0-q2
36 |   --    bit 3-14: not used (but connected to input bits 4-15
   |       anyway)
37 |   --    bit 15: not used (set to '0')
38 |
39 |   reg0_in <= in_16(0);
40 |   reg1_in <= in_16(1);
41 |   reg2_in <= in_16(2);
42 |   rst_in <= in_16(3);
43 |   out_16(0) <= reg0_out;
44 |   out_16(1) <= reg1_out;
45 |   out_16(2) <= reg2_out;
46 |   out_16(3 to 14) <= in_16(4 to 15);
47 |   out_16(15) <= '0';
48 |
49 |   OUT_PROC : process(clk_in, rst_in)
50 |   begin
51 |     if (rst_in='1') then
52 |       reg0_out <= '0';
53 |       reg1_out <= '0';
```

```
54        reg2_out <= '0';
55      elsif (clk_in'EVENT and clk_in='1') then
56        reg0_out <= reg0_in;
57        reg1_out <= reg1_in;
58        reg2_out <= reg2_in;
59      end if;
60    end process OUT_PROC;
61
62  end arch;
```

Appendix D

HWOS

This is some of the code from the HWOS. It is highly recommended to read the actual code in the ZIP-file or the Doxygen-documentation.

D.1 The HWOS daemon

These are the source files (*.c) and the header files for the top-level entities of the HWOS daemon, the message server, the scheduler, the placer and the timer.

Listing D.1: hdaemon.h

```
1  /*! \file hwos.h
2   *  \brief [Hardware OS Interface] Top level daemon for HWOS.
3   *
4   *  This daemon communicates with a number of hardware modules through device drivers.
5   *  The main goal of this software is to schedule run time on the FPGA for different modules.
6   *  Another goal is to allocate/deallocate memory for the module.
7   *
8   *  Original author (2010): Vegard Endresen
9   *
10  *  Modified (2011) by: Sindre Hansen
11  *      - Rewrote daemon.
12  *      - Rewrote message interface and HWOS-library.
13  *      - Added scheduler.
14  *      - Added placer.
15  *      - Added threads for message server and scheduler.
16  */
17
18
19 #include "hstructures.h"
20
21 #ifndef HDAEMON_H
22 #define HDAEMON_H
23
24 #include "../platform_defines.h"
25 #if SUZAKU_BUILD
26         #define DAEMON_LOG "/var/hwos_daemon.log"
27 #else
28         #define DAEMON_LOG "../logs/hwos_daemon.log"
```

```
29 #endif
30
31 //! Device file root folder on Suzaku.
32 #define DEVICE_ROOT "/var/tmp/dev"
33
34 #endif
```

Listing D.2: hdmsg.h

```
1  /*! \file hwos.h
2   *  \brief [Hardware OS Message Server] Top level daemon for
        HWOS message server.
3   *
4   *  Author (2011): Sindre Hansen
5   */
6
7
8  #ifndef HDMSG_H
9  #define HDMSG_H
10
11 enum hdmevent {
12         HDME_QUIT                                  //! Quit the
               message server thread.
13 };
14
15 /*! \brief Main function for message server thread.
16  */
17 void* hdmsg_main();
18 int hdmsg_notify(enum hdmevent event);
19
20 /*! \brief Send a dummy/sync message to the message server.
21  *
22  * @return 0 on success. Negative on failure.
23  */
24 int hdmsg_synchronize();
25
26 #endif
```

Listing D.3: hdplacer.h

```
1  /*! \file hdplacer.h
2   *  \brief [Hardware OS Placer] Top level module for
        placement on FPGA.
3   *
4   *  The responsibilities of this module is as follows:
5   *
6   *  - The module also takes care of periodically
        interrupting the process running on the FPGA and
7   *  saving it's state. It also writes the state back to a
        process running on the FPGA if it is
8   *  to be resumed.
9   *
10  *  - This module takes care of placement (of a process
        chosen by the scheduler) on the FGPA.
11  *  In this version of the HWOS, the module always places
        the reconfigurable module at
12  *  the same place using ICAP.
13  *
14  *  Original author (2010): Sindre Hansen
15  *
16  */
17
18 #ifndef HDPLACER_H
19 #define HDPLACER_H
20
21
22 enum hdpevent {
23         HDPE_QUIT                                  //! Quit the placer
               thread.
```

```
24          ,HDPE_TIMER                         //! The timeslice
                has expired.
25  };
26
27  void* hdplacer_main();
28  int hdplacer_notify(enum hdpevent event);
29
30  #endif
```

Listing D.4: hdsched.h

```
1   /*! \file hwos.h
2    *  \brief [Hardware OS Scheduler] Top level interface for
         scheduler.
3    *
4    *  This defines the interface at top-level for the
         scheduler. This is
5    *  typically only functions to notify the scheduler on
         events etc.
6    *
7    *  This module pretty much works as a server.
8    *
9    *  Original author (2011): Sindre Hansen
10   */
11
12  #include <hprocess.h>
13
14  #ifndef HDSCHED_H
15  #define HDSCHED_H
16
17
18  enum hdsevent {
19  HDSE_NEW_PROCESS                    //! A new process has
        arrived to the HWOS.
20  ,HDSE_QUIT                          //! Quit the placer
        thread.
21  ,HDSE_SHORT                         //! Perform short-term
        scheduling.
22  ,HDSE_RESCHED_TIMER                 //! Reschedule process (
        was timer interrupted).
23  ,HDSE_RESCHED_PRIORITY              //! Reschedule process (
        was replaced by higher priority process).
24  ,HDSE_RESCHED_IO                    //! Reschedule process (
        needed unavailable IO).
25  };
26
27  enum hdsreturn {
28  HDSR_NO_BITFILE = -2                  //! Bitfile does not
        exist.
29  ,HDSR_TOO_MANY = -1                   //! Too many processes
        in system already.
30  ,HDSR_READY = 0                       //! Process has been
        added to a ready queue.
31  };
32
33  /*! \brief Main function for scheduler thread.
34   */
35  void* hdsched_main();
36
37  /*! \brief Reschedule the given process.
38   *
39   *  This function is called when the given an
40   *  event has occured for the given process.
41   *
42   *  @param process The process to be rescheduled.
43   *  @return 0 on success. Negative on failure.
44   */
45  int hdsched_reschedule(struct hprocess* process, enum
        hdsevent event);
```

```
46
47 /*! \brief Notify the scheduler on new event.
48  *
49  *  @param event The notified event.
50  */
51 int hdsched_notify(enum hdsevent event);
52
53 /*! \brief Return the number of accepted processes in the
       scheduler.
54  *
55  *  This is the number of all processes that has been
       accepted.
56  *  It does not include processes in the HPS_NEW-queues.
57  *
58  *  This number can be modifed by the scheduler at the same
59  *  time and is therefore secured with a mutex.
60  *
61  *  @return Number of processes.
62  */
63 int hdsched_processes_number();
64
65 /*! \brief Add a new process to one of the HPS_NEW-queues.
66  *
67  *  The HPS_NEW-queues can be accessed by another thread at
       the same
68  *  time and is therefore secured with a mutex.
69  *
70  *  @param process The process to be added.
71  *  @return 0 on success. Negative on failure.
72  */
73 int hdsched_add_new_process(struct hprocess* process);
74
75 /*! \brief Get process in front of the HPS_NEW-queue with
       highest priority.
76  *
77  *  The HPS_NEW-queues can be modifed by another thread at
       the same
78  *  time and is therefore secured with a mutex.
79  *
80  *  @return The process.
81  */
82 struct hprocess* hdsched_get_new_process();
83
84
85 /*! \brief Get next process to be placed on the FPGA.
86  *
87  *  The process can be set by the scheduler at the same
88  *  time and is therefore secured with a mutex.
89  *
90  *  @return The process.
91  */
92 struct hprocess* hdsched_get_next_process();
93
94 #endif
```

Listing D.5: hdtimer.h

```
1 /*! \file hdtimer.h
2  *  \brief [Hardware OS Timer] Timer for the HWOS Placer.
3  *
4  *  This module simply generates a timer interrupt to the
       placer module
5  *  at the interval specified by hdtimer_get_timeslice().
6  *
7  *  Original author (2010): Sindre Hansen
8  *
9  */
10
11
```

```
12 #ifndef HDTIMER_H
13 #define HDTIMER_H
14
15 enum hdtevent {
16         HDTE_QUIT                                   //! Quit the timer
               thread.
17 };
18
19
20 int hdtimer_notify(enum hdtevent event);
21 const int hdtimer_get_timeslice();
22 void* hdtimer_main();
23
24 #endif
```

Listing D.6: hdaemon.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <signal.h>
5  #include <pthread.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8
9  #include <hlog.h>
10 #include <hmem.h>
11
12 #include "hdaemon.h"
13 #include "hdsched.h"
14 #include "hdmsg.h"
15 #include "hdplacer.h"
16 #include "hdtimer.h"
17
18
19 //! Initialize this process as a daemon (Linux actually has
       it's own interface for this).
20 static int init_daemon_();
21 static void signal_handler_(int sig);
22
23
24 static void signal_handler_(int sig)
25 {
26         switch (sig) {
27                 case SIGINT:
28                 case SIGTERM:
29                         hdmsg_notify(HDME_QUIT);
30                         hdmsg_synchronize();
31                         break;
32         }
33 }
34
35 static int init_daemon_()
36 {
37         // Fork off the parent process.
38         // Exit the parent process if success.
39         pid_t pid;
40         pid = fork();
41         if (pid < 0) {
42                 return -1;
43         } else if (pid > 0)
44                 exit(EXIT_SUCCESS);
45
46         // Change the file mode mask (should not inherit
               umask from parent).
47         umask(0);
48
49         // Create a new SID for the child process.
```

```
50          pid_t sid;
51          sid = setsid();
52          if (sid < 0) {
53                  return -1;
54          }
55
56          // Change the current working directory.
57          if ((chdir("/")) < 0) {
58                  return -1;
59          }
60
61          // Close out the standard file descriptors.
62          close(STDIN_FILENO);
63          close(STDOUT_FILENO);
64          close(STDERR_FILENO);
65
66          hlog_write(HLOG_NORMAL, "Daemon initialized.\n");
67
68          return 0;
69  }
70
71
72  int main( int argc, const char* argv[] )
73  {
74          // Connect the different signals (that the daemon
                can receive) to signal_handler_ function.
75          signal(SIGINT, signal_handler_);
76          signal(SIGTERM, signal_handler_);
77
78          if (hlog_init(DAEMON_LOG) < 0) {
79                  printf("Error. Could not init log file.
                        Daemon is stopping.\n");
80                  exit(EXIT_FAILURE);
81          }
82
83          // Init the daemon.
84      //      if (init_daemon_() < 0)
85        //              exit(EXIT_FAILURE);
86
87          // Initialize a dynamic memory module with 256 lines
                .
88          //dmemInit(256);
89
90          static pthread_t scheduler_thread;
91          static pthread_t placer_thread;
92          static pthread_t message_server_thread;
93          static pthread_t timer_thread;
94
95          // Takes care of scheduling processes.
96          pthread_create(&scheduler_thread, NULL, hdsched_main
                , (void*)NULL);
97          // Takes care of placing processes on the hardware.
98          pthread_create(&placer_thread, NULL, hdplacer_main,
                (void*)NULL);
99          // Generates timer interrupts.
100         pthread_create(&timer_thread, NULL, hdtimer_main, (
                void*)NULL);
101         // Takes care of receiving and processing messages
                from client programs.
102         // This thread is also the master thread and
                responsible for exiting all other threads.
103         pthread_create(&message_server_thread, NULL,
                hdmsg_main, (void*)NULL);
104
105         // Wait for each thread to finish before exiting the
                 daemon.
106         pthread_join(timer_thread, NULL);
107         pthread_join(scheduler_thread, NULL);
```

```
108             pthread_join(placer_thread, NULL);
109             pthread_join(message_server_thread, NULL);
110
111             hlog_close();
112
113             exit(EXIT_SUCCESS);
114     }
```

Listing D.7: hdmsg.c

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <fcntl.h>
6  #include <errno.h>
7  #include <unistd.h>
8  #include <syslog.h>
9  #include <string.h>
10 #include <signal.h>
11 //! Message queue functions.
12 #include <sys/ipc.h>
13 #include <sys/msg.h>
14 //! Sleep, unused at the moment.
15 #include <unistd.h>
16 #include <pthread.h>
17
18 #include <hlog.h>
19 #include <hmsg.h>
20 #include <hmqueue.h>
21 #include <hmem.h>
22 // Not used.
23 //#include <devicelist.h>
24 #include <integer_functions.h>
25 #include <hstructures.h>
26 #include <hprocess.h>
27 #include <hevent.h>
28
29 #include "hdaemon.h"
30 #include "hdsched.h"
31 #include "hdmsg.h"
32 #include "hdplacer.h"
33 #include "hdtimer.h"
34
35 /*! \brief Process a received message.
36  *
37  *      @param req_msg The message to be processed.
38  *      @return The message to be sent back to client on
        success. NULL on failure.
39  */
40 static void* process_request_(void* req_msg);
41 static void cleanup_(int exit_code);
42 static int handle_event_();
43
44 //! The event handler for this thread.
45 static struct hevent* event_handler = NULL;
46
47
48 static int handle_event_()
49 {
50 enum hdmevent incoming_event = (enum hdmevent)hevent_wait(
        event_handler);
51
52 // An incoming event is handled here.
53 switch (incoming_event) {
54 case HDME_QUIT:
55     // Quit the thread.
56     printf("wait_event_: Exiting message server thread.\n");
```

```
57      cleanup_(EXIT_SUCCESS);
58      break;
59  }
60
61  return 0;
62  }
63
64
65  void* process_request_(void* req_msg)
66  {
67  hlog_write(HLOG_NORMAL, "process_request_: Processing
        message.\n");
68
69  // The response message to be returned.
70  void* resp_msg = NULL;
71
72  hlog_write(HLOG_DEBUG, "process_request_: command=");
73  hlog_write_integer(hmsg_get_command(req_msg));
74  hlog_write_text(".\n");
75
76  switch (hmsg_get_command(req_msg)) {
77  // GCC compiler demands brackets on this case.
78  case HMC_ALLOC: {
79  // Commented away. Not ready to be used.
80      /*
81      hlog_write(HLOG_NORMAL, "process_request_: HMC_ALLOC\n")
            ;
82      resp_msg = hmsg_create(HMT_CTRL);
83      int* base_addr = calloc(1, sizeof(*base_addr));
84      int alloc_ret = hmem_allocate(char_to_int(hmsg_get_data(
            req_msg), 0), base_addr);
85      if (alloc_ret < 0) {
86          hlog_write(HLOG_ERROR, "process_request_: Failed to
                allocate memory for HMC_ALLOC.\n");
87      } else {
88          hlog_write(HLOG_DEBUG, "process_request_: Allocated
                memory for HMC_ALLOC.\n");
89          hmsg_set_return(resp_msg, *base_addr);
90      }
91      free(base_addr);
92      base_addr = NULL;
93      alloc_ret = -1;
94      */
95      break;
96      }
97  case HMC_EXEC:
98          hlog_write(HLOG_NORMAL, "process_request_: HMC_EXEC\n"
                );
99          resp_msg = hmsg_create(HMT_CTRL);
100         // Not ready yet.
101         //hmem_instruction_write(0, hmsg_get_data(req_msg),
                hmsg_get_size(req_msg));
102         hmsg_set_return(resp_msg, HMR_OK);
103         break;
104 case HMC_FREE:
105         hlog_write(HLOG_NORMAL, "process_request_: HMC_FREE\n"
                );
106         resp_msg = hmsg_create(HMT_CTRL);
107         // Not ready yet.
108         //hmem_free(char_to_int(hmsg_get_data(req_msg), 0));
109         hmsg_set_return(resp_msg, HMR_OK);
110         break;
111 case HMC_LDDEV:
112         hlog_write(HLOG_NORMAL, "process_request_: HMC_LDDEV\n
                ");
113         resp_msg = hmsg_create(HMT_CTRL);
114         hmsg_set_return(resp_msg, HMR_OK);
```

```
115         break;
116 case HMC_RMDEV:
117         hlog_write(HLOG_NORMAL, "process_request_: HMC_RMDEV\n");
118         resp_msg = hmsg_create(HMT_CTRL);
119         hmsg_set_return(resp_msg, HMR_OK);
120         break;
121 case HMC_RMQUE: {
122     // Connect to an existing queue.
123     // Note: If the queue is registered already, the old queue will be removed
124     // and the new queue registered instead.
125     hlog_write(HLOG_NORMAL, "process_request_: HMC_RMQUE\n");
126     resp_msg = hmsg_create(HMT_CTRL);
127     // Connect to an existing queue. Use sender's identifier as key.
128     struct hmqueue* reg_mqueue = hmqueue_add(hmsg_get_sender(req_msg), HMQ_CONNECT);
129     if (reg_mqueue == NULL) {
130         hlog_write(HLOG_ERROR, "process_request_: Could not connect to message queue.\n");
131         hmsg_set_return(resp_msg, HMR_ERROR);
132     } else {
133         hlog_write(HLOG_DEBUG, "process_request_: Connected to message queue. msqid_req=");
134         hlog_write_integer(hmqueue_get_id(reg_mqueue));
135         hlog_write_text(". key=");
136         hlog_write_integer(hmqueue_get_key(reg_mqueue));
137         hlog_write_text(".\n");
138         hmsg_set_return(resp_msg, HMR_OK);
139     }
140     reg_mqueue = NULL;
141     break;
142 }
143 case HMC_READ:
144     hlog_write(HLOG_NORMAL, "process_request_: HMC_READ\n");
145     resp_msg = hmsg_create(HMT_CTRLMEM);
146     // Not ready yet.
147     //hmem_data_read(0, hmsg_get_data(resp_msg), hmsg_get_address(req_msg), hmsg_get_size(req_msg));
148     hmsg_set_return(resp_msg, HMR_OK);
149     break;
150 case HMC_UMQUE:
151     hlog_write(HLOG_NORMAL, "process_request_: HMC_UMQUE\n");
152     hmqueue_remove_by_key(hmsg_get_sender(req_msg));
153     break;
154 case HMC_WRITE:
155     hlog_write(HLOG_NORMAL, "process_request_: HMC_WRITE\n");
156     resp_msg = hmsg_create(HMT_CTRL);
157     // Not ready yet.
158     //hmem_data_write(0, hmsg_get_data(req_msg), hmsg_get_address(req_msg), hmsg_get_size(req_msg));
159     hmsg_set_return(resp_msg, HMR_OK);
160     break;
161 case HMC_REGPROC:
162     // Register a new process and wake up scheduler.
163     hlog_write(HLOG_NORMAL, "process_request_: HMC_REGPROC\n");
164     resp_msg = hmsg_create(HMT_CTRL);
165     struct hprocess* process_new = hprocess_create(hmsg_get_nice(req_msg));
166     hlog_write(HLOG_DEBUG, "process_request_: Filename in msg: ");
167     hlog_write_text(hmsg_get_bitfilename(req_msg));
168     hlog_write_text("\n");
```

```
169     hprocess_set_bitfilename( process_new,
            hmsg_get_bitfilename(req_msg) );
170     if (process_new == NULL) {
171         hmsg_set_return(resp_msg, HMR_NOPID);
172     } else {
173         hprocess_set_bitfilename(process_new,
                hmsg_get_bitfilename(req_msg));
174         hmsg_set_return(resp_msg, HMR_OK);
175         hdsched_add_new_process(process_new);
176         hdsched_notify(HDSE_NEW_PROCESS);
177     }
178     break;
179 default:
180     break;
181 }
182
183 return resp_msg;
184 }
185
186
187 static void cleanup_(int exit_code)
188 {
189   if (exit_code == EXIT_FAILURE) {
190       hlog_write(HLOG_ERROR, "HWOS message server terminated
            with error=");
191       hlog_write_integer(errno);
192       hlog_write_text(".\n");
193   }
194   else
195       hlog_write(HLOG_NORMAL, "HWOS message server
            terminated normally.\n");
196
197   // Quit the other threads also.
198   hdtimer_notify(HDTE_QUIT);
199   hdplacer_notify(HDPE_QUIT);
200   hdsched_notify(HDSE_QUIT);
201
202   hevent_remove(event_handler);
203   pthread_exit(NULL);
204 }
205
206
207 int hdmsg_notify(enum hdmevent event)
208 {
209         hevent_notify(event_handler, (int)event);
210
211         return 0;
212 }
213
214
215 int hdmsg_synchronize()
216 {
217         // Send dummy/sync message to message server
218         // (to make sure it handles event).
219         void* sync_msg = hmsg_create(HMT_CTRL);
220         hmsg_send(sync_msg, hmqueue_get(
                hmqueue_get_daemonkey()));
221         free(sync_msg);
222
223         return 0;
224 }
225
226
227 void* hdmsg_main()
228 {
229 // Init the message queue and return the identifier.
230 struct hmqueue* mqueue;
231 mqueue = hmqueue_add(hmqueue_get_daemonkey(), HMQ_CREATE);
```

```
232 |
233 | // Message to be received.
234 | void* req_msg;
235 |
236 | // Create event queue.
237 | event_handler = hevent_create(HEVENT_NONBLOCKING);
238 | if (event_handler == NULL) {
239 |     cleanup_(EXIT_FAILURE);
240 | }
241 |
242 | // Service loop for message server.
243 | while (1) {
244 |     // Blocking wait for a message on queue msqid. Put
|         message in req_msg.
245 |     hlog_write(HLOG_NORMAL, "hdmsg_main: Waiting for message
|         .\n");
246 |     req_msg = hmsg_receive(mqueue);
247 |
248 |     if (req_msg == NULL) {
249 |         hlog_write(HLOG_ERROR, "hdmsg_main: req_msg is NULL
|             .\n");
250 |         cleanup_(EXIT_FAILURE);
251 |     }
252 |
253 |     void* resp_msg;
254 |     // Process the received message and make ready the
|         response message.
255 |     resp_msg = process_request_(req_msg);
256 |
257 |     if (resp_msg != NULL) {
258 |         // Get the message queue for the sender and send the
|             response.
259 |         hmsg_send(resp_msg, hmqueue_get(hmsg_get_sender(
|             req_msg)));
260 |     }
261 |
262 |     // Handle any events from the other threads.
263 |     handle_event_();
264 | }
265 |
266 | cleanup_(EXIT_SUCCESS);
267 | }
```

Listing D.8: hdplacer.c

```
1  | #include <stdio.h>
2  | #include <stdlib.h>
3  | #include <errno.h>
4  | #include <unistd.h>
5  | #include <pthread.h>
6  |
7  | #include <hlog.h>
8  | #include <hstructures.h>
9  | #include <hplacer.h>
10 | #include <hprocess.h>
11 | #include <hmsg.h>
12 | #include <hmqueue.h>
13 | #include <hevent.h>
14 |
15 | #include "hdplacer.h"
16 | #include "hdmsg.h"
17 | #include "hdsched.h"
18 | #include "../platform_defines.h"
19 |
20 |
21 | static void cleanup_(int exit_code);
22 |
23 | /*! \brief Handle an incoming event.
```

```
24 |  *
25 |  *   This functions locks a mutex and waits on a
26 |  *   condition variable.
27 |  */
28 | static int wait_event_();
29 | static struct hprocess* get_current_process_();
30 | static int set_current_process_(struct hprocess* process);
31 |
32 | static struct hprocess* current_process = NULL;
33 | //! The event handler for this thread.
34 | static struct hevent* event_handler = NULL;
35 | //! If this is true, this thread should ask the server to
   |     quit all threads when exiting.
36 | static int notify_server_quit = 1;
37 | static const char* bitfilespath = "../bitfiles";
38 | static const char* icapdevice = "/var/icap";
39 |
40 | static int wait_event_()
41 | {
42 |         enum hdpevent incoming_event = (enum hdmevent)
   |             hevent_wait(event_handler);
43 |
44 |         struct hprocess* next_process =
   |             hdsched_get_next_process();
45 |         struct hprocess* current_process =
   |             get_current_process_();
46 |         int replace_current = 0;
47 |
48 |         // An incoming event is handled here.
49 |         switch (incoming_event) {
50 |                 case HDPE_QUIT:
51 |                         // Do not ask server to quit (the
   |                             server asked this thread to quit)
   |                             .
52 |                         notify_server_quit = 0;
53 |                         // Quit the thread.
54 |                         cleanup_(EXIT_SUCCESS);
55 |                         break;
56 |                 case HDPE_TIMER:
57 |                         // Timer interrupt.
58 |                         if (current_process != next_process)
59 |                                 replace_current = 1;
60 |                         break;
61 |         }
62 |
63 |         // Replace process if current and next are unequal.
64 |         if (replace_current) {
65 |                 printf("hdplacer_main: Replacing process.
   |                     old-pid=%d. new-pid=%d\n",
   |                     hprocess_get_pid(current_process),
   |                     hprocess_get_pid(next_process));
66 |                 // Not ready yet.
67 |                 //hplacer_interrupt_process(current_process)
   |                     ;
68 |                 //hplacer_load_process(next_process);
69 |                 set_current_process_(next_process);
70 |                 hdsched_reschedule(current_process,
   |                     HDSE_RESCHED_TIMER);
71 |         } else
72 |                 printf("hdplacer_main: Keeping current
   |                     process on FPGA. pid=%d\n",
   |                     hprocess_get_pid(next_process));
73 |
74 |         return 0;
75 | }
76 |
77 |
78 | static struct hprocess* get_current_process_()
```

```
79 {
80         return current_process;
81 }
82
83
84 static int set_current_process_(struct hprocess* process)
85 {
86         current_process = process;
87
88         return 0;
89 }
90
91
92 static void cleanup_(int exit_code)
93 {
94         // Only notify server if not done already.
95         if (notify_server_quit) {
96                 notify_server_quit = 0;
97                 hdmsg_notify(HDME_QUIT);
98         }
99
100        hdmsg_synchronize();
101        pthread_exit(NULL);
102 }
103
104
105 int hdplacer_notify(enum hdpevent event)
106 {
107        hevent_notify(event_handler, (int)event);
108
109        return 0;
110 }
111
112
113 void* hdplacer_main()
114 {
115        // Create event handler.
116        event_handler = hevent_create(HEVENT_BLOCKING);
117        if (event_handler == NULL) {
118                notify_server_quit = 0;
119                cleanup_(EXIT_FAILURE);
120        }
121
122        hplacer_set_bitfilespath(bitfilespath);
123        hplacer_set_icapdevice(icapdevice);
124
125        // Sleep a bit to initialize scheduler.
126        sleep(1);
127        // Service loop for placer.
128        while (1) {
129                // Ask scheduler to find a new process to
                       place.
130                hdsched_notify(HDSE_SHORT);
131                wait_event_();
132        }
133
134        hevent_remove(event_handler);
135        cleanup_(EXIT_SUCCESS);
136 }
```

Listing D.9: hdsched.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <errno.h>
4 #include <sys/types.h>
5 #include <sys/stat.h>
6 #include <unistd.h>
```

```
7  #include <pthread.h>
8
9  #include <hlog.h>
10 #include <hstructures.h>
11 #include <hprocess.h>
12 #include <hsqueue.h>
13 #include <hsqlist.h>
14 #include <hplacer.h>
15 #include <hlist.h>
16 #include <hevent.h>
17
18 #include "hdsched.h"
19 #include "hdmsg.h"
20
21
22 static void cleanup_(int exit_code);
23
24 //! Suspend the thread until a new event occurs.
25 static int wait_event_();
26
27 /*! \brief Performs the (long-term) scheduling of a new
       process.
28  *
29  *  This function should decide if the given process should
        be accepted
30  *  by the system or not.
31  *
32  *  @param process Points to the first process in the
        HPS_NEW-queue.
33  *
34  *  @return Negative if process not accepted. 0 or positive
        value if it was.
35  *  @retval HDSR_TOO_MANY Too many processes in system.
36  *  @retval HDSR_NO_BITFILE Bitfile does not exists.
37  *  @retval HDSR_READY Process added to a ready queue.
38  */
39 static int schedule_longterm_(struct hprocess* process);
40 static int schedule_shortterm_();
41 static int increment_process_number_();
42 static int decrement_process_number_();
43
44 /*! \brief Get first queue with given priority or create a
       queue and insert it at correct place in list.
45  *
46  *  @param list The list of queues.
47  *  @param priority The priority to be compared.
48  *  @return The exisiting queue or a new queue (at correct
        place in list) on success. NULL on failure.
49  */
50 static struct hsqueue* create_queue_by_priority_(int list,
       int priority);
51
52 /*! \brief Set next process to be placed on the FPGA.
53  *
54  *  The process can be fetched by another thread at the same
55  *  time and is therefore secured with a mutex.
56  *
57  *  @param process The process.
58  *  @return 0 on success. Negative on failure.
59  */
60 int set_next_process_(struct hprocess* process);
61
62
63 //! Mutex for access to the queue of new processes.
64 static pthread_mutex_t new_hsqueue_mutex =
       PTHREAD_MUTEX_INITIALIZER;
65 //! Ask the server to quit all threads.
66 static int notify_server_quit = 1;
```

```
67 //! Number of (accepted) processes in the system and it's
       mutex.
68 static int processes_num = 0;
69 static pthread_mutex_t processes_num_mutex =
       PTHREAD_MUTEX_INITIALIZER;
70 //! Next process to be placed on the FPGA and it's mutex.
71 static struct hprocess* next_process = NULL;
72 static pthread_mutex_t next_process_mutex =
       PTHREAD_MUTEX_INITIALIZER;
73 //! Queue for processes that need rescheduling and it's
       mutex.
74 static struct hlist* resched_queue = NULL;
75 static pthread_mutex_t resched_queue_mutex =
       PTHREAD_MUTEX_INITIALIZER;
76 //! Mutex for HPS_READY-queue.
77 static pthread_mutex_t ready_queue_mutex =
       PTHREAD_MUTEX_INITIALIZER;
78 //! The event handler for this thread.
79 static struct hevent* event_handler = NULL;
80
81
82 static int increment_process_number_()
83 {
84         int ret = -1;
85         pthread_mutex_lock( &processes_num_mutex );
86         processes_num += 1;
87         ret = processes_num;
88         pthread_mutex_unlock( &processes_num_mutex );
89
90         return ret;
91 }
92
93
94 static int decrement_process_number_()
95 {
96         int ret = -1;
97         pthread_mutex_lock( &processes_num_mutex );
98         if (processes_num < 1)
99                 processes_num = 0;
100        else
101                processes_num -= 1;
102
103        ret = processes_num;
104        pthread_mutex_lock( &processes_num_mutex );
105
106        return ret;
107 }
108
109
110 static void cleanup_(int exit_code)
111 {
112        // Only notify server if not done already.
113        if (notify_server_quit) {
114                notify_server_quit = 0;
115                hdmsg_notify(HDME_QUIT);
116        }
117
118        hdmsg_synchronize();
119        pthread_exit(NULL);
120 }
121
122
123 static struct hsqueue* create_queue_by_priority_(int list,
       int priority)
124 {
125 struct hsqueue* queue = hsqlist_get_last_queue(list);
126
127 // Walk through the queues from the last (lowest priority)
       to
```

```
128 // the first (highest priority).
129 // Return the first queue that has larger or equal priority to
130 // the given priority.
131 while (hsqueue_get_prev(queue) != NULL) {
132     if ( hprocess_get_priority ( hsqueue_get_first_process(queue) ) >= priority)
133         return queue;
134     queue = hsqueue_get_prev(queue);
135 }
136
137 if (hprocess_get_priority( hsqueue_get_first_process(queue) ) < hprocess_base_priority()) {
138     // All queues have less priority (or there is no queues).
139     // Create new queue and put it in front of list.
140     queue = hsqueue_create();
141     hsqlist_insert_queue_before(HPS_READY, hsqlist_get_first_queue(HPS_READY), queue);
142 } else if (hprocess_get_priority( hsqueue_get_first_process(queue) ) > hprocess_base_priority()) {
143     // The queue found have larger priority than the given priority.
144     // Create new queue and put it behind that queue.
145     struct hsqueue* new_queue = hsqueue_create();
146     if (hsqueue_get_next(queue) == NULL)
147         hsqlist_add_queue(HPS_READY, new_queue);
148     else
149         hsqlist_insert_queue_before(HPS_READY, hsqueue_get_next(queue), new_queue);
150     queue = new_queue;
151 } // Else: The queue found has the given priority.
152
153 return queue;
154 }
155
156
157 static int schedule_shortterm_()
158 {
159         // Get the HPS_READY-queue with highest priority.
160         pthread_mutex_lock(&ready_queue_mutex);
161         struct hsqueue* ready_queue = hsqlist_get_first_queue(HPS_READY);
162         struct hprocess* process = hsqueue_dequeue_process(ready_queue);
163         pthread_mutex_unlock(&ready_queue_mutex);
164
165         if (process != NULL) {
166                 set_next_process_(process);
167         }
168
169         return 0;
170 }
171
172
173 static int schedule_longterm_(struct hprocess* process)
174 {
175         if ( hdsched_processes_number() + 1 > hprocess_max_processes() )
176             return HDSR_TOO_MANY;
177
178         // Check if bitfile is regular file.
179         struct stat statbuffer;
180         char* filename_full = hplacer_create_full_bitfilename(process);
181         hlog_write(HLOG_DEBUG, "schedule_longterm_: Filename for process: ");
182         hlog_write_text(filename_full);
```

```
183             hlog_write_text("\n");
184
185             stat(filename_full, &statbuffer);
186             free(filename_full);
187             if ( !S_ISREG(statbuffer.st_mode) )
188                     return HDSR_NO_BITFILE;
189
190             // Process can be added to the HPS_READY-queue with
                    the base
191             // priority.
192             hprocess_set_priority(process,
                    hprocess_base_priority());
193             pthread_mutex_lock(&ready_queue_mutex);
194             struct hsqueue* queue = create_queue_by_priority_(
                    HPS_READY, hprocess_base_priority() );
195             hsqueue_enqueue_process(queue, process);
196             pthread_mutex_unlock(&ready_queue_mutex);
197             increment_process_number_();
198
199             return HDSR_READY;
200 }
201
202
203 static int wait_event_()
204 {
205 enum hdsevent incoming_event = (enum hdsevent)hevent_wait(
        event_handler);
206
207 // An incoming event is handled here.
208 switch (incoming_event) {
209 case HDSE_QUIT: {
210 // Do not ask server to quit (the server asked this thread
        to quit).
211 notify_server_quit = 0;
212 cleanup_(EXIT_FAILURE);
213 break;
214 }
215 case HDSE_NEW_PROCESS: {
216       struct hprocess* new_process = hdsched_get_new_process
            ();
217       int sched_long = schedule_longterm_(new_process);
218       hlog_write(HLOG_DEBUG, "wait_event_: (scheduler) A new
            process has been added.\n");
219       hlog_write(HLOG_DEBUG, "Return value from long-term
            scheduler: ");
220       hlog_write_integer(sched_long);
221       hlog_write_text(".\n");
222
223       // Scheduler refuses the new process.
224       // TODO: Should notify the message server and let it
            send a response to the client.
225       if (sched_long < 0)
226           hprocess_remove(new_process);
227
228       break;
229 }
230 case HDSE_SHORT:
231 schedule_shortterm_();
232 break;
233 case HDSE_RESCHED_TIMER: {
234       struct hprocess* resched_process = hlist_dequeue(
            resched_queue);
235       if (resched_process != NULL) {
236 struct hsqueue* ready_queue =
237   create_queue_by_priority_(HPS_READY, hprocess_get_priority
        (resched_process));
```

```
238             hsqueue_enqueue_process(ready_queue,
                    resched_process);
239         }
240         break;
241     }
242 default:
243     break;
244 }
245
246 return 0;
247 }
248
249
250 int set_next_process_(struct hprocess* process)
251 {
252         // Lock the mutex to get a stable value of the
                process.
253         pthread_mutex_lock( &next_process_mutex );
254         next_process = process;
255         pthread_mutex_unlock( &next_process_mutex );
256
257         return 0;
258 }
259
260
261 int hdsched_reschedule(struct hprocess* process, enum
        hdsevent event)
262 {
263         pthread_mutex_lock( &resched_queue_mutex );
264         hlist_enqueue(resched_queue, process);
265         pthread_mutex_unlock( &resched_queue_mutex );
266
267         printf("Rescheduled process. pid=%d\n",
                hprocess_get_pid(process));
268
269         hdsched_notify(event);
270
271         return 0;
272 }
273
274
275 int hdsched_add_new_process(struct hprocess* process)
276 {
277         //! TODO: This function always adds the new process
                to the first of the HPS_NEW-queues.
278         //! A future version of the scheduler might want to
                add it to another queue based on
279         //! the process' initial priority.
280         pthread_mutex_lock( &new_hsqueue_mutex );
281         hsqueue_enqueue_process(hsqlist_get_first_queue(
                HPS_NEW), process);
282         pthread_mutex_unlock( &new_hsqueue_mutex );
283
284         return 0;
285 }
286
287
288 struct hprocess* hdsched_get_new_process()
289 {
290         struct hprocess* new_process = NULL;
291
292         pthread_mutex_lock( &new_hsqueue_mutex );
293         new_process = hsqueue_dequeue_process(
                hsqlist_get_first_queue(HPS_NEW) );
294         pthread_mutex_unlock( &new_hsqueue_mutex );
295
296         return new_process;
297 }
298
```

```
299 |
300 | struct hprocess* hdsched_get_next_process()
301 | {
302 |         struct hprocess* ret = NULL;
303 |         // Lock the mutex to get a stable value of the
    |             process.
304 |         pthread_mutex_lock( &next_process_mutex );
305 |         ret = next_process;
306 |         pthread_mutex_unlock( &next_process_mutex );
307 |
308 |         return ret;
309 | }
310 |
311 |
312 | int hdsched_processes_number()
313 | {
314 |         int ret = -1;
315 |         pthread_mutex_lock( &processes_num_mutex );
316 |         ret = processes_num;
317 |         pthread_mutex_unlock( &processes_num_mutex );
318 |
319 |         return ret;
320 | }
321 |
322 |
323 | int hdsched_notify(enum hdsevent event)
324 | {
325 |         hevent_notify(event_handler, (int)event);
326 |
327 |         return 0;
328 | }
329 |
330 |
331 | void* hdsched_main()
332 | {
333 |         hsqlist_create(HPS_READY);
334 |         struct hsqueue* new_hsqueue = hsqueue_create();
335 |         hsqlist_create(HPS_NEW);
336 |         hsqlist_add_queue(HPS_NEW, new_hsqueue);
337 |
338 |         // Create event handler.
339 |         event_handler = hevent_create(HEVENT_BLOCKING);
340 |         if (event_handler == NULL) {
341 |                 notify_server_quit = 0;
342 |                 cleanup_(EXIT_FAILURE);
343 |         }
344 |
345 |         // Create queue for processes that need rescheduling
    |             .
346 |         resched_queue = hlist_create();
347 |
348 |         // Service loop for scheduler.
349 |         while (1) {
350 |                 printf("hdsched_main:_Waiting\n");
351 |                 // Suspend the scheduler until a new event.
352 |                 wait_event_();
353 |         }
354 |
355 |
356 |         hevent_remove(event_handler);
357 |         cleanup_(EXIT_SUCCESS);
358 | }
```

Listing D.10: hdtimer.c

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <pthread.h>
```

```
4  #include <unistd.h>
5
6  #include <hevent.h>
7
8  #include "hdplacer.h"
9  #include "hdmsg.h"
10 #include "hdtimer.h"
11
12
13 static void cleanup_(int exit_code);
14
15 /*! \brief Handle an incoming event.
16  *
17  *  This functions locks a mutex, but does not wait on a
18  *  condition variable (events are handled synchronously).
19  */
20 static int handle_event_();
21
22 //! The event handler for this thread.
23 static struct hevent* event_handler = NULL;
24
25 //! If this is true, this thread should ask the server to
       quit all threads when exiting.
26 static int notify_server_quit = 1;
27
28
29 static int handle_event_()
30 {
31         enum hdtevent incoming_event = (enum hdtevent)
              hevent_wait(event_handler);
32
33         // An incoming event is handled here.
34         switch (incoming_event) {
35                 case HDTE_QUIT:
36                         // Do not ask server to quit (the
                             server asked this thread to quit)
                             .
37                         notify_server_quit = 0;
38                         // Quit the thread.
39                         cleanup_(EXIT_SUCCESS);
40                         break;
41         }
42
43         return 0;
44 }
45
46
47 static void cleanup_(int exit_code)
48 {
49         // Only notify server if not done already.
50         if (notify_server_quit) {
51                 notify_server_quit = 0;
52                 hdmsg_notify(HDME_QUIT);
53         }
54
55         hdmsg_synchronize();
56         pthread_exit(NULL);
57 }
58
59
60 int hdtimer_notify(enum hdtevent event)
61 {
62         hevent_notify(event_handler, (int)event);
63
64         return 0;
65 }
66
67 const int hdtimer_get_timeslice()
68 {
```

```
69          return 1000000;
70  }
71
72
73  void* hdtimer_main()
74  {
75          // Create event handler.
76          event_handler = hevent_create(HEVENT_NONBLOCKING);
77          if (event_handler == NULL) {
78                  notify_server_quit = 0;
79                  cleanup_(EXIT_FAILURE);
80          }
81
82          // Sleep a bit to initialize scheduler.
83          sleep(1);
84          // Service loop for placer.
85          while (1) {
86                  // Wait for one timeslice.
87                  usleep(hdtimer_get_timeslice());
88                  // Notify placer that timer has expired.
89                  hdplacer_notify(HDPE_TIMER);
90                  // Handle an incoming event.
91                  handle_event_();
92          }
93
94          hevent_remove(event_handler);
95          cleanup_(EXIT_SUCCESS);
96  }
```

D.2 HWOS-library

These are some of the selected header files (*.h) from the HWOS library.

Listing D.11: hdev.h

```
1   /*! \file hdev.h
2    *  \brief [Hardware OS Device Driver Interface] Keep track
         of device drivers in the HWOS.
3    *
4    *  This functionality is not really used in this version of
         the HWOS.
5    *  The module can function as a base for further
         development. It is probably a good idea
6    *  to perform the actual creation of a device driver in
         hdev_add (now it only adds a
7    *  description to a list).
8    *
9    * Based on devicelist by Vegard Endresen.
10   * Sindre Hansen (2011): Rewritten.
11   *      - Changed names and public interface.
12   *      - Hidden the device structure.
13   *      - Encapsulated variables and data structures.
14   */
15
16
17  #include "hstructures.h"
18
19  #ifndef HDEV_H
20  #define HDEV_H
21
22  //! Longest allowed path for device file.
23  #define HDEV_PATH_SIZE 40
24
25  /*! \brief Size of the device list.
```

```
26 |  *
27 |  *  @return Number of elements in the device list.
28 |  */
29 | int hdev_size();
30 |
31 | /*! \brief Get device by major number.
32 |  *
33 |  *  @return Number of elements in the device list.
34 |  */
35 | struct hdev* hdev_get_by_major(int major);
36 |
37 | /*! \brief Add a device description.
38 |  *
39 |  *  This adds a description of a character device
40 |  *  with a path and a major number.
41 |  *
42 |  *  @param major The major number of the device.
43 |  *  @param device_path Path of the device.
44 |  *  @return A pointer to a struct representing the device on
|       success. NULL on failure.
45 |  */
46 | struct hdev* hdev_add(int major, char* device_path);
47 |
48 | /*! \brief Remove a device from the HWOS.
49 |  *
50 |  *  @param device Pointer to the struct representing the
|       device.
51 |  *  @return 0 on success. Negative on failure.
52 |  */
53 | int hdev_remove(struct hdev* device);
54 |
55 | /*! \brief Get major number for device.
56 |  *
57 |  *  @param device Pointer to the device.
58 |  *  @return Major number (positive integer) on success.
|       Negative on failure.
59 |  */
60 | int hdev_get_major(struct hdev* device);
61 |
62 | /*! \brief Remove device by major number.
63 |  *
64 |  *  @param major Major number.
65 |  *  @return 0 on success. Negative on failure.
66 |  */
67 | int hdev_remove_by_major(int major);
68 |
69 | /*! \brief Remove all devices in the system.
70 |  *
71 |  *  @return 0 on success. Negative on failure.
72 |  */
73 | int hdev_remove_all();
74 |
75 | /*! \brief Get first device in list.
76 |  *
77 |  *  @return Pointer to the device on success. NULL on
|       failure.
78 |  */
79 | struct hdev* hdev_get_first();
80 |
81 | /*! \brief Get last device in list.
82 |  *
83 |  *  @return Pointer to the device on success. NULL on
|       failure.
84 |  */
85 | struct hdev* hdev_get_last();
86 |
87 | /*! \brief Get next device in list.
88 |  *
89 |  *  @param device Pointer to the device that has the next
```

```
     field.
90   *   @return Pointer to the device on success. NULL on
         failure.
91   */
92  struct hdev* hdev_get_next(struct hdev* device);
93
94  /*! \brief Get previous device in list.
95   *
96   *   @param device Pointer to the device that has the
         previous field.
97   *   @return Pointer to the device on success. NULL on
         failure.
98   */
99  struct hdev* hdev_get_prev(struct hdev* device);
100
101
102 #endif
```

Listing D.12: hevent.h

```
1   /*! \file hevent.h
2    *  \brief [Hardware OS Event Interface] Notify and handle
         events between threads.
3    *
4    *  This module contains functionality for handling events
         between threads.
5    *  Events can be handled both blocking (asynchronously) or
         non-blocking (synchronously).
6    *  The underlying implemenation use System V message queues
         .
7    *
8    *  Author (2011): Sindre Hansen
9    *
10   */
11
12  #ifndef HEVENT_H
13  #define HEVENT_H
14
15  #define HEVENT_BLOCKING 1
16  #define HEVENT_NONBLOCKING 0
17
18  #include "hstructures.h"
19
20  /*! \brief Creates an event handler.
21   *
22   *  @param async Set to true if hevent_wait should block
         while waiting for events.
23   *  @return The event handler object.
24   */
25  struct hevent* hevent_create(int blocking);
26
27  /*! \brief Notify an event.
28   *
29   *  @param event Event handler.
30   *  @param notification The actual event/notification. Must
         be an integer.
31   *  @return 0 on success. Negative on failure.
32   */
33  int hevent_notify(struct hevent* event, int notification);
34
35  /*! \brief Wait on event.
36   *
37   *  Wait may be performed in a blocking or
38   *  non-blocking manner (see hevent_create).
39   *
40   *  @param event Event handler.
41   *  @return The event on success. -1 on failure.
42   */
43  int hevent_wait(struct hevent* event);
```

```
44 |
45 | /*! \brief Remove an event handler.
46 |  *
47 |  *  @param event Event handler.
48 |  *  @return 0 on success. -1 on failure.
49 |  */
50 | int hevent_remove(struct hevent* event);
51 |
52 |
53 | #endif
```

Listing D.13: hicap.h

```
1  | /*! \file hicap.h
2  |  *  \brief [Internal Configuration Port Interface] Interface
       for partial reconfiguration on the FPGA.
3  |  *
4  |  *  This module makes it possible to write a partial
       bitstream to the ICAP on the FPGA. Based on
5  |  *  the important work done by Sverre Hamre in his thesis.
6  |  *
7  |  *  Original author (2009): Sverre Hamre
8  |  *  Modified (2011): Sindre Hansen
9  |  *      - Customized interface for the HWOS.
10 |  *      - Added conversion from CLB column to absolute
       column.
11 |  *
12 |  */
13 |
14 | #ifndef HICAP_H
15 | #define HICAP_H
16 |
17 | //! Number of CLB columns on the XC4VFX12 (UG070, page 184).
18 | #define HC_CLBCOLS_XC4VFX12             24
19 | /*! Number of CLB minor rows on the XC4VFX12 (UG070, page
       184).
20 |  *  A minor row is defined here as a row that is 1 CLB in
       vertical width and
21 |  *  spanning the entire device horizontally.
22 |  */
23 | #define HC_CLBMINORROWS_XC4VFX12        64
24 |
25 |
26 | // TODO: These defines should be given the HC_-prefix (
       Sindre Hansen).
27 |
28 | #define DUMMY_PACKET                0xFFFFFFFFUL
29 | #define SYNC_PACKET                 0xAA995566UL
30 | #define NOP_PACKET                  0X20000000UL
31 | #define CMD_PACKET                  0x30008001UL
32 | #define FAR_PACKET                  0x30002001UL
33 | #define IDCODE_PACKET               0x30018001UL
34 | #define STAT_PACKET                 0x2800E001UL
35 |
36 |
37 | #define WCFG_CMD                    1     // Write configuration
       comand.
38 | #define LFRM_CMD                    3     // Write last frame
       command.
39 | #define RCFG_CMD                    4
40 | #define RCRC_CMD                    7
41 | #define DESYNC_CMD                  14
42 |
43 |
44 | /* Configuration registers */
45 | #define FDRI                        2    //Frame data register
       input
46 | #define STAT                        7    //Status register
47 |
```

```
48 /* Constant to use for CRC check when CRC has been disabled */
49 #define XHI_DISABLED_AUTO_CRC          0x0000DEFCUL
50 #define XHI_TYPE_1_PACKET_MAX_WORDS    1024
51
52 #define WORDS_PR_FRAME                 41
53
54 #define TYPE2                          30  //Bitposition for type 2 header
55 #define TYPE1                          29  //Bitposition for type 1 header
56
57 #define REG_ADDR                       13  //Bitposition for register address in type 1 header
58
59 #define OP_READ                        27  //Bitposition for read op
60 #define OP_WRITE                       28  //Bitposition for write op
61
62 /* Device specific ID code */
63 #define V4FX12ID                       0x21E58093;  // Value read out from the virtex fx12 used.
64
65
66 /*! \brief Open the ICAP-device.
67  *
68  *  @param icap_device Path to the device in the system. Example: /dev/icap
69  *  @param device_type Type of the Virtex-4 device. Only "XC4VFX12" is supported at this time.
70  *  @return A file descriptor for the device. -1 on failure.
71  */
72 int hicap_open(char* icap_device, char* device_type);
73
74 /*! \brief Close the ICAP-device.
75  *
76  *  @param icap_handle The handle/file descriptor to the device as returned by hicap_open.
77  *  @return 0 on success. Negative on failure
78  */
79 int hicap_close(int icap_handle);
80
81 /*! \brief Write a bitstream to ICAP.
82  *
83  *  Each row spans the entire FPGA horizontally and is 16 CLBs in vertical width.
84  *  The real row address start from 0 at the top of the device and increments downwards.
85  *
86  *  @param icap_handle The handle/file descriptor to the device as returned by hicap_open.
87  *  @param infilename The filename for the bitfile.
88  *  @param real_row Specify a row from 0 to [max rows - 1]. See definition of row in description.
89  *  @param clb_column Specify a CLB column from 0 to [max CLB columns - 1] to start from.
90  *  @param frames_offset Specify an offset of frames within the given address (normally 0).
91  *  @param frames The number of frames to be written. 22 frames are 8 CLBs + 1 HCLK + 8 CLBs.
92  *  @return 0 on success. Negative on failure
93  */
94 int hicap_write(int icap_handle, char* infilename, int real_row, int clb_column, int frames_offset, int frames);
95
96 /*! \brief Get status of the ICAP-device.
97  *
```

```
98  *  Refer to the work of Sverre Hamre to find out what this
        function should
99  *  return.
100 *
101 *  @param icap_handle Handle for icap device. Returned by
        hicap_open.
102 *  @return 0 on failure. Undefined on success?
103 */
104 int hicap_get_status(int icap_handle);
105
106
107 #endif
```

Listing D.14: hlist.h

```
1   /*! \file hlist.h
2    *  \brief [Hardware OS Double Linked List Interface]
        General functionality for double linked lists.
3    *
4    *  General purpose double linked list structure. Each list
        (hlist) has elements (hlelement).
5    *  Each element has data pointed to by void-pointers.
6    *
7    *  Author (2011): Sindre Hansen
8    *
9    */
10
11  #ifndef HLIST_H
12  #define HLIST_H
13
14  #include "hstructures.h"
15
16  /*! \brief Get size of elements in list.
17   *
18   *  @param list Pointer to list.
19   *  @return Number of elements in list.
20   */
21  int hlist_size(struct hlist* list);
22
23  /*! \brief Get data for the given element.
24   *
25   *  @param element Pointer to element.
26   *  @return Pointer to the element's data.
27   */
28  void* hlelement_get_data(struct hlelement* element);
29
30  /*! \brief Set data for the given element.
31   *
32   *  @param element Pointer to element.
33   *  @param data Pointer to the data.
34   *  @return 0 on success. Negative on failure.
35   */
36  int hlelement_set_data(struct hlelement* element, void* data
        );
37
38  /*! \brief Get previous element in list.
39   *
40   *  @param element Pointer to element that has the previous
        field.
41   *  @return Pointer to the previous element on success. NULL
         on failure.
42   */
43  struct hlelement* hlelement_get_prev(struct hlelement*
        element);
44
45  /*! \brief Get next element in list.
46   *
47   *  @param element Pointer to element that has the next
        field.
```

```
48  *  @return Pointer to the next element on success. NULL on
        failure.
49  */
50 struct hlelement* hlelement_get_next(struct hlelement*
       element);
51
52 /*! \brief Create a new list.
53  *
54  *  @return Pointer to the new list structure on success.
        NULL on failure.
55  */
56 struct hlist* hlist_create();
57
58 /*! \brief Enqueue a new element in a list.
59  *
60  *  @param list Pointer to the given list.
61  *  @param data_element Pointer to the data that should be
        connected to the element.
62  *  @return Pointer to the new element on success. NULL on
        failure.
63  */
64 struct hlelement* hlist_enqueue(struct hlist* list, void*
       data_element);
65
66 /*! \brief Insert a new element after another in the given
       list.
67  *
68  *  @param list Pointer to the given list.
69  *  @param element Element that the new element should be
        placed after.
70  *  @param data_element Pointer to the data that should be
        connected to the element.
71  *  @return Pointer to the new element on success. NULL on
        failure.
72  */
73 struct hlelement* hlist_insert_after(struct hlist* list,
       struct hlelement* element, void* data_element);
74
75 /*! \brief Insert a new element before another in the given
       list.
76  *
77  *  @param list Pointer to the given list.
78  *  @param element Element that the new element should be
        placed before.
79  *  @param data_element Pointer to the data that should be
        connected to the element.
80  *  @return Pointer to the new element on success. NULL on
        failure.
81  */
82 struct hlelement* hlist_insert_before(struct hlist* list,
       struct hlelement* element, void* data_element);
83
84 /*! \brief Remove a given element from the given list.
85  *
86  *  @param list Pointer to the given list.
87  *  @param element Pointer to the given element.
88  *  @return Pointer to the elements data on success. NULL on
        failure.
89  */
90 void* hlelement_remove(struct hlelement* element, struct
       hlist* list);
91
92 /*! \brief Create an list element that does not belong to
       any list.
93  *
94  *  @param data_element Pointer to the data that should be
        connected to the element.
95  *  @return Pointer to the new element on success. NULL on
        failure.
```

```
96  */
97  struct hlorphan* hlorphan_create(void* data_element);
98
99  /*! \brief Get next element for the given element.
100  *
101  *  @param orphan Pointer to the element that has the next
         field.
102  *  @return Pointer to the element on success. NULL on
         failure.
103  */
104 struct hlorphan* hlorphan_get_next(struct hlorphan* orphan);
105
106 /*! \brief Set next field for the given element.
107  *
108  *  @param orphan Pointer to the element that has the next
         field.
109  *  @param next Pointer to the next element.
110  *  @return 0 on success. Negative on failure.
111  */
112 int hlorphan_set_next(struct hlorphan* orphan, struct
       hlorphan* next);
113
114 /*! \brief Get previous element for the given element.
115  *
116  *  @param orphan Pointer to the element that has the
         previous field.
117  *  @return Pointer to the element on success. NULL on
         failure.
118  */
119 struct hlorphan* hlorphan_get_prev(struct hlorphan* orphan);
120
121 /*! \brief Set previous field for the given element.
122  *
123  *  @param orphan Pointer to the element that has the
         previous field.
124  *  @param prev Pointer to the previous element.
125  *  @return 0 on success. Negative on failure.
126  */
127 int hlorphan_set_prev(struct hlorphan* orphan, struct
       hlorphan* prev);
128
129 /*! \brief Get pointer to data for the given element.
130  *
131  *  @param orphan Pointer to the element that has the data
         field.
132  *  @return Pointer to the data on success. NULL on failure.
133  */
134 void* hlorphan_get_data(struct hlorphan* orphan);
135
136 /*! \brief Remove the first element (dequeue) from the list.
137  *
138  *  @param list Pointer to the given list.
139  *  @return Pointer to the element's data on success. NULL
         on failure.
140  */
141 void* hlist_dequeue(struct hlist* list);
142
143 /*! \brief Remove the given list and it's elements.
144  *
145  *  Note that this will NOT remove the data elements
146  *  connected to each list element.
147  *
148  *  @param list Pointer to the given list.
149  *  @return 0 on success. Negative on failure.
150  */
151 int hlist_remove(struct hlist* list);
152
153 /*! \brief Get first element of the given list.
154  *
```

```
155  *  @param list Pointer to the given list.
156  *  @return Pointer to the first element on success. NULL on
         failure.
157  */
158 struct hlelement* hlist_get_first(struct hlist* list);
159
160 /*! \brief Get last element of the given list.
161  *
162  *  @param list Pointer to the given list.
163  *  @return Pointer to the last element on success. NULL on
         failure.
164  */
165 struct hlelement* hlist_get_last(struct hlist* list);
166
167 #endif
```

Listing D.15: hlog.h

```
1  /*! \file hevent.h
2  *  \brief [Hardware OS Event Interface] Notify and handle
       events between threads.
3  *
4  *  This module contains functionality for handling events
       between threads.
5  *  Events can be handled both blocking (asynchronously) or
       non-blocking (synchronously).
6  *  The underlying implemenation use System V message queues
       .
7  *
8  *  Author (2011): Sindre Hansen
9  *
10 */
11
12 #ifndef HEVENT_H
13 #define HEVENT_H
14
15 #define HEVENT_BLOCKING 1
16 #define HEVENT_NONBLOCKING 0
17
18 #include "hstructures.h"
19
20 /*! \brief Creates an event handler.
21  *
22  *  @param async Set to true if hevent_wait should block
       while waiting for events.
23  *  @return The event handler object.
24  */
25 struct hevent* hevent_create(int blocking);
26
27 /*! \brief Notify an event.
28  *
29  *  @param event Event handler.
30  *  @param notification The actual event/notification. Must
       be an integer.
31  *  @return 0 on success. Negative on failure.
32  */
33 int hevent_notify(struct hevent* event, int notification);
34
35 /*! \brief Wait on event.
36  *
37  *  Wait may be performed in a blocking or
38  *  non-blocking manner (see hevent_create).
39  *
40  *  @param event Event handler.
41  *  @return The event on success. -1 on failure.
42  */
43 int hevent_wait(struct hevent* event);
44
45 /*! \brief Remove an event handler.
```

```
46  *
47  *  @param event Event handler.
48  *  @return 0 on success. -1 on failure.
49  */
50 int hevent_remove(struct hevent* event);
51
52
53 #endif
```

Listing D.16: hmqueue.h

```
1  /*! \file hmqueue.h
2   *  \brief [Hardware OS Message Queue Interface]
        Functionality for keeping message queue identifiers.
3   *
4   *  This is simply a double linked list of System V message
        queues.
5   *  This module should probably be rewritten so it uses
        hlist internally.
6   *
7   *  Original author (2010): Vegard Endresen
8   *
9   *  Modified (2011): Sindre Hansen
10  *      - Changed naming of functions.
11  *      - Used more encapsulating.
12  *
13  */
14
15 #ifndef HMQUEUE_H
16 #define HMQUEUE_H
17
18 //! Types of modes when creating a message queue.
19 enum hmqueue_mode {
20          HMQ_CREATE,
21          HMQ_CONNECT
22 };
23
24 /*! \brief Get previous element in list.
25  *
26  *  @param element Pointer to element that has the previous
        field.
27  *  @return Pointer to the previous element on success. NULL
        on failure.
28  */
29 struct hmqueue* hmqueue_get_prev(struct hmqueue* element);
30
31 /*! \brief Get next element in list.
32  *
33  *  @param element Pointer to element that has the next
        field.
34  *  @return Pointer to the next element on success. NULL on
        failure.
35  */
36 struct hmqueue* hmqueue_get_next(struct hmqueue* element);
37
38 /*! \brief Get key of a given message queue.
39  *
40  *  @param element Pointer to list element that has the
        message queue.
41  *  @return The key (positive integer) on success. Negative
        on failure.
42  */
43 int hmqueue_get_key(struct hmqueue* element);
44
45 /*! \brief Get ID of message queue.
46  *
47  *  @param element Pointer to list element that has the
        message queue.
```

```
48  *  @return ID of the message queue on success. Negative on
        failure.
49  */
50  int hmqueue_get_id(struct hmqueue* element);
51
52  /*! \brief Get size of elements in message queue list.
53  *
54  *  @return Number of elements in list.
55  */
56  int hmqueue_size();
57
58  /*! \brief Get ID of message queue identified by key.
59  *
60  *  @param key A key that identifies the message queue.
61  *  @return ID of the message queue on success. Negative on
        failure.
62  */
63  int hmqueue_get_id_by_key(int key);
64
65  /*! \brief Get message queue element identified by key.
66  *
67  *  @param key A key that identifies the message queue.
68  *  @return List element on success. NULL on failure.
69  */
70  struct hmqueue* hmqueue_get(int key);
71
72  /*! \brief Adds a message queue and returns the queue
        identifier.
73  *
74  *  Based on "mode" this function either connects to an
        exisiting
75  *  message queue or creates an new message queue.
76  *
77  *  @param key Unique key for the new queue.
78  *  @param mode Mode for adding a message queue to the list.
79  *  @return The message queue struct for the new queue on
        success. NULL on failure.
80  */
81  struct hmqueue* hmqueue_add(int key, enum hmqueue_mode mode)
        ;
82
83  /*! \brief Remove the given list element.
84  *
85  *  @param element Pointer to list element that has the
        message queue.
86  *  @return 0 on success. Negative on failure.
87  */
88  int hmqueue_remove(struct hmqueue* element);
89
90  /*! \brief Remove the given message queue by key.
91  *
92  *  @param key Key that identifies the message queue.
93  *  @return 0 on success. Negative on failure.
94  */
95  int hmqueue_remove_by_key(int key);
96
97  /*! \brief Remove all message queues in list.
98  *
99  *  @return 0 on success. Negative on failure.
100 */
101 int hmqueue_remove_all();
102
103 /*! \brief Get first element in list.
104 *
105 *  @return The first list element.
106 */
107 struct hmqueue* hmqueue_get_first();
108
109 /*! \brief Get last element in list.
```

```
110  *
111  *  @return The last list element.
112  */
113 struct hmqueue* hmqueue_get_last();
114
115 /*! \brief Get the key of the message servers message queue.
116  *
117  *  @return The unique key of the message server.
118  */
119 int hmqueue_get_daemonkey();
120
121 #endif
```

Listing D.17: hmsg.h

```
1  /*! \file hmsg.h
2   *  \brief [Hardware OS Message Interface] Functionality for
        messages and message queues.
3   *
4   *  Defines an interface for message passing between the
        HWOS and a client application.
5   *  The message system is implemented using System V Message
        Queues.
6   *
7   *  Based on (2010): Vegard Endresen
8   *
9   *  Author (2011): Sindre Hansen
10  *      - Some commands in enum hmsg_command taken from work
        done by Vegard Endresen (2010).
11  */
12
13 #include "hstructures.h"
14
15 #ifndef HMSG_H
16 #define HMSG_H
17
18 //! Size of data for messages sent to HWOS.
19 #define HMSG_DATA_SIZE 128
20
21 //! Definition of message types.
22 enum hmsg_type {
23          HMT_ALL
24          ,HMT_CTRL
25          ,HMT_CTRLMEM
26          ,HMT_REGPROC
27 };
28
29
30 //! Definition of commands that can be sent through messages
        from a client application.
31 enum hmsg_command {
32          HMC_ALLOC                   //! Allocate BRAM memory.
33          ,HMC_EXEC                   //! Write program to
              instruction segment of BRAM.
34          ,HMC_FREE                   //! Free BRAM memory.
35          ,HMC_LDDEV                  //! Load device driver.
36          ,HMC_READ                   //! Read data from backend.
37          ,HMC_RMDEV                  //! Remove device driver.
38          ,HMC_RMQUE                  //! Register receive message
              queue to HWOS.
39          ,HMC_UMQUE                  //! Unregister receive
              message queue to HWOS.
40          ,HMC_WRITE                  //! Write data to backend.
41          ,HMC_SET_BITFILE            //! Set filename for
              bitstream file (deprecated?)
42          ,HMC_REGPROC                //! Register a process.
43 };
44
```

```
45
46 //! Definition of return values that can be sent back to a
       client application.
47 enum hmsg_return {
48           HMR_OK=0                    //! No errors.
49           ,HMR_ERROR=-1               //! Errors when
                 processing command.
50           ,HMR_NOPID=-2               //! No exists process
                 with the given PID.
51 };
52
53
54 /*! \brief Filename for FPGA bitstream.
55  *
56  *  @param msg Message of type HMT_REGPROC.
57  *  @return The filename.
58  */
59 char* hmsg_get_bitfilename(void* msg);
60
61 /*! \brief Set filename for FPGA bitstream.
62  *
63  *  @param msg Message of type HMT_REGPROC.
64  *  @param bitfilename The filename.
65  *  @return 0 on success. Negative on failure.
66  */
67 int hmsg_set_bitfilename(void* msg, char* bitfilename);
68
69 /*! \brief Get nice value (user defined priority).
70  *
71  *  @param msg Message of type HMT_REGPROC.
72  *  @return Positive nice value on success. Negative on
       failure.
73  */
74 int hmsg_get_nice(void* msg);
75
76 /*! \brief Set nice value (user defined priority).
77  *
78  *  @param msg Message of type HMT_REGPROC.
79  *  @param nice The positive value.
80  *  @return 0 on success. Negative on failure.
81  */
82 int hmsg_set_nice(void* msg, int nice);
83
84 /*! \brief Get sender ID for the message.
85  *
86  *  In this version of the HWOS, this is
87  *  always the message queue ID of the queue
88  *  owned by the sender.
89  *
90  *  @param msg Message of any type.
91  *  @return Positive message queue ID on success. Negative
       on failure.
92  */
93 int hmsg_get_sender(void* msg);
94
95 /*! \brief Set sender ID for the message.
96  *
97  *  In this version of the HWOS, this is
98  *  always the message queue ID of the queue
99  *  owned by the sender.
100 *
101 *  @param msg Message of any type.
102 *  @param id The message queue ID.
103 *  @return 0 on success. Negative on failure.
104 */
105 int hmsg_set_sender(void* msg, int id);
106
107 /*! \brief Get type of the message.
108  *
```

```
109 |  *  @param msg Message of any type.
110 |  *  @return The type (positive integer) on success. Negative
    |       on failure.
111 |  */
112 | int hmsg_get_type(void* msg);
113 |
114 | /*! \brief Get size of the data field.
115 |  *
116 |  *  @param msg Message of type HMT_CTRLMEM.
117 |  *  @return The size (positive integer) on success. Negative
    |       on failure.
118 |  */
119 | int hmsg_get_size(void* msg);
120 |
121 | /*! \brief Set size of the data field.
122 |  *
123 |  *  @param msg Message of type HMT_CTRLMEM.
124 |  *  @param size The size of the data field.
125 |  *  @return 0 on success. Negative on failure.
126 |  */
127 | int hmsg_set_size(void* msg, int size);
128 |
129 | /*! \brief Set return value for the specified message.
130 |  *
131 |  *  @param msg Pointer to a response message.
132 |  *  @param retval Return value.
133 |  *  @return Positive on success. Negative on failure.
134 |  */
135 | int hmsg_set_return(void* msg, enum hmsg_return retval);
136 |
137 | /*! \brief Get return value for the specified message.
138 |  *
139 |  *  @param msg Message of type HMT_CTRL or HMT_CTRLMEM.
140 |  *  @return The return value (constrained by enum
    |       hmsg_return).
141 |  */
142 | enum hmsg_return hmsg_get_return(void* msg);
143 |
144 | /*! \brief Get data (string of characters) for message.
145 |  *
146 |  *  @param msg Message of type HMT_CTRLMEM.
147 |  *  @return Pointer to data on success. NULL on failure.
148 |  */
149 | char* hmsg_get_data(void* msg);
150 |
151 | /*! \brief Set data (string of characters) for message.
152 |  *
153 |  *  @param msg Message of type HMT_CTRLMEM.
154 |  *  @param data Pointer to the data. Size not larger than
    |       HMSG_DATA_SIZE.
155 |  *  @return 0 on success. Negative on failure.
156 |  */
157 | int hmsg_set_data(void* msg, char* data);
158 |
159 | /*! \brief Set address field for a message.
160 |  *
161 |  *      @param msg Message of type HMT_CTRLMEM.
162 |  *      @return Positive on success. Negative on failure.
163 |  */
164 | int hmsg_set_address(void* msg, int address);
165 |
166 | /*! \brief Get address field for a message.
167 |  *
168 |  *      @param msg Message of type HMT_CTRLMEM.
169 |  *      @return Positive address on success. Negative on
    |       failure.
170 |  */
171 | int hmsg_get_address(void* msg);
172 |
```

```
173 /*! \brief Get command for message.
174  *
175  *  @param msg Message of any type.
176  *  @return The command entry (positive integer) for the
         message. Negative on failure.
177  */
178 enum hmsg_command hmsg_get_command(void* msg);
179
180 /*! \brief Set command for message.
181  *
182  *  @param msg Message of any type.
183  *  @param The command entry (positive integer) for the
         message.
184  *  @return 0 on success. Negative on failure.
185  */
186 int hmsg_set_command(void* msg, enum hmsg_command command);
187
188 /*! \brief Create and allocate memory for a message.
189  *
190  *  @param type The type of the new message.
191  *  @return Pointer to the memory of the new message on
         success. NULL on failure.
192  */
193 void* hmsg_create(int type);
194
195 /*! \brief Destroy message.
196  *
197  *  Actually just runs free(msg) if msg != NULL.
198  *
199  *  @param type Message of any type.
200  *  @return 0 on success. NULL on failure.
201  */
202 int hmsg_remove(void* msg);
203
204 /*! \brief Send a message.
205  *
206  *  Puts a message in a message queue.
207  *
208  *  @param msg Message of any type.
209  *  @param msq Message queue. Message is put in this queue.
210  *  @return 0 on success. Negative value on failure.
211  */
212 int hmsg_send(void* msg, struct hmqueue* msq);
213
214 /*! \brief Blocking wait for an incoming message.
215  *
216  *  @param msq The message queue where the message arrives.
217  *  @return Pointer to the received message on success. NULL
         on failure.
218  */
219 void* hmsg_receive(struct hmqueue* msq);
220
221
222 #endif
```

Listing D.18: hplacer.h

```
1  /*! \file hplacer.h
2   *  \brief [Hardware OS Placer Interface] Responsible for
       interrupting and replacing a running process on the FPGA
       .
3   *
4   *  Author (2011): Sindre Hansen
5   *
6   */
7
8  #ifndef HPLACER_H
9  #define HPLACER_H
10
```

```
11
12 #include "hstructures.h"
13
14
15 /*! \brief Set the full filename of the ICAP device.
16  *
17  *  Example: "/dev/icap"
18  *
19  *  @param device The full filename to the device.
20  */
21 int hplacer_set_icapdevice(const char* device);
22
23 /*! \brief Set the path of the bitfiles.
24  *
25  *  Sets the path to the files where the process
26  *  bitfiles are located.
27  *
28  *  @param path The path without the '/' at the end.
29  */
30 int hplacer_set_bitfilespath(const char* path);
31
32 /*! \brief Create full bitfilename.
33  *
34  *  This will allocate a string containing the path of the
        bitfiles and
35  *  the file name of the given process' bitfilename.
36  *
37  *  @param process Take bitfilename from this process.
38  */
39 char* hplacer_create_full_bitfilename(struct hprocess*
        process);
40
41 /*! \brief Interrupt a process running on the FPGA.
42  *
43  *  This assumes that the process is already placed on the
        FPGA and is running.
44  *
45  */
46 int hplacer_interrupt_process(struct hprocess* process);
47
48 /*! \brief Start a process running on the FPGA.
49  *
50  *  This assumes that the process is already placed on the
        FPGA, but not running.
51  *
52  */
53 int hplacer_start_process(struct hprocess* process);
54
55 /*! \brief Communicate with the running FPGA-process and
        make it send the
56  *          state information back to the HWOS.
57  *
58  *  This assumes that the process is already placed on the
        FPGA and is stopped.
59  *  This version of the HWOS will always store the state
        information in a file.
60  *
61  */
62 int hplacer_save_state(struct hprocess* process);
63
64 /*! \brief Load state information from memory/disk into the
        running process
65  *          on the FPGA.
66  *
67  *  This version of the HWOS will always fetch the state
        information from
68  *  a file on disk.
69  *
70  */
71 int hplacer_load_state(struct hprocess* process);
```

```
72
73 /*! \brief Load process into FPGA area.
74  *
75  *  This will place the given process on the FPGA.
76  *
77  */
78 int hplacer_load_process(struct hprocess* process);
79
80
81 #endif
```

Listing D.19: hprocess.h

```
1  /*! \file hprocess.h
2   *  \brief [Hardware OS Process Interface] Interface for a
       process on the FPGA.
3   *
4   *  Author (2011): Sindre Hansen
5   *
6   */
7
8  #ifndef HPROCESS_H
9  #define HPROCESS_H
10
11 #include "hstructures.h"
12
13 //! Lowest priority possible for a process.
14 //#define HPROCESS_LOWEST_PRIORITY 0
15
16 #define HPROCESS_MAX_FILENAME_SIZE 100
17
18 //! Number of possible states.
19 #define HPS_NUMBER 5
20 //! Definition of states for a process in the HWOS.
21 enum hpstate {
22         HPS_NEW                 //! Not running, just
              registered. An application has just asked to run
              this process.
23        ,HPS_READY               //! Not running, but is
              scheduled for placement and runtime on the FPGA.
              Not placed on FPGA.
24        ,HPS_BLOCKED             //! Blocked (waiting for an
               event or preempted). Not placed on FPGA.
25        ,HPS_BLOCKED_PLACED      //! Blocked (waiting for an
               event or preempted). Placed on FPGA.
26        ,HPS_RUNNING             //! Running. Process is
              placed and running on the FPGA.
27 };
28
29
30 /*! \brief Get base priority.
31  *
32  *  Returns the priority given to processes when
33  *  they are first accepted to the system.
34  *
35  *  @return Base priority of processes.
36  */
37 const int hprocess_base_priority();
38
39 /*! \brief Get maximum processes in system.
40  *
41  *  @return Maximum number of processes in the system.
42  */
43 const int hprocess_max_processes();
44
45 /*! \brief Set bitfilename for the process.
46  *
47  *  @param filename Filename.
```

```
48  *  @param filename Name of file where the FPGA bitstream is
        .
49  *  @return Maximum number of processes in the system.
50  */
51 int hprocess_set_bitfilename(struct hprocess* process, char*
        filename);
52
53 /*! \brief Get bitfilename for the process.
54  *
55  *  @return Name of file where the FPGA bitstream is.
56  */
57 char* hprocess_get_bitfilename(struct hprocess* process);
58
59 /*! \brief Get process by PID.
60  *
61  *  Worst case time of this function is O(n),
62  *  where n is the number of processes.
63  *
64  *  @param pid Process ID.
65  *  @return Pointer to the process on success. NULL on
        failure.
66  */
67 struct hprocess* hprocess_get(int pid);
68
69 /*! \brief Create process.
70  *
71  *  @param nice User defined value to define priority for
        process.
72  *  @return Process pointer on success. NULL on failure.
73  */
74 struct hprocess* hprocess_create(int nice);
75
76 /*! \brief Set priority for process.
77  *
78  *  @param pid Process ID for process.
79  *  @param priority Priority for process.
80  *  @return The new priority (larger than 0) on success.
        Negative on failure.
81  */
82 int hprocess_set_priority(struct hprocess* process, int
       priority);
83
84 /*! \brief Get priority for process.
85  *
86  *  @return The priority for the process on success.
        Negative on failure.
87  */
88 int hprocess_get_priority(struct hprocess* process);
89
90 /*! \brief Set nice value for process.
91  *
92  *  This is a user defined value that affects the priority
        of the process.
93  *
94  *  @param process Affected process.
95  *  @param nice Nice value for process.
96  *  @return The new nice value (larger than 0) on success.
        Negative on failure.
97  */
98 int hprocess_set_nice(struct hprocess* process, int nice);
99
100 /*! \brief Get nice value for process.
101  *
102  *  @param process Pointer to process.
103  *  @return The nice value for the process on success.
        Negative on failure.
104  */
105 int hprocess_get_nice(struct hprocess* process);
106
```

```
107 /*! \brief Get PID for process.
108  *
109  *  @param process Pointer to process.
110  *  @return The PID for the process on success. Negative on
         failure.
111  */
112 int hprocess_get_pid(struct hprocess* process);
113
114 /*! \brief Get next process in the list of processes.
115  *
116  *  @param process Pointer to process that has the next
         field.
117  *  @return Pointer to the next process. NULL on failure.
118  */
119 struct hprocess* hprocess_get_next(struct hprocess* process)
        ;
120
121 /*! \brief Get previous process in the list of processes.
122  *
123  *  @param process Pointer to process that has the previous
         field.
124  *  @return Pointer to the previous process. NULL on failure
         .
125  */
126 struct hprocess* hprocess_get_prev(struct hprocess* process)
        ;
127
128 /*! \brief Get state of process.
129  *
130  *  @param process Pointer to process.
131  *  @return State (positive integer) for process. Negative
         on failure.
132  */
133 int hprocess_get_state(struct hprocess* process);
134
135 /*! \brief Get state of process.
136  *
137  *  @param process Pointer to process.
138  *  @return State (positive integer) for process. Negative
         on failure.
139  */
140 int hprocess_is_valid_pid(int pid);
141
142 /*! \brief Remove process.
143  *
144  *  @param process Pointer to process.
145  *  @return 0 on success. Negative on failure.
146  */
147 int hprocess_remove(struct hprocess* process);
148
149 /*! \brief Get pointer to the queue the process belongs to.
150  *
151  *  @param process Pointer to process.
152  *  @return Pointer to the queue on success. NULL on failure
         .
153  */
154 struct hsqueue* hprocess_get_parent_queue(struct hprocess*
        process);
155
156 #endif
```

Listing D.20: hsignal.h

```
1 /*! \file hsignal.h
2  *  \brief [Hardware OS Signal Interface] Defines possible
       signals that can be connected between modules.
3  *
4  *  Author (2011): Sindre Hansen
5  *
```

```
6  */
7
8  #ifndef HSIGNAL_H
9  #define HSIGNAL_H
10
11 enum hsignal {
12         //! hsqlist to hsqueue. A queue has been added to a
               list.
13         HSIGNAL_ADD_QUEUE,
14         //! hsqlist to hsqueue. A queue has been removed
               from a list.
15         HSIGNAL_REMOVE_QUEUE,
16         //! hsqueue to hprocess. A process has been added to
               a queue.
17         HSIGNAL_ADD_PROCESS,
18         //! hsqueue to hprocess. A process has been removed
               from a queue.
19         HSIGNAL_REMOVE_PROCESS
20 };
21
22 #endif
```

Listing D.21: hsqlist.h

```
1  /*! \file hsqlist.h
2   *  \brief [Hardware OS Scheduler Interface] Defines a list
        of process queues.
3   *
4   *  The main purpose of this module is to keep lists of
        process queues.
5   *  Since there is a small and constant number of lists (as
        many as there are
6   *  process states), lists are referenced by a number from 0
         to HPS_NUMBER.
7   *
8   *  Author (2011): Sindre Hansen
9   *
10  */
11
12 #ifndef HSQLIST_H
13 #define HSQLIST_H
14
15
16 #include "hstructures.h"
17
18 /*! \brief Get size of list.
19  *
20  *  @param list The list.
21  *  @return Size of the list.
22  */
23 int hsqlist_size(int list);
24
25 /*! \brief Create new list.
26  *
27  *  @param list The list.
28  *  @return 0 on success. Negative on failure.
29  */
30 int hsqlist_create(int list);
31
32 /*! \brief Get queue by priority in list.
33  *
34  *  @param list The list.
35  *  @param priority The priority of the processes in the
        queue.
36  *  @return Pointer to the queue on success. NULL on failure
        .
37  */
38 struct hsqueue* hsqlist_get_queue(int list, int priority);
39
```

```
40 /*! \brief Add a queue to the list.
41  *
42  *  @param list The list.
43  *  @param queue The queue to be added.
44  *  @return 0 on success. Negative on failure.
45  */
46 int hsqlist_add_queue(int list, struct hsqueue* queue);
47
48 /*! \brief Insert queue before another queue in the list.
49  *
50  *  @param list The list.
51  *  @param second_queue The queue that is already in the
        list.
52  *  @param queue The queue to be inserted.
53  *  @return 0 on success. Negative on failure.
54  */
55 int hsqlist_insert_queue_before(int list, struct hsqueue*
      second_queue, struct hsqueue* queue);
56
57 /*! \brief Get state of the processes in the list of queues.
58  *
59  *  This functions assumes that this list has a list of
        queues
60  *  and that the all processes in the queues has the same
        state.
61  *
62  *  @param list The list.
63  *  @return State (positive integer) of the processes.
        Negative on failure.
64  */
65 int hsqlist_get_state(int list);
66
67 /*! \brief Get first queue in the list.
68  *
69  *  @param list The list.
70  *  @return Pointer to queue on success. NULL on failure.
71  */
72 struct hsqueue* hsqlist_get_first_queue(int list);
73
74 /*! \brief Get last queue in the list.
75  *
76  *  @param list The list.
77  *  @return Pointer to queue on success. NULL on failure.
78  */
79 struct hsqueue* hsqlist_get_last_queue(int list);
80
81 /*! \brief Checks if the given list value is in valid range.
82  *
83  *  @param list The list.
84  *  @return 1 if list value is in valid range. 0 if not.
85  */
86 int hsqlist_is_valid_range(int list);
87
88 /*! \brief Remove the given queue from the list.
89  *
90  *  @param list The list.
91  *  @param queue Pointer to the queue.
92  *  @return 0 on success. Negative on failure.
93  */
94 int hsqlist_remove_queue(int list, struct hsqueue* queue);
95
96 /*! \brief Remove the given list.
97  *
98  *  @param list The list.
99  *  @return 0 on success. Negative on failure.
100 */
101 int hsqlist_remove(int list);
102
103 /*! \brief Register the given signal.
```

```
104  *
105  *   This functionality is part of an
106  *   observer-observable pattern
107  *   between the list module and the queue
108  *   module (to keep correct
109  *   encapsulation).
110  *
111  *   The queue module register a given signal
112  *   with the list module.
113  *
114  *   After a signal is registered, changes done
115  *   in the list will make sure internal structure
116  *   in the affected queues are updated.
117  *
118  *   @param list The signal as defined by enum hsignal in
         hsignal.h.
119  *   @param function Pointer to the callback function in the
         hqueue module.
120  *   @return 0 on success. Negative on failure.
121  */
122 int hsqlist_register_signal(int signal, int (*function)());
123
124 #endif
```

Listing D.22: hstructures.h

```
1  /*! \file hstructures.h
2   *  \brief [Hardware OS public structures] Public structures
         used in the HWOS.
3   *
4   *   Author (2011): Sindre Hansen
5   *
6   */
7
8  #ifndef HSTRUCTURES_H
9  #define HSTRUCTURES_H
10
11 //! A process in the HWOS. Can be running on the FPGA.
12 struct hprocess;
13
14 //! A message queue in the HWOS. Used by the message server.
15 struct hmqueue;
16
17 //! A queue of processes in the HWOS.
18 struct hsqueue;
19
20 //! A device driver for a backend on the FPGA.
21 struct hdev;
22
23 //! A generic list structure for use in the HWOS.
24 struct hlist;
25
26 //! A generic list element for use in the generic list
       structure.
27 struct hlelement;
28
29 //! A generic list element that does not belong to any list
       element.
30 //! It has a next and a previous field.
31 struct hlorphan;
32
33 //! A pointer to a block of BRAM-memory on the FPGA.
34 struct hvmemptr;
35
36 //! An event for use in the event system between threads in
       the HWOS.
37 struct hevent;
38
39 #endif
```