Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Computer Engineering and Networks Laboratory

Winter term 02/03

Diploma Thesis 2003.11

Reconfigurable OS Prototype

Ruppen Michael

Advisors:

Herbert Walder
Matthias Dyer

Supervisor:

Prof. Dr. Lothar Thiele

Computer Engineering and Networks Laboratory (TIK)

Abstract

When thinking of an Operating System, most people think of a Personal Computer running Microsoft Windows or any Linux distribution. The task of such Operating Systems is to provide an execution environment for applications and share the available resources between these applications. Furthermore, most Personal Computers have one CPU and this single CPU must be shared, too. Now, today's Multimedia application must perform calculation-intensive tasks, for example filtering or streaming, and the performance of the whole system suffers from these calculations. These Audio-Filters, for example, can be implemented in hardware. The hardware can run the different calculation-steps in parallel, whereas the CPU only provides sequential calculation. This parallel-mode highly increases performance and, besides, saves precious resources like the CPU, for example.

Since it would be quite expensive to develop a specific hardware (so-called ASIC, Application Specific Integrated Circuit) for each single task, FPGAs (Field Programmable Gate Array) have been developed to overcome some drawbacks (time-to-market, price...). Furthermore, today's FPGAs are able to accomodate very large circuits (for example, a 128-bit parallel AES encryption circuitry). Recent generations of FPGAs provide a very interesting and useful feature, called Partial Reconfiguration. Partial Reconfiguration makes it possible to re-configure only a subset of the available chip area while the remaining circuitry keeps on running. Thus, these FPGAs can be seen as dynamically allocatable resource. Unfortunately, they do not provide infinite resources and, thus, there is the need of a higher instance which controls these resources, a Hardware Operating System. This Hardware Operating Systems is different to "normal" Operating System for two main reasons: First, the Operating System itself is split into a hardware and a software part and, second, execution runs in parallel. Such a Hardware Operating System has been implemented and this report shows the features of the Hardware Operating System.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an introduction to my diploma thesis "Reconfigurable OS Protoype". The first section gives a short overview of the structure of this report. Then, I would like to talk about the motivation on implementing such a Hardware Operating System. The third section defines the terms OS-Frame and task and throws a light on the different part such an operating system consists of. The fourth section gives an overview of the hard- and software tools that were used during this thesis. Finally, the last section defines the goal of this diploma thesis.

1.1 Overview

This report is intended to represent the work done during my diploma thesis. The second chapter deals with the main hardware part, the XESS Board. The third chapter reveals the inside of the XILINX® Virtex™ FPGA architecture and its configuration peculiarities. The fourth chapter introduces the hardware side of the OS-Frame and shows the approaches that have led to the current implementation. The fifth chapter illuminates the software side on the Host-PC, namely the classes and data structures that have been implemented to obtain the desired functionality.

1.2 Motivation: Why implementing a Hardware OS-Frame

Today's Embedded Systems may consist of a large variety of different processing elements, memories and I/O devices. These processing elements can be implemented in so-called ASIC's (Application Specific Integrated Circuits) but there are some drawbacks. ASIC's have a long time-to-market span and they are static processing elements. Static means that they cannot be adapted to new tasks without changing

the whole implementation. These circumstances lead to the introduction of programmable hardware devices such as FPGAs.

Today's FPGAs can accomodate very large circuits. Moreover, new series of FPGA's make it possible to reconfigure only a specific part of the available chip area; this is called *partial reconfiguration*. Partial reconfiguration can be done during runtime without affecting the other parts of the chip. For instance, imagine three circuits A, B and C. A and B are currently running. One can now replace circuit A by C without affecting the running circuit B.

Because of this partial reconfigurability of recent FPGA's, they can be viewed as dynamic allocatable resources. One can imagine, that the correct replacement of circuits and the control of the chip's resources should be carried out under a higher instance, the so-called Hardware OS-Frame. The allocation and sharing of the available resources of today's FPGAs make the usage of a controlling instance unavoidable.

1.3 The Hardware OS-Frame

1.3.1 Terms

The term *Hardware OS-Frame*, or *OS-Frame* for short, denotes the composition of interacting hard- and software components and services running on the FPGA and the Host-PC, respectively. The fact that the Hardware OS-Frame is split into a hardware part running on the FPGA and a software part running on a Host-PC distincts this kind of operating system from pure software operating systems such as Linux, Windows etc. Another important property of the Hardware OS-Frame is its real-time characteristic. Since different circuits on a FPGA run in parallel, the operating system should run in real-time, too, in order to provide the desired functionality. The second term that is important in this context is called *hardware task* or just *task* for short. A task is a part of an application's functionality, for instance a FIR filter or a decoding algorithm (further examples will be introduced in section 4.4). The two terms *OS-Frame* and *task* are used throughout the whole report.

1.3.2 The Hard- and Software Parts of the OS-Frame

In order to provide different services to the task developers, the OS-Frame is split into two parts: The hardware on the FPGA and the software on the Host-PC. Fig. 1.1 depicts this splitting. There are several services the OS-Frame should provide, for instance buffers for intertask communication (see [10]), I/O Pin allocation etc.

Fig. 1.1 does not intent to show too much details, since the different parts of the OS-Frame will be introduced in the following chapters.

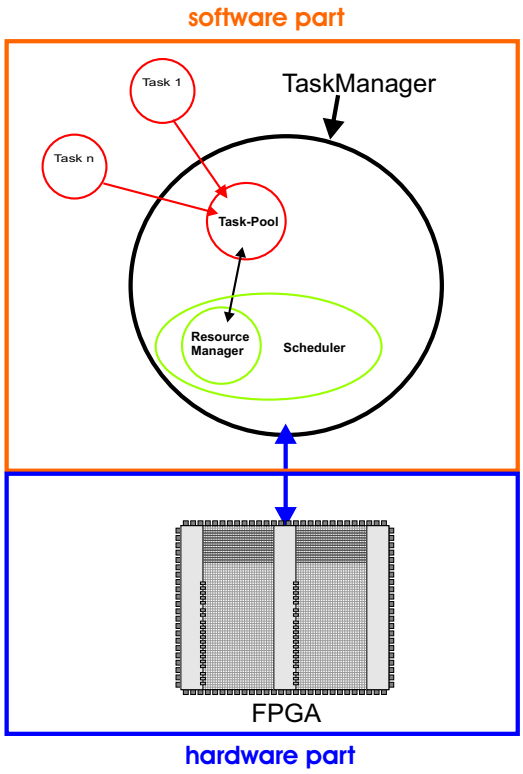

Figure 1.1: The two parts of the OS-Frame

1.4 The Hard- and Software Tool-Set

1.4.1 The Hardware

There are three main hardware parts that have been used during this thesis, namely the XESS XSV-800 Developement Board, a PCI7200 I/O Card and a x86 Host-PC. Since the XESS Board is somewhat a complex Developement Board, a seperate chapter (see 2) is dedicated to it. The PCI7200 I/O Card provides 32 programmable inputs and 32 programmable outputs, as well. It is shipped with the corresponding C++ classes and libraries to provide a defined programming interface. This I/O card was used to configure the FPGA on the XESS Board and to send runtime-configuration-data to the OS-Frame on the chip. The Host-PC was a generic Intel Pentium III processor with a clock frequency of 1 GHz. It was equipped with 512 MB of RAM and 20 GB of harddisk memory. Well, these are the most important hardware components that were used during the diploma thesis.

1.4.2 The Software

As one can imagine, different software toolsuites were used to produce the bitstreams for the FPGA on one hand and to develop the Host-PC software on the other hand. The bitstreams were generated by the XILINX® ISE 5.0 Foundation Software. This package supports the whole VHDL design flow, that means it builds the correct bitstream out of the VHDL code. Besides, this software was extended with the recently released Modular Design Feature, also published by XILINX® . This packages allows a team of designers to independently develop different parts of the design and merge (or *assemble*) them into one design, or one bitstream. It also allows to produce partial bitstreams and that was the main reason why we had chosen this specific software package.

To obtain a running application on the Host-PC that is able to control to OS-Frame on the FPGA, we've chosen Microsoft Visual C++ 6.0 Professional as software developement tool. This toolsuite is able to produce executables out of C++ code. We've chosen C++ as main programming language, since the PCI7200 I/O Card is delivered with a C++ API (classes and libraries) which allows the user to make use of its I/O ports in a very convenient way.

Chapter 2

The XESS XSV-800 Prototyping Board

This chapter is intended to give a short summary of the features the XESS XSV-800 Prototyping Board provides, as well as the restrictions that come along when using this board. Fig. 2.1 shows a photograph of the XESS Board. Since there are quite a lot of features, only the important ones are covered in this chapter. To obtain more information about the various interfaces, the interested reader is refered to [1].

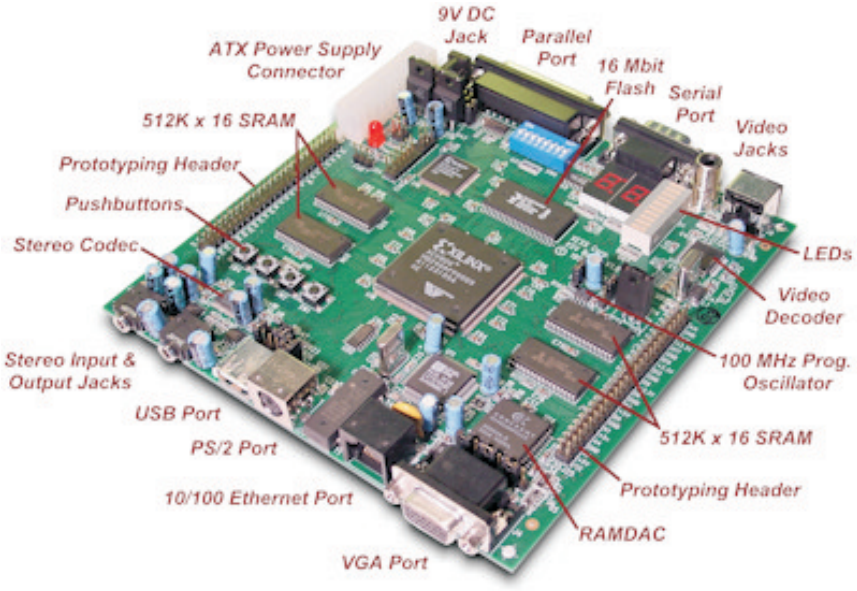

Figure 2.1: Photograph of the XESS XSV-800

2.1 The Features of the XESS Board

As one can see in Fig. 2.1, the XESS Board provides a lot of interesting features to developers. Table 2.1 shows the available resources:

Chip	Description
XILINX® Virtex XCV800 FPGA	The programmable device with 888 Kgates in a 240-pin HQPF package
XILINX® XC95108 CPLD	CPLD to manage the configuration flow of the FPGA
Dallas DS1075 oscillator	Programmable oscillator to generate the clock signal for the FPGA & CPLD
Intel 28F016S5 Flash RAM	Flash RAM, accessible by FPGA & CPLD
Winbond AS7C4096 SRAM	Two independant SRAM banks with 512K x 16 bits of memory
Philips SAA7113 Video Decoder	Digitizes NTSC, SECAM and PAL video signals
BT481A RAMDAC	Generates video signals for VGA monitors with 24-bit color depth
AK4520A Audio Codec	Digitizes two analog inputs for the FGPA and/or converts serial bit streams from the FPGA into two analog output signals
LXT970A Ethernet PHY	Manages the physical Ethernet (LAN) Layer
Expansion Headers	Two Expansion Headers with 25 pins to connect external systems to the FPGA and vice versa
10-segment bargraph LED	LEDs that can be used by the FPGA & the CPLD
7-segment LED digits	Two 7-segment LED digits that can be used by the FPGA & the CPLD
Philips PDIUSBP11A USB Interface	high-speed and low-speed USB interface connected to the FPGA
Parallel Port	Parallel Port interface connected to the CPLD that is used for the configuration of the FPGA & the CPLD
Serial Port (RS232)	Serial Port interface connected to the CPLD

Table 2.1: The available resources on the XESS Board

There are only a few features that are relevant in the context of this thesis. These are the Audio Codec, the RS232 interface, the 10-segment bargraph LED and the Expansion Headers. These resources are statically allocated by the OS-Frame. As mentionned above, there are some restrictions coming along when using

these resources. The following section explains the allocated features in detail and hints at the resulting restrictions.

2.2 The Allocated Resources

Fig. 2.2 shows the different components and their interconnections on the XESS Board. Obviously, there are some components that are connected to the same FGPA-pin's and these circumstances lead to some problems, for instance, the SRAM banks are connected to the same pins as the Expansion Headers are. If one would like to use the Expansion Headers, the SRAM banks must be disabled, since the outputs of the SRAM chips are in a floating state per default.

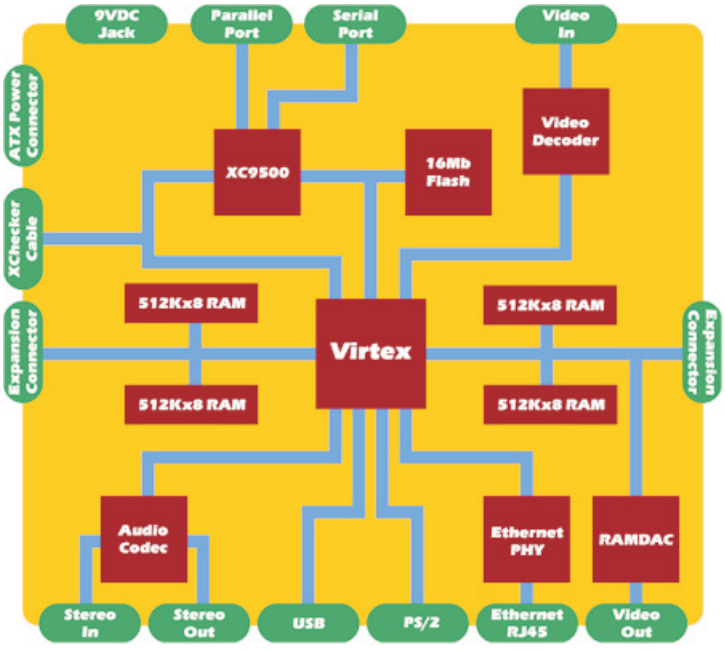

Figure 2.2: Blockdiagram of the XESS Board

2.2.1 The Audio Codec

Table 2.2 shows the interface of the AK4520A Audio Codec chip.

The are two operations the Audio Codec supports: First, it takes two analog input channels from the jack J1 (Stereo IN), digitizes these values and sends them via the SDOUT pin to the FPGA as a serial bit stream and, second, it accepts a serial bit stream from the FPGA and converts this bit stream into two analog signal that are sent to jack J2 (Stereo OUT). The FPGA sends the serial clock to the SCLK pin and the Codec synchronizes the bit streams with this serial clock.

Audio Codec Pin	Virtex FPGA Pin
MCLK	3
LRCK	4
SCLK	5
SDIN	6
SDOUT	7

Table 2.2: The Audio Codec Pin connections

The LRCK is used to select either the left or the right channel as source/destination of the serial data. The MCLK synchronizes all operations done within the Audio Codec.

The FPGA's pins 3, 4, 5, 6 and 7 are only used to connect to the Audio Codec and vice versa, so there don't exist any conflicts with the remaining circuitry of the XESS Board. This means, one can use the Audio Codec without any resulting restrictions.

2.2.2 The RS232 Interface

As one can see in Fig. 2.2 there are no direct connections between the RS232 port and the Virtex FPGA. The RS232 port is only connected to the CPLD and this means, that the CPLD must be configured accordingly to re-direct data streams from the RS232 RD port to the FPGA on one hand, and from the FPGA to the RS232 TD pin on the other hand. Table 2.3 depicts the connections between the RS232 port and the CPLD.

RS232 Pin	CPLD Pin
RTS	82
TD	81
CTS	85
RD	80

Table 2.3: The RS232 Pin connections

Since the current implementation only makes use of the TD and RD pins, we neglect the RTS and CTS pins in further discussions. By programming the CPLD accordingly, the connection between the RS232 and FPGA can be established. Table 2.4 shows the pins that have been used to re-direct the RS232 signals from the CPLD to the FPGA and vice versa.

Pins 80 and 81 of the CPLD are only connected to the RS232 port and, thus, no problems can occur when allocating these pins. However, pins 19 and 20 of the CPLD and pins 141 and 144, as well, are connected to other circuitry, too. FPGA-Pin 141 is connected to CPLD-Pin 19, to one pin of the right 7-segment digit LED

RS232 signal	CPLD Pin	CPLD re-direct Pin	FPGA Pin
TD	81	20	144
RD	80	19	141

Table 2.4: Pin assignement to connect the RS232 to the FPGA

(S4) and to pin A3 of the Flash RAM. FPGA-Pin 144 is connected to CPLD-Pin 20, to one of pin of the right 7-segment digit LED (S5) and to pin A4 of the Flash RAM. Since both 7-segment pins (S4 & S5) are input pins and both Flash RAM pins (A3 & A4) are input (address) pins, there are not severe problems. The only restriction is that the Flash RAM and the mentioned 7-segment digit pins cannot be allocated for other use, since they should not be driven by multiple sources.

2.2.3 The 10-segment Bargraph LED

The last resource that has been allocated as a general-purpose I/O resource is the 10-segment bargraph LED. Table 2.5 shows the FPGA-Pins that are connected to these 10 LED's:

Bargraph Pin	Virtex FPGA Pin
BAR0	152
BAR1	154
BAR2	157
BAR3	160
BAR4	162
BAR5	169
BAR6	168
BAR7	173
BAR8	131
BAR9	171

Table 2.5: 10-segment Bargraph Pin assignment

Again, there are some restrictions that come along when using the 10-segment bargraph LED: All pins are connected to the FPGA on one hand and to some pins of the Flash RAM on the other hand. This results once more in that the Flash RAM cannot be used for other purpose and must, therefore, be disabled.

2.2.4 The Parallel Port

Another resource that has statically been allocated is the Parallel Port. The Parallel Port is used to configure, or program, the CPLD and the Virtex FPGA. Since the Parallel Port is not connected to the Virtex FPGA directly, the CPLD must be

configured correctly in order to download bitstreams on the FPGA.

The CPLD only supports the JTAG configuration mode. This mode uses four signals to program the chip. The Virtex FPGA supports different programming modes, or programming interfaces. One of them is called the SelectMAP® interface. This configuration interface needs eight data lines and, thus, the CPLD must redirect the signals coming from the Parallel Port to the SelectMAP interface. In addition to the eight data lines, this interface needs some controlling lines. The SelectMAP configuration flow and its corresponding signals are introduced in the next chapter and are not discussed here.

Since this SelectMAP interface is quite important for the configuration of the Virtex FPGA, there are a few dedicated connections between the CPLD and the Virtex' SelectMAP pins, but, again, there is some circuitry that must be disabled when using the SelectMAP interface, namely the left 7-segment digit LED and the Flash RAM's data pins.

2.2.5 The Expansion Headers

The last important resource on the XESS Board that has been allocated are the two Expansion Headers. Each of them provides 25 I/O pins that can be used as general purpose I/O. In the context of this thesis, they have been allocated to control the OS-Frame on the FPGA via the Host-PC. The protocol of this PC-Interface is discussed in chapter 5. There is one problem that appears when using the Expansion Headers: These are connected to the same FPGA-Pins as the SRAM banks are, and, thus, the SRAM chips must be disabled. In this case, the chip select ($\overline{CE}$) signal of the SRAM must be asserted (this disables the SRAM chips) since the data pins of the SRAM chips are floating when no proper address is applied and this can disturb the functionality of the PC-Interface. Table 2.6 shows the two $\overline{CE}$ signal of the SRAM banks:

SRAM bank	Virtex FPGA Pin
$\overline{CE}$ (left)	186
$\overline{CE}$ (right)	109

Table 2.6: $\overline{CE}$ signal for the SRAM banks

Since only the Expansion Headers and the SRAM banks are connected to the same FPGA-Pins, no other circuitry affects the PC-Interface and by disabling the SRAM banks with the $\overline{CE}$ signals, the proper functionning of the PC-Interface can be guaranteed.

2.3 Summary

This chapter has introduced the features of XESS XSV-800 Prototyping Board used during this thesis. This board offers quite a lot of interesting options, but also brings along some restrictions we had to take into consideration. There are a few features that have been cut out to guarantee a correct and stable functionning of the OS-Frame. The disabled circuitry consists of the Flash RAM, the SRAM banks on both sides and both 7-segment digit LEDs.

Chapter 3

The XILINX® Virtex™ Architecture

This chapter is dedicated to the most important resource on the XESS XSV-800 Prototyping Board: The Virtex™ XCV800 FPGA. The next few sections are intended to give an introduction on the inside of the Virtex on one hand and on the configuration flow on the other hand. There are some special qualities that come along when starting to partially (re-)configure the FPGA and that's why this chapter has been inserted in this report. This chapter merely covers 'thesis-specific' features and, thus, the reader is advised to read the application notes [2], [3] and [5] in order to obtain more detailed information on the Virtex' inside and the configuration flow.

3.1 Architecture Overview

This section reveals the architecture of the Virtex FPGA. Fig. 3.1 depicts a very coarse overview of the Virtex architecture. The Virtex device contains a certain amount of configurable logic blocks, so-called CLBs, input-output blocks (IOBs), block RAMs, clock resources, programmable routing and configuration circuitry.

The next three subsections cover the most important parts of the Virtex FPGA, namely the IOBs, the CLBs and the routing resources.

3.1.1 Input/Output Blocks (IOBs)

Fig. 3.2 depicts the internal structure of an IOB. One IOB provides three storage elements which function either as level sensitive latches or as edge-triggered D-type flip-flop. The flip-flops share the same clock signal but each of them has a seperate CE (clock enable) signal. Additionally, the storage elements share a Set/Reset signal which can either be of a synchronous or asynchronous manner.

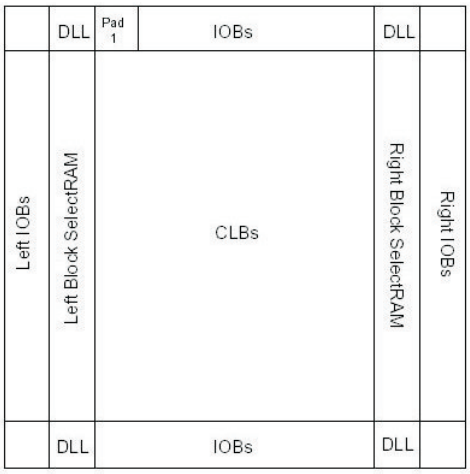

Figure 3.1: Virtex Architecture Overview

Furthermore, each IOB provides a weak pull-up resistor and a weak pull-down resistor, respectively. These resistors make it possible to force the pad's state either into a logic high or a logic low level. Another advantage of this capability is that these pads are able to generate a defined logic level for the circuitry the user has programmed; this fact becomes important when using the XILINX® Modular Design Tools (see chapter 6). The exact usage of the pull-up, or pull-down, primitives will be explained in a later section (see 4.2.3.1).

3.1.2 Configurable Logic Blocks (CLBs)

A Configurable Logic Block, or CLB for short, consits of several parts that should be examined in this subsection. Fig. 3.3 shows a such a CLB. A CLB consists of two slices which contain the same elements.

The basic building block of a CLB is the so-called logic cell (LC). Each LC contains a 4-input function generator (implemented as 4-input look-up table), carry logic and a storage element (D-Type flip-flop). A CLB includes four LCs, organized in two equal slices (see Fig. 3.3).

Look-Up Tables Each look-up table, or LUT for short, supports two operation modes: First, it can operate as a function generator as stated above, or, second, each LUT provides a 16 x 1-bit synchronous RAM. Besides, the two LUTs of a slice can be combined to provide either a 16 x 2-bit or 32 x 1-bit synchronous RAM or a 16 x 1-bit dual-port synchronous RAM.

Storage Elements The storage elements in a Virtex slice can be configured either as edge-triggered d-type flip-flop or as level-sensitive latches. The D inputs

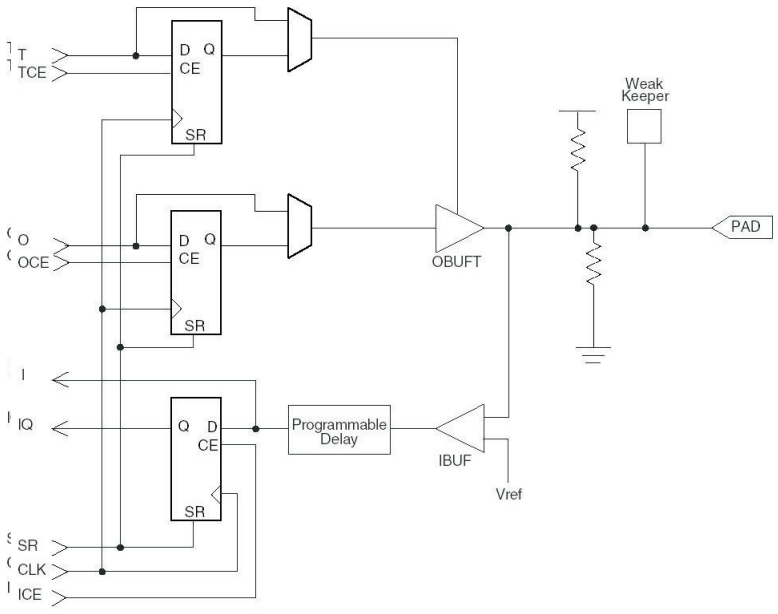

Figure 3.2: Virtex Input/Output Block (IOB)

can either be driven by the LUTs or directly from the slice inputs.

Furthermore, the flip-flops of a slice provide a Clock Enable signal and one either synchronous or asynchronous Set/Reset signal. Altering one of those control signal affects both flip-flops of a slice.

BUFTs Each Virtex CLB contains two 3-state drivers (BUFTs) that are connected to on-chip busses. Each BUFT has an independant 3-state enable pin and an independant input pin. These BUFTs are connected to horizontal Tri-State Lines as depicted in Fig. 3.4.

Block SelectRAM The Virtex architecture provides two BlockRAM columns, one being on the left side and the other on the right side of the die. They both extend the full height of the device. Both columns consists of several memory blocks with one memory block being four CLBs high. Thus, the Virtex XCV800 which has 56 CLB-rows, has got 14 memory blocks on each side and, thus, a total of 28 blocks.

3.1.3 Programmable Routing Matrix

There are several routing resource available on the chip. These resource are allocated according to the timing constraints or other purpose that should be fulfilled. Those different routing 'classes' are introduced in this subsection.

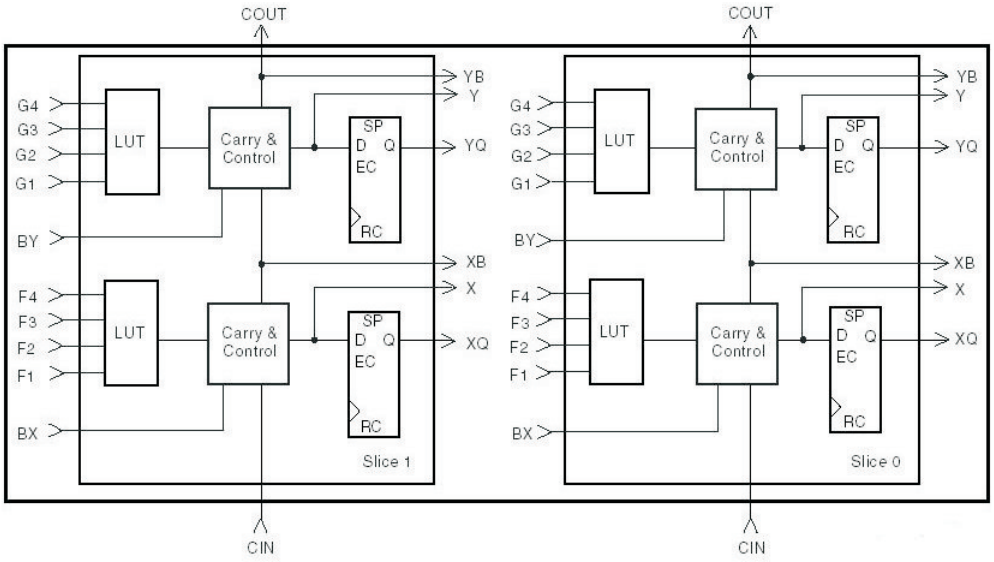

Figure 3.3: The two slices of a Configurable Logic Block (CLB)

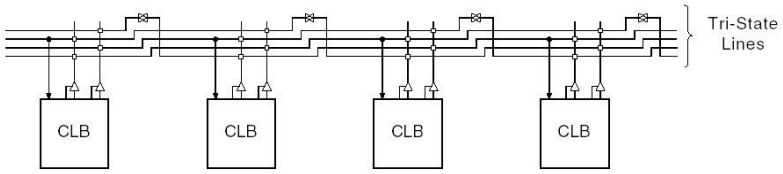

Figure 3.4: Horizontal Routing Resource: Tri-State Lines

General Purpose Routing Most signals are routed on the general purpose routing. The general routing resources are located in vertical and horizontal routing channels associated with the rows and columns of the CLB matrix.

Global Routing Global routing resources distribute clock signals and other signal with very high fanout throughout the device. There are two nets that provide this high fanout routing:

- The primary global routing resources consist of four dedicated global nets with their corresponding input pin to distribute high-fanout clocks with a very low skew. Each of them can drive every CLB, IOB and RAM clock pins.
- The secondary global routing resources consist of 24 backbone lines. They are intended for a more general global routing since they are not restricted to drive clock pins.

Dedicated Routing Some classes of signals require dedicated routing resources. Virtex devices support two major signal classes which make use of these dedicated resources:

- Horizontal routing resources provide 3-state busses. Per CLB row, four bus lines are available. These lines, so-called Tri-State Lines, are driven by the BUFTs as explained above (see Fig. 3.4).
- Two dedicated nets per CLB carry signals vertically to the adjacent CLB.

Clock Distribution Virtex devices provide four high-fanout clock distribution nets with very low skews (primary global routing resource). Each of them is driven by a global buffer, two on top center and two on the bottom center of the device. Furthermore, there are four dedicated clock pins that drive the corresponding global buffer (see Fig. 3.5).

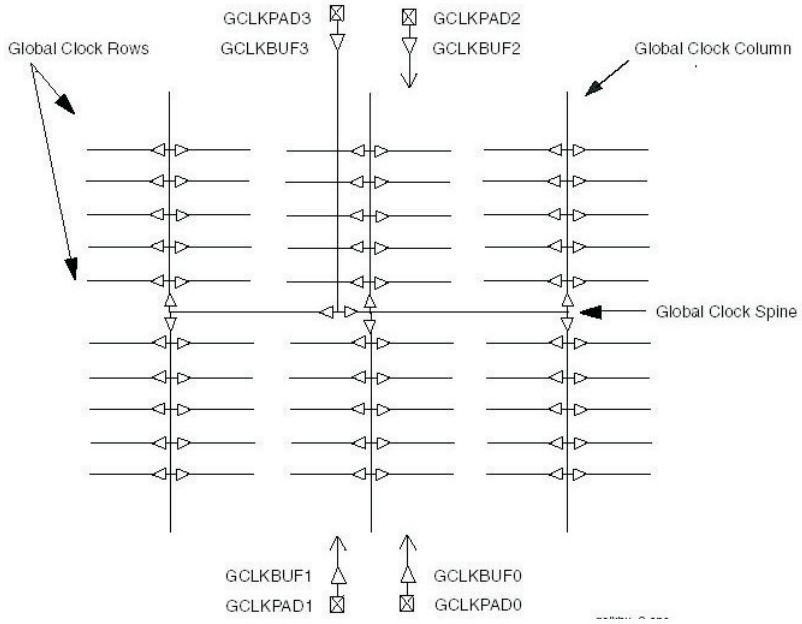

Figure 3.5: Global Clock Distribution

3.2 Configuration Flow for the Virtex FPGA

This section gives an overview of the configuration of the Virtex FPGA. There are eigth different configuration modes as shown in table 3.1. The current configuration mode is chosen by asserting the proper signals to pins M2, M1 and M0. Apart from the four basic modes (the first four ones in the table), the user can force the IOBs

into a logic high level during configuration. After configuration, the pull-ups are de-asserted and the IOBs fall back into their previous state.

Configuration Mode	M2	M1	M0	Pull-ups
Master Serial	0	0	0	No
Slave Serial	1	1	1	No
SelectMAP	1	1	0	No
Boundary Scan (JTAG)	1	0	1	No
Master Serial (w/pull-ups)	1	0	0	Yes
Slave Serial (w/pull-ups)	0	1	1	Yes
SelectMAP (w/pull-ups)	0	1	0	Yes
Boundary Scan (JTAG) (w/pull-ups)	0	0	1	Yes

Table 3.1: Virtex Configuration Modes

NOTE: This section only covers the SelectMAP configuration mode since this is the one being used during the thesis.

3.2.1 Initialization and Timing

The initialization sequence is quite simple: Upon power-up, the FPGA configures the internal circuitry. The $\overline{\text{INIT}}$ signal is held low during this initialization sequence. As soon as the $\overline{\text{INIT}}$ signal goes high, the configuration may start.

3.2.2 The SelectMAP Mode

Fig. 3.6 depicts the situation when configuring the Virtex™ FPGA via the SelectMAP interface. There are quite a lot of signals that must be controlled during configuration.

As one can see, the SelectMAP interface provides an 8-bit parallel configuration interface. These eight data lines are bidirectional, that means they can be used for configuration and readback.

DATA Pins (D[0:7]) The pins D0 through D7 work as an 8-bit wide, bidirectional bus when using the SelectMAP mode. Thus, configuration bitstream are written byte-wise to the data pins with D0 being the Most Significant Bit (MSB). The $\overline{\text{WRITE}}$ signal controls the direction of the bus.

$\overline{\text{WRITE}}$ When this signal is asserted low, configuration data is written to the data bus and, thus, when $\overline{\text{WRITE}}$ is asserted high, configuration data is read from the bus. This means, the device is being configured when $\overline{\text{WRITE}}$ is low.

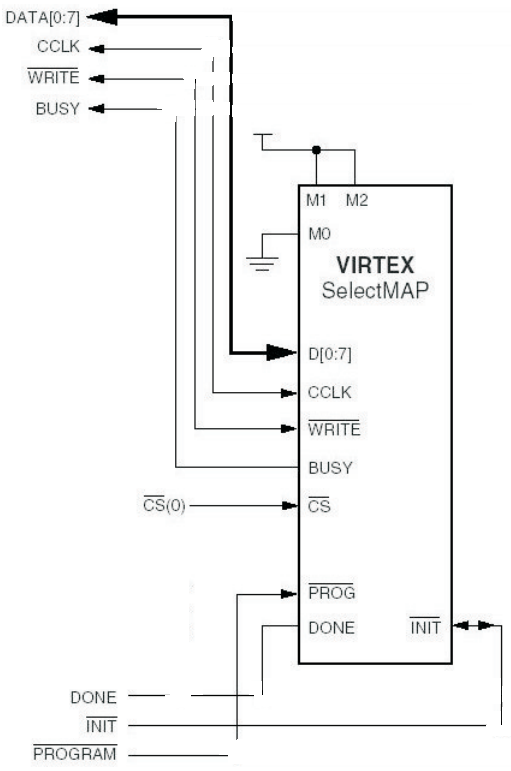

Figure 3.6: SelectMap Configuration Setup for Virtex™ devices

$\overline{\text{CS}}$ The Chip Select input ($\overline{\text{CS}}$) enables the SelectMAP data bus. When reading or writing data from or to the data bus, $\overline{\text{CS}}$ must be asserted low. When the signal is high, the data bus is disabled.

BUSY This signal is only to be taken into consideration when the $\overline{\text{CS}}$ signal is asserted low (when the data bus is enabled). If BUSY is low, the FPGA reads the next byte on the next rising clock edge of the CCLK signal where both $\overline{\text{CS}}$ and $\overline{\text{WRITE}}$ are asserted low. When BUSY is driven high, this indicates, that the internal configuration circuitry is not ready and, thus, the current byte is ignored and must be reloaded as soon as BUSY gets low again (see Fig. 3.7). BUSY is tri-stated when $\overline{\text{CS}}$ is not asserted.

This BUSY signal is only needed when one wants to configure the FPGA with frequencies above 50 MHz. When CCLK is below the 50 MHz border, the BUSY signal needn't be taken into account.

CCLK The CCLK pin is an input clock pin that synchronizes all loading and reading of the data bus for configuration and readback. Besides, the CCLK drives all internal configuration circuitry. The CCLK may be driven either by a free running

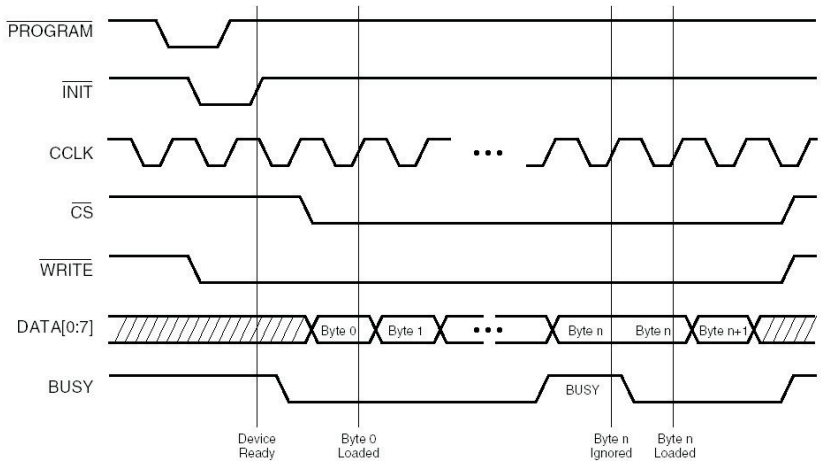

Figure 3.7: The BUSY signal during configuration (above 50 MHz)

oscillator or an externally-generated signal.

3.2.3 The Configuration Flow

The configuration data format of the Virtex family is independant from the configuration mode; it follows always the pattern: Imagine the Virtex configuration memory as a rectangular array of bits. The bits are grouped in one-bit wide portions, so-called *frames*. Thus, frames are the smallest unit that can be written (or read back).

Frames are grouped together into larger units called *columns*. There exist five different types of columns as depicted in table 3.2.

Column Type	# of frames	# per Device
Center	8	1
CLB	48	# of CLB columns
IOB	54	2
Block SelectRAM Interconnect	27	# Block SelectRAM columns
Block SelectRAM Content	64	# Block SelectRAM columns

Table 3.2: Configuration Column Types

As mentioned above, configuration data is separated into frames and the frames are grouped together into columns. Fig. 3.8 depicts the configuration columns for a standard Virtex device.

There's one center column which consists of 8 frames. This center column contains the configuration data for the four global clock buffers and their corresponding clock nets. Two IOB columns hold the configuration for the IOBs on the left and

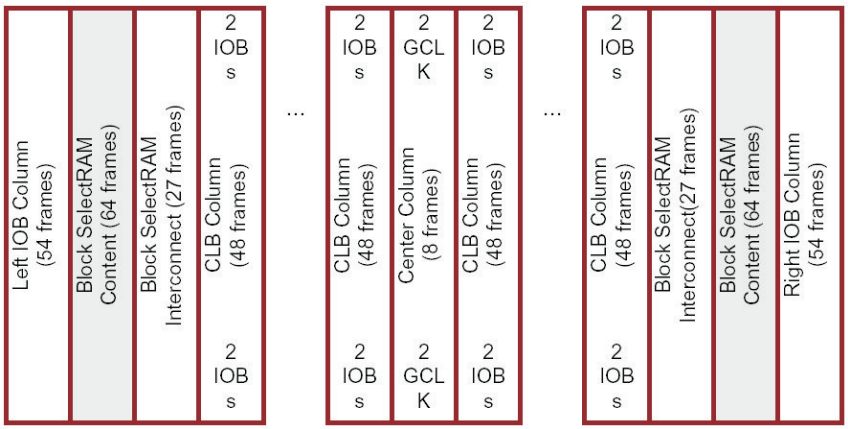

Figure 3.8: Virtex™ Configuration Column Example

right side of the die. The CLB columns contain the configuration for the CLBs and the adjacent IOBs on the top and the bottom, respectively. The remaining two column types contain information about the Block SelectRAM; one contains interconnection-data and the other one contains the content-related parts.

3.2.3.1 Configuration Addressing

The total address space is divided into two different block types: RAM and CLB. The RAM block type only contains the Block SelectRAM content columns, whereas the CLB block type contains all other column types (Center, CLB, IOB and Block SelectRAM Interconnect). Both address spaces are divided into major and minor address. Each column has its own, unique major address within the RAM or CLB address space. Each frame has its unique minor address within its configuration column.

For Virtex devices, the major addressing scheme starts with 0 for both, the RAM and CLB address space. Thus, the center column of the CLB address space has major address 0. The ascending major addresses alternate between the right and left side of the die, that means the first CLB column of the right side has major address 1 and the first CLB column on the left side of the die has major address 2 and so on. Thus, odd major addresses point to CLB columns of the right side of the die, whereas even major addresses point to those on the left side. After the CLB columns, the IOBs on the right and left side follow and, finally, the Block SelectRAM Interconnect columns follow the IOB cofiguration columns. Fig. 3.9 depicts this situation: First, the center column with major address 0, then the CLB columns with alternating major addresses up to 24, third, the IOB columns with major address 25 on the right side and 26 on the left side of the die and, finally, the Block SelectRAM Interconnect columns with major addresses 27 on the right side and 28 on the left side. This consecutive order (Center, CLB, IOB, BlockRAM

Interconnect) must be followed when configuring the device.

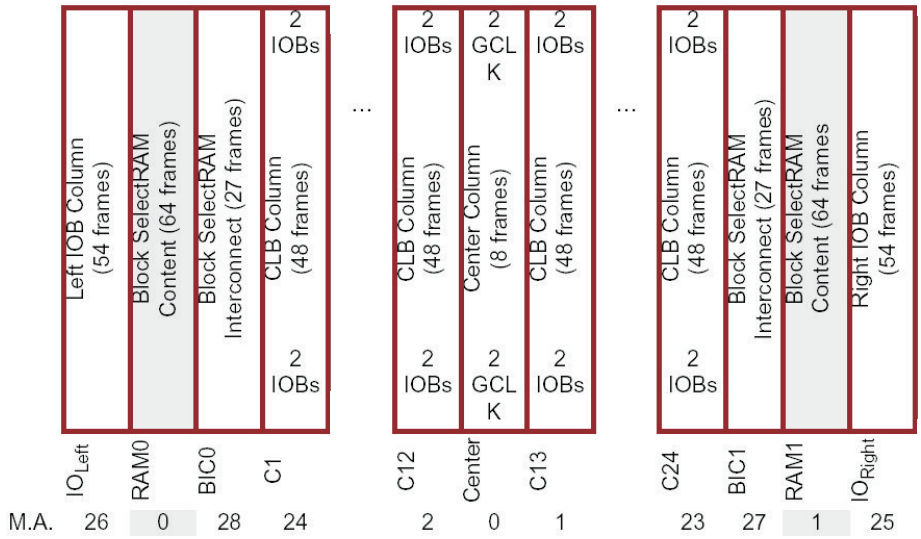

Figure 3.9: Addressing scheme for Virtex™ devices: Major Addresses

RAM block type columns are addressed in the same manner as CLB columns are: The major address for the **left** Block SelectRAM reads 0 and the major address for the **right** one reads 1 (Virtex™ devices only contain two Block SelectRAM resources).

Frames Since frames are the atomic unit of the configuration data of the Virtex device, let's have a closer look at them: Frame size depends on the number of CLB rows of the current device: the more rows the larger the frames. The frame size can be calculated using the following formula: $18 \times (\#\text{CLB\_rows} + 2)$. To this calculated size, a certain amount of padding zeroes are added to fit in 32-bit words. For the FPGA used in this thesis, the Virtex™ XCV800 which has 56 CLB rows, the frame size calculates to 1088 bits and the number of 32-bit words, thus, is equal to 34.

Fig. 3.10 depicts the frame organisation of the CLB block type frames: The first 18 bits are used for the top IOB, then the bits for the CLBs follow, and, finally, the last 18 bits are reserved for the bottom IOB.

Figure 3.10: CLB block type frame

Fig. 3.11 depicts the organisation of the RAM block type columns: First, there

are 18 padding bits, then the content data of the RAM cells follow and, finally, there are again 18 padding bits.

Figure 3.11: RAM block type frame

3.2.3.2 Writing Configuration Data to the FPGA

Virtex configuration can be seen as a sequence of commands and corresponding configuration data. For an initial configuration, the following sequence of commands and data must be followed:

1. Issue one or more pad words (FF FF FF FF in hexadecimal)
2. Issue Sync word (AA 99 55 66 in hexadecimal)
3. Reset CRC register (this register is intended to check the correctness of the written data)
4. Set configuration specific flags (see [3] for more details)
5. Set the FAR (see below) to the starting address
6. Issue a WCFG (write configuration, see below) to the CMD register
7. Write the number of words to be written to the FDRI register (see below)
8. Write data frames

A command is organized as a packet with a command header and, depending on the command, with optional data words. For the Virtex™ series, a variety of commands have been defined. In this report, only a few of them are introduced:

3.2.3.3 Configuration Registers

The general command header format is depicted in Fig. 3.12. The first three bits identify this command header by setting the type bits to 001. The OP bits specify whether this command issues a read (01) or write (10) operation. Then, bits 26 through 17 are set to zero. The next four bits, 16 through 13, specify the configuration register address (see table 3.3 for the appropriate values). Bits 12 and 11 are set to zero and the last 11 bits, bits 10 through 0 specify the amount of data words that follow the current command (max. value is 2047 words).

Type			OP	Register Address												RSV	Word Count											
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	x	x	0	0	0	0	0	0	0	0	0	0	x	x	x	x	0	0	x	x	x	x	x	x	x	x	x	x	x

Figure 3.12: Command Header Format

Register Name	Mnemonic	R/W	Binary Address
CRC	CRC	R/W	0000
Frame Address	FAR	R/W	0001
Frame Data Input	FDRI	W	0010
Frame Data Output	FDRO	R	0011
Command	CMD	R/W	0100
Control	CTL	R/W	0101
Control Mask	MASK	R/W	0110
Status	STAT	R	0111
Legacy Output	LOUT	W	1000
Configuration Option	COR	R/W	1001
Frame Lentgh	FLR	R/W	1011

Table 3.3: Configuration Register Address

The Command Register (CMD) The content of the Command Register is interpreted by configuration state machine. This register controls the operation of the configuration state machine, the Frame Data Register (FDR), and some of the global signals. The effect of each command is shown in table 3.4.

Thus, the command header for the Command Register has always the same structure: 30 00 80 01 (written in hexadecimal format, so each digit represents four bits of the command header). The 1 at the end tells the configuration state machine that one word out of the collection depicted in table 3.4 (Note: Table 3.4 is not complete, please refer to [3] on page 18 for a complete list of available commands) follows the CMD. For example, if one wants to reset the CRC register, the following sequence must be performed:

30 00 80 01 CMD
00 00 00 07 Code for Reset CRC as shown in table 3.4

If one wants to write some configuration data, the following command sequence must be issued:

30 00 80 01 CMD
00 00 00 01 Write Configuration Data (WCFG)
30 00 20 01 FAR (Frame Address Configuration Register)
0- – – – Block Type CLB (00); Major/Minor Address

Cmd	Code	Description
WCFG	0001	**Write Configuration Data** - Used prior to writing configuration data to the FDRI. This command is followed by a FDRI configuration command which specifies the number of words to be written.
RCFG	0100	**Read Configuration Data** - Used prior to reading configuration data from the FDRO.
START	0101	**Begin Startup Sequence** - This command starts the Startup Sequence. This command is also used to start a shutdown sequence prior to partial re-configuration. The Startup Sequence begins with the next successful CRC check.
RCRC	0111	**Reset CRC** - Used to reset the CRC register. This register contains the Cyclic Redundancy Check value of the bitstream.
SWITCH	1001	**Switch CCLK Frequency** - This command is used to change the frequency of the Master CCLK.

Table 3.4: Configuration Commands

30 00 4- – FDRI (Frame Data Input Configuration Register) + Word Count

Fig. 3.14 shows a complete configuration example with an initial Shutdown Sequence, Reconfiguration of the CLB Address Space and a typical Startup Sequence.

Frame Address Register (FAR) The Frame Address Register holds the address of the current frame. This address is subdivided into three parts: First, the Block Type (CLB or RAM), second, the major address and, third, the minor address. Fig. 3.13 depicts the structure of the Frame Address Fields.

				Block Type			Major Address (Column Address)								Minor Address (Frame Address)													
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0	0

Figure 3.13: Frame Address Fields

Table 3.5 shows the two possible values for the Block Type bits in the Frame Address Fields (Fig. 3.13. The major address selects the column within the address space defined by the Block Type field. The minor address selects the frame within the column specified by the major address. If the CLB address space has been chosen, the minor address is incremented automatically each time a full data frame has been read from the FDRI. If the last frame of the current column has been read, the major address is incremented and the minor address is set to zero. This address adjustments are done automatically by the configuration state machine. However,

the major address is not incremented automatically if the RAM address space has been chosen. This means, one has to set the proper major address each time a data frame is to be written.

Block Type	Code
CLB	00
RAM	01

Table 3.5: Block Type Codes

Configuration Example Fig. 3.14 shows a complete configuration flow for a (partial) reconfiguration. In this excerpt, only the CLB address space is re-written, BlockRAM contents remain unaffected. The first 13 lines initiate a Shutdown Sequence of the target device. Then, a new configuration for the CLBs is written to the device and, in the end, the obligate Startup Sequence is issued.

3.3 Summary

This chapter does not claim to be a complete description of the configuration (and readback) flow of the Virtex™ FPGA. Interested readers are refered to [2] and [3] for further information. For example, major and minor address are calculated with specific formulae and those are defined and explained in the references. Besides, Application Notes 138 and 151 include some more examples about configuration-related stuff.

```
FFFF FFFF     Dummy Word
AA99 5566     Sync Word. Does not realign an already synchronized SelectMap port
3001 2001     COR
0080 FF2F     Shutdown bit set, optionally set bit 29 DRIVE_DONE
              Alternately COR may be read, bit 15 set, and written back
3000 8001     CMD
0000 0005     Start Shutdown
3000 8001     CMD
0000 0007     Reset CRC
0000 0000
0000 0000     Clock shutdown sequence
0000 0000
0000 0000
3000 8001     CMD
0000 0008     Assert GHIGH
3000 8001     CMD
0000 0001     Write Configuration
3000 2001     FAR
0--- ----     Block Type CLB (00); Major/Minor Address
3000 4---     FDRI + Word Count if count is <1024. Otherwise use TypeII Header.

              Write FRAME DATA
              If CLB Frames are not written to non-consecutive addresses repeat FAR word, followed
              by new Major/Minor Address followed by FDRI + Word Count and then Frame Data.

3000 0001     Write CRC
---- ----     CRC
3000 8001     CMD
0000 0003     LFRM
3000 4---     FDRI + Word Count

              Write PAD FRAME

3001 2001     COR
0080 3F2D     Default COR Options
              Alternately COR may be read, bit 15 cleared, and written back
3000 8001     CMD
0000 0005     Begin Startup
3000 0001     Write CRC
---- ----     CRC
              0000 0000
0000 0000     4 Dummy Words
0000 0000
0000 0000
```

Figure 3.14: Configuration Example (excerpt)

Chapter 4

The OS-Frame on the FPGA

This chapter provides a detailed look at the structure of the OS-Frame on the FPGA. The OS-Frame consists of several components which will be explained in this chapter. Section 4.1 treats the organization of the available chip area, the second section (4.2) describes the most important communication means being the BusMacro, section 4.3 deals with the STI (Standard Task Interface) and the corresponding PADs that have been allocated and section 4.4 introduces the tasks that have been written to test the OS-Frame.

4.1 Organization of the FPGA

As introduced in chapter, the FPGA consists of hundreds of CLB's which are distributed all over the chip area. This area can now be divided into several sub-areas. To achieve this, one can use the AREA_GROUP constraint in the constraints file.

Figure 4.1 shows the different sub-areas of the FPGA. The leftmost module is part of the OS-Frame. In this current version (V0.2), there's no logic functionality implemented in this module. The only thing it is used for, is to provide the Standard Task Interface (STI) to task_slot 0 and to route some internal signals to the PADs. The STI will be introduced in section 4.3. The module called task_slot 0 is intended for being a reconfigurable area where the tasks can be loaded into. The tasks must conform to the STI, otherwise the system doesn't work properly any more. The module in the middle is, again, part of the OS-Frame. This part implements the interface between OS-Frame and Host PC; implementation details and the protocol specification are described in section 5. Furthermore, this module provides the STI for task_slot 1. The module task_slot 1 is again reconfigurable which means, that different tasks can be loaded into this area. The right-most module is again part of the OS-Frame. There, no logic is implemented since this module only connects the internal signals to the PADs.

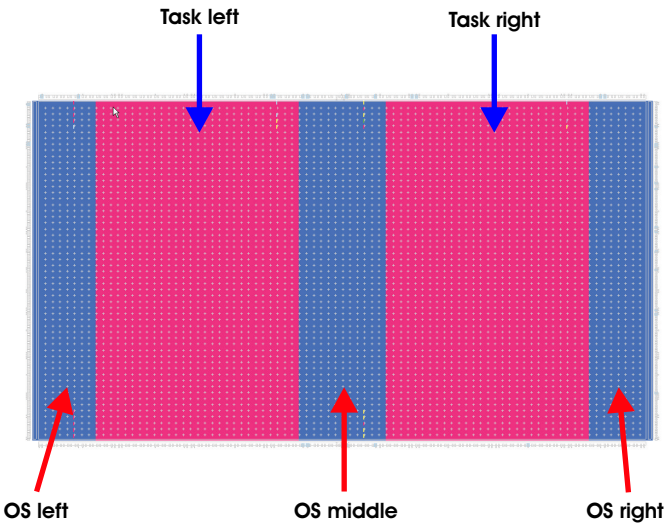

Figure 4.1: The organization on the FPGA

To divide the FPGA into these five areas, one can make use of a certain constraint, called AREA_GROUP. The syntax of this contraints is as follows:

INST *"name of instance"* **AREA_GROUP** *"name of the area group"*;
AREA_GROUP *"name of area group"* **RANGE** = *"*$CLBx_0y_0$*:*$CLBx_1y_1$*"*

With the first line, the instance *"name of instance"* of a specific module, say os_left, and all related logic is wrapped up in an AREA_GROUP with name *"name of area group"*. The second line specifies the CLB-range, where all related logic is put into, that means the range defines the boundaries of the module. The third line (see Fig. 4.2) is specific to the *Modular Design Flow* (this design flow will be introduced in chapter 6). For an example see Fig. 4.2.

```
INST “os_left” AREA_GROUP = “os_left” ;
AREA_GROUP “os_left” RANGE = “CLB_R1C1:CLB_R56C8” ;
AREA_GROUP “os_left” ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE ;
```

Figure 4.2: Usage of the AREA_GROUP constraint

Note: The corresponding logic of a module is only placed inside the module boundaries when the amount of required CLB's is smaller or equal than the number of the CLB's provided by the area group range.

4.2 Communication via the BusMacro

Imagine the following situation: You have two modules and one of those is partially reconfigurable. Furthermore, these modules want to communicate which each other.

Now, when another module is loaded into the FPGA, it is not sure, that the connection points match, that means signals may not be routed correctly across module boundaries. To prevent that fault, XILINX® has published a special macro, the BusMacro. This section is dedicated to this special macro as it is the main communication means.

4.2.1 The structure of the BusMacro

As described in chapter 3, the Virtex FPGA provides an certain amount of TriState lines (depending on the size of the FPGA, for example 216 TriState lines for the Virtex XCV800). The BusMacro uses these TriState lines to communicate across the module boundaries. Figure 4.3 shows, how the BusMacros connect to these TriState lines.

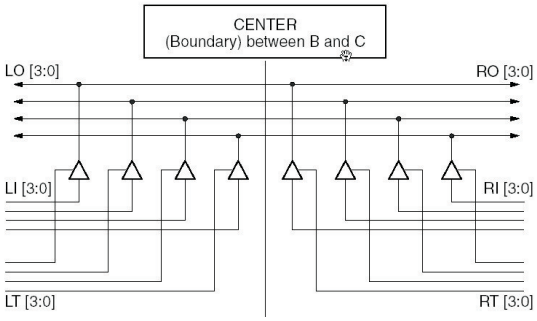

Figure 4.3: The BusMacro

As one can see in Fig. 4.3, each BusMacro uses eight TriState Buffers which provide four inputs on the left side and four inputs on the right side. Each input can be enabled seperately, but one cannot enable input l/r(0) on the left and right side at the same time, you can only enable the input 0 on either the left or the right side of the macro. The outputs are available on both sides, but are only routed to side where they are used (left input li(0) and right input ri(0) belong to output o(0), that means if one connects to input ri(0) on the right side, output o(0) is routed to the left side).

4.2.2 The Location of the BusMacros

Since this BusMacro is the point the modules connect to, it must be at certain locations on the FPGA. This means, the programmer has to assure, that they are always positioned at the same location. Again, this is done in user constraints file with a constraint called *LOC*. This constraint is applied in the following way:

INST *"name of BusMacro instantiation"* **LOC** = *"CLBx_0y_0"*;

This constraint is called *location* or *placement constraint*. With this location constraint, the BusMacro is placed at a certain position on the FPGA and this 'reference point' is never altered. Thus, the modules only need to connect to that fixed points and the communication can take place. Fig. 4.4 shows an excerpt of the user constraints file used for the OS-Frame.

```
INST “bm_task_left_inst0” LOC = “TBUF_R52C5.0” ;
```

Figure 4.4: The *location* constraint

Thus, the connections between two modules never reach the module boundaries and are, because of that, never routed unpredictably. So, the connection is established and communication can take place.

4.2.3 Common Pitfalls

In this section, the two main problems that come along when using BusMacros are explained. The first one is how secure VCC and GND signals can be achieved (4.2.3.1) and the second one is how open outputs are handled correctly (4.2.3.2).

4.2.3.1 VCC and GND: The safe method

Fig. 4.5 shows an example of how this BusMacro is used in the OS-Frame. As one can see, there are lot of GND and VCC signals required, to either enable or disable a certain input on either the left or the right side. Assuming we have two modules of which the left one is called os_left and the right one task_left. Now, if the programmer wants input 0 of the left side to be enabled, (s)he has to enable that input by assigning a logic '0' to the signal lt(0) (active low). To get this logic '0', the *Place and Route tool* provided by Xilinx sets a PAD into a PULLDOWN state and routes this signal to the BusMacro. Maybe, there are a lot of other BusMacro which require a logic '0' and because of that, the tool routes this logic '0' across the module boundaries and this won't work if someone tries to reconfigure the FPGA. To avoid this problem, one has to allocate two PADs for each module, one being a logic '0' and the second one being a logic '1'. In this case, the signals are routed inside the module area and do not cross the module boundaries. This method is safe, whereas the first method can lead to an unpredictable state.

Again, there's a certain constraint to attach a PULLDOWN, or PULLUP respectively, property to a certain PAD. Assuming that the net which delivers the BusMacro with a logic '0' is called *gnd_os_left*, the constraint looks as follows:

NET *"gnd_os_left"* **PULLDOWN ;**
NET *"gnd_os_left"* **LOC =** *"P###"* **;**

```
bm_task_left_inst0: component bm_4b
    port map (
        li(3)           => t0_reset_carrier,
        li(2)           => gnd_os_left,
        li(1)           => t0_enable_carrier,
        li(0)           => gnd_os_left,

        lt(3)           => gnd_os_left,
        lt(2)           => vcc_os_left,
        lt(1)           => gnd_os_left,
        lt(0)           => vcc_os_left,

        ri(3)           => gnd_task_left,
        ri(2)           => t0_finished_carrier,
        ri(1)           => gnd_task_left,
        ri(0)           => t0_ledbar_carrier(9),

        rt(3)           => vcc_task_left,
        rt(2)           => gnd_task_left,
        rt(1)           => vcc_task_left,
        rt(0)           => gnd_task_left,

        o(3)            => t0_reset_from_bm,
        o(2)            => t0_finished_from_bm,
        o(1)            => t0_enable_from_bm,
        o(0)            => t0_ledbar_from_bm(9)
    );
```

Figure 4.5: Instantiation of a BusMacro

The first line attaches the PULLDOWN to a net and the second line defines the PAD this signal is coming from. PULLUP's are generated in the same way (just write PULLUP instead of PULLDOWN).

4.2.3.2 Handling of Open Outputs

The second problem that comes along with the usage of the BusMacro are open outputs: If a certain input is not used on neither the left nor the right side of the BusMacro, an open output will result. The programmer is not allowed to leave this output open, since the Modular Design Tools won't handle this circumstance properly. Thus, the implementation of the modules will fail. To avoid this failure, there exists a certain technique which handles this problem properly. This work-around can be described as follows: Let's say, output o(3) of a certain BusMacro is open, that means it has no input on neither the left nor the right side. Thus, this ouput is not needed and the output signal must be *patched*. To achieve this goal, one needs two signals with the first one connected to the open output o(3) and the second one being the patching signal. The patch is done in a process which simply assigns the first signal to the second one at each rising (or active) clock edge. Since the Xilinx Synthesis Tool (XST) would remove the signals and the flipflop generated by this process (they don't have any meaning for the XST), the programmer has to make sure, that this optimization step doesn't occur. This can be done with the usage of the **keep** attribute. The keep attribute must be applied to the signal that is connected to the open output. For an example, see Fig. 4.6.

```
– attribute declaration
attribute keep : string ;

– the required signals
signal input_connect, input_patch : std_logic ;

– apply the attribute to the signals
attribute keep of input_connect : signal is “true” ;
attribute keep of input_patch : signal is “true” ;

... – other VHDL code

patcher: process(clock, input_connect)
begin
      if(clock’event and clock = ‘1’) then
            input_patch <= input_connect ;
      end if ;
end process patcher ;
```

Figure 4.6: The 'open-output-workaround'

4.3 The Standard Task Interface (STI)

In this section, the Standard Task Interface (STI) of the current OS-Frame V0.2 is introduced. The STI defines, which resources can be accessed by a task.

4.3.1 Why defining an STI

Chapter 3 has introduced the XILINX® Virtex Architecture. It's obvious, that the available resources on the chip are limited according to the size of the chip. These resources have to be shared between OS-Frame and the tasks. Thus, the signals from the PADs to the tasks cannot be routed freely. We've figured out several possibilities to keep the system as scalable as possible. The first possibility consisted of a routing matrix (crossbar-switch) that would have provided a highly scalable system, since each single input could have been connected to a certain output. In fact, this routing matrix has used too much resources of the available chip area (for example, a 32x32 matrix has used 50% of the area). Thus, we've thrown away this possibility.

The second possibility looked as follows: The number of in- and outputs of a task were limited to a certain amount, let's say 10 inputs and 10 outputs, and the PADs (the amount and their location) were fixed, too. That means we allocated certain PADs and the tasks could only connect to these PADs. The appropriate outputs of the different tasks were driven into a multiplexer, for example output 0 of task 0 and task 1 were driven into one multiplexer, and the output of the multiplexer was connected to the PAD. The inputs were handeled in the same manner, that means the signal coming from the PAD was connected to the multiplexer and the outputs of the multiplexer were routed to the tasks. Thus, this solution consisted of 20 multiplexers, 10 of them with two inputs and one output the other 10 of them with one input and two outputs respectively (see Fig. 4.7).

There were two main reasons why we've thrown away this design: First, during the design phase of the project, more than two tasks were foreseen and that would

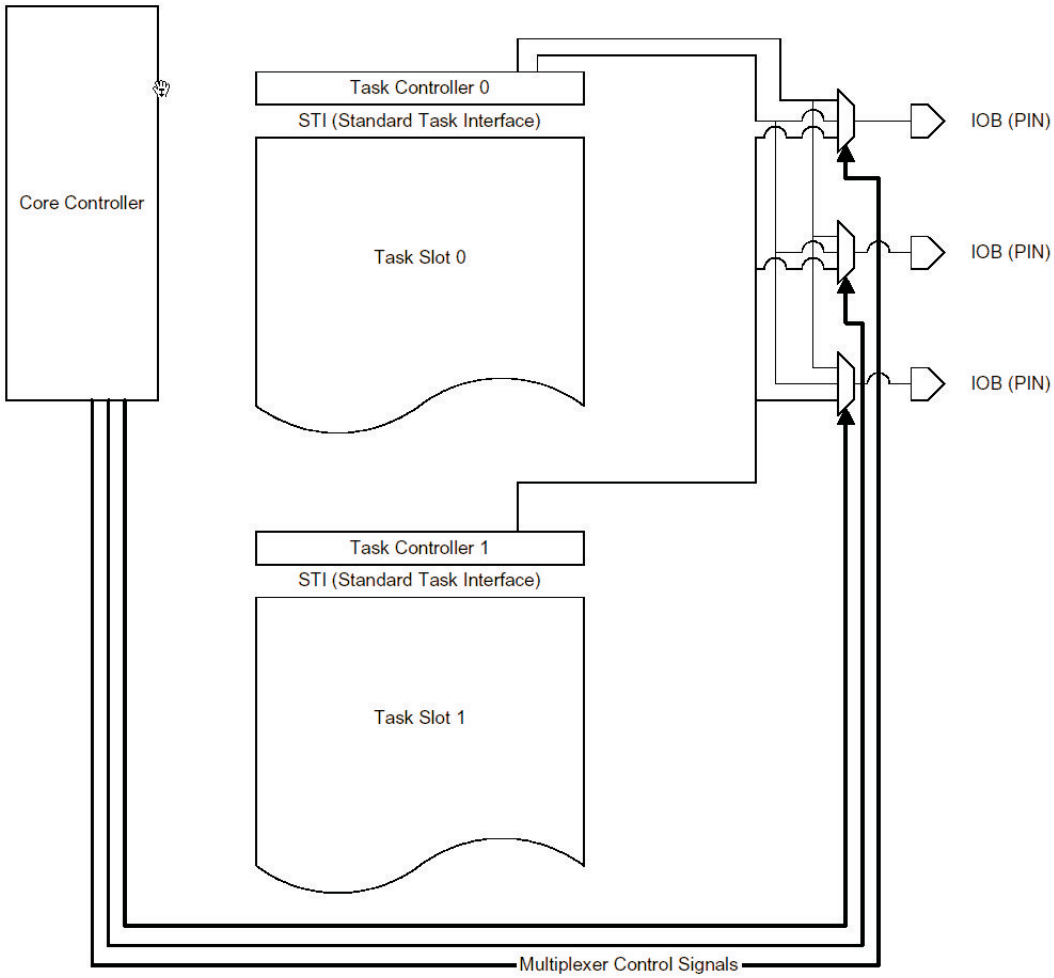

Figure 4.7: The multiplexer design

have led into a large routing overhead and, second, all these multiplexers have to be set when a new task is loaded into the OS-Frame and, thus, there's a remarquable configuration overhead.

The design we've chosen is introduced in the next section. It is a simplified variant of the second possibility.

4.3.2 The current Standard Task Interface (STI)

The Standart Task Interface of the current OS-Frame Version 0.2 defines the following outputs: 10 Ledbar outputs, the Codec Audio outputs and the RS232 TX output. Furthermore, it defines the following inputs: The Codec Audio input and the RS232 RX input (see Fig. 4.8).

The ledbar outputs are shared between both tasks, that means task left and task

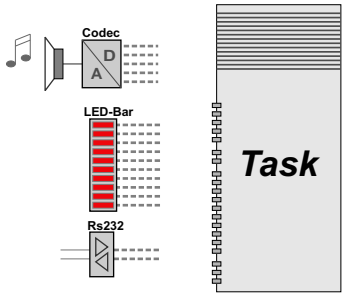

Figure 4.8: The in- and outputs of the STI

right can use the ledbar outputs at same time (this doesn't lead to any problems). The other outputs are restricted, that means only one task, either the left or the right one, is allowed to use the Codec Audio or the RS232 TX outputs. For example, if task left is "talking" to the Codec Audio chip, the right task is not allowed to "talk" to that chip at the same time. This resource management is done on the Host-PC. The software on the Host-PC (introduced in chapter 5) controls the whole OS-Frame and decides, whether a task is allowed to be loaded into a certain slot or not, depending on the resources the current task in the other slot is using. Since this resource management is done on the Host-PC, the signals of task left and task right are simply ORed before they are routed to the PADs (see Fig. 4.9). These circumstances lead to a easier hardware but, on the other hand, to a more complicated software.

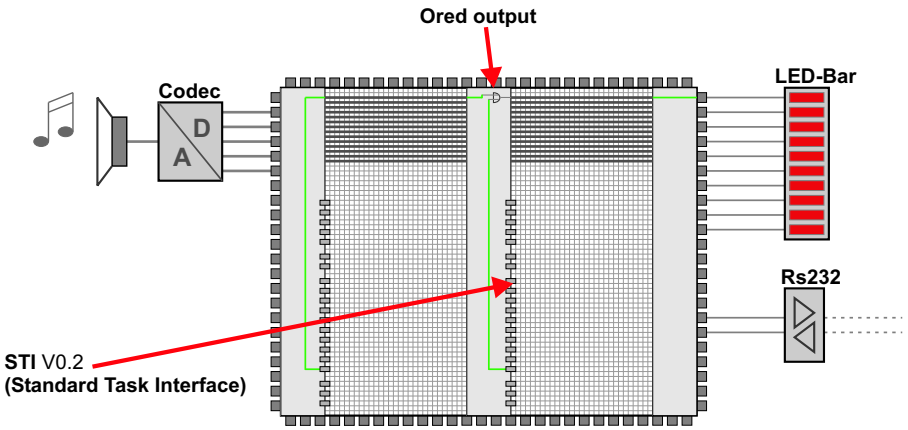

Figure 4.9: Signal routing on the FPGA

Apart from the general purpose I/O described so far, there are two control signals: t_reset and t_enable. The first one is used the reset the task during runtime and the second one can be used to either enable or disable the clock signal. The t_finished output is used by the OS-Frame. During runtime, t_finished should be '0' and as soon as a task has finished its work, t_finished must be asserted high. This

is registered by the application on the Host-PC and the task can be replaced by another one.

4.4 The Tasks

This section introduces the tasks that have been written in order to test the functionality of the OS-Frame. There are eight different tasks which can be downloaded onto the FPGA. As you can imagine, they are not very complicated, since the purpose is merely to test the OS-Frame and this can be done with quite simple task.

KnightRider and KnightRider single These two tasks require the 10-segment bargraph LED. They produce the following light pattern: LED 0 is illuminated and all other LEDs are dark. Then, the second LED is illuminated and all other (including LED 0) LEDs are dark and so on. When the top has been reached, it produces the same pattern from top to bottom. The only difference between these two tasks is, that the KnightRider task never stops voluntarily, whereas the KnightRider single task once goes from bottom to top and back again. Having done so, the t_finished signal is set to high and the task has finished.

Audio Low, Audio Low single, Audio High and Audio High single These four tasks require the Audio Codec resource. They generate a data streams which can be heard as a single tone coming out of the speakers. The difference between the Audio Low and Audio High tasks is that the frequency of the generated is different (the latter ones generated a tone with a higher frequency). The difference between Audio Low/High and Audio Low/High single is that Audio Low/High single generates a short beep, whereas Audio Low/High never comes to an end. Again, when Audio Low single has come to its end, it asserts high the t_finished signal.

RS232 single This tasks writes the string "Hello World" to the RS232 interface with baud-rate of 115200. Having done this, the task finishes and asserts t_finished high.

Up-Down single This task generates another light pattern on the 10-segment bargraph LED. The generated pattern represents 10 bits of common up-counter. When the counter has reached its maximum value, it starts counting down and the same 10 bits are displayed on the bargraph. When the task has finished its work (the count value has reached zero again), the t_finished signal is asserted and the task supsends.

Chapter 5

The Software of the OS-Frame

This chapter introduces the software that runs on the Host-PC in order to control the hardware that runs on the FPGA. The application is written in C++, since the PCI7200 is delivered with a C++ interface (header files and the corresponding library). This chapter is subdivided into the following sections: Section 5.1 shows what actions the application performs, section 5.2 provides detailed information about the implemenation and, finally, section 5.3 shows the way the Host-PC and the hardware communicate with each other.

5.1 The Application

The application has a variety of features in order to provide full control over the hardware part of the OS-Frame on the FPGA. Remember, the application is part of the OS-Frame, too, since the Hardware OS-Frame consists of hard- and software parts. This section gives an overview of the actions that can be performed with this application.

5.1.1 The GUI of the Application

Fig. 5.1 shows the main window of the application. As one can see, there are a few checkboxes and buttons which let the user interact with the OS-Frame. When the application is started, the hardware part of the OS-Frame is downloaded to the FPGA. This initial configuration contains the PC-Interface (5.3) and two dummy-tasks which don't do anything. When the initial configuration has been downloaded, the OS-Frame is ready.

After downloading the bitstream, the are two operation modes provided by the application: First, the application automatically controls the hardware by scheduling a new task when one of the currently running tasks has finished and, second, the user can manually select a task for downloading. When automatic scheduling

Figure 5.1: The Main-Window of the Application

is enabled, the application automatically selects a task which does not cause any resource conflicts.

Another important part of the application is the ability to dynamically add new tasks to the Task-Pool during runtime. When the user issues the 'Add New Task' command, a new window appears. This window is shown in Fig. 5.2. As one can see, there are a few values that must be specified when adding a new task to the Task-Pool: The paths of the bitfiles (one for the left and one for the right slot, the name of the task, then the initial values for the reset and enable signals and, finally, the resources the current tasks uses during runtime.

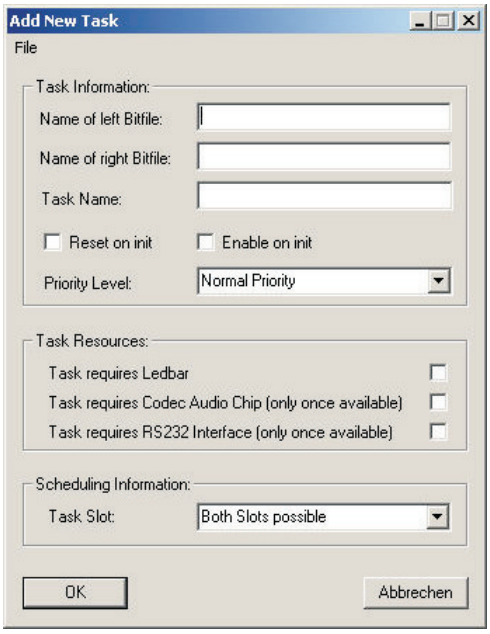

Figure 5.2: The 'Add New Task'-Window

5.1.2 The Task-Pool and The Scheduler/Resource-Manager

Basically, the Task-Pool consists of a linked list of available Tasks. It is subdivided into two queues: One High-Priority queue and one Normal-Priority queue. Inside the queue, the tasks are maintained in a double-linked list of *Task* objects (see 5.2). There are two reasons for which the scheduler is invoked: First, one of the running tasks has finished its work and, second, the user wants to download a new task. The second case merely invokes the Resource-Manager, but since the Resource-Manager and the scheduler have a narrow relationship they are considered the same in this context.

5.1.2.1 The Scheduler

The scheduler is invoked prior to downloading a new task onto the FPGA. It chooses one task of the Task-Pool for downloading (in the case of automatic scheduling) and sends it to the FPGA or, in the case of a user action, checks whether resource-conflicts may occur and if not, downloads the chosen task onto the chip. Basically, the scheduler performs the following actions:

1. First, it checks if the left task_slot is empty. If so, it sets a flag to indicate that a new task may be downloaded. If the left task_slot is not empty, the scheduler checks the state of the left task and if its status is TASK_FINISHED, sets

a flag to indicate that a new task could be downloaded. If the status is not TASK_FINISHED, the left task_slot remains unaffected.

2. The same as in 1. for the right task_slot.
3. It selects the first TaskQueue for searching for new tasks (in the current implementation the first queue is the HIGH_PRIORITY TaskQueue.
4. If the left task_slot has been chosen in 1. for receiving a new task, the scheduler invokes the Resource-Manager that looks for a feasible task in the current TaskQueue. If one has been found, this task is downloaded to the FPGA. If no task has been selected, a second flag is set to indicate that the left task_slot has not received a new task so far.
5. The same as in 4. for the right task_slot.
6. If both task_slots have received a new task or no more TaskQueues are available, the scheduler finishes its work. If one (or even both) task_slot(s) is not satisfied, the scheduler chooses the next TaskQueue, in this case this is the NORMAL_PRIORITY TaskQueue, and goes again to 4.

5.1.2.2 The Resource-Manager

The work the Resource-Manager has to do can be explained in the following way: Imagine, the left task_slot contains a task that has some allocated resources. If the right task_slot is to receive a new task, the resources of the left (running) task are checked against the resources the right (waiting) task would allocate. If there results any conflict, the task may not be downloaded onto the FPGA.

Currently, there are three resources that can be allocated by any task. These are the 10-segment bargraph LED, the Audio Codec and the RS232 Interface. Table 5.1 shows where resource conflicts may occur: If the resource is declared 'Shared', there are no consequences when downloading the new task. Otherwise, if the resource is declared 'Restricted', the second task may not be downloaded since the resources are explicitly assigned to one running task.

Resource	Mode
10-segment bargraph LED	Shared
Audio Codec	Restricted
RS232 Interface	Restricted

Table 5.1: The Resources on the FPGA

The 10-segment bargraph LED can be used by two tasks at the same time. Thus, this resource is declared 'Shared'. Since the Audio Codec cannot process two serial data streams at the same, this resource must be 'Restricted'. The RS232 Interface is not allowed to be driven by two different sources at the same time and, thus, this resource is 'Restricted', too.

5.2 Implementation Details

This section is intended to give a detailed overview of the implemented classes. Basically, there are three classes that have been implemented during this thesis: class **Task**, class **TaskQueue** and class **TaskManager**. Each of these classes is going to be handled separately in this section.

5.2.1 Class Task

Each time a task enters the application, a class Task is allocated for this specific task. This class contains all the information related to that certain task: The paths of the bitfiles (one for the left task_slot and one for the right task_slot), the name of the task, the resources this task requires, its status .... Table 5.2 describes the elements shown in the code excerpt below:

```
class Task
{
public:
        Task();
        virtual ~Task();
        Task(...);

        bool        operator==(Task b);
        struct Resources {
                bool        ledbar;
                bool        codec;
                bool        rs232;
        };
        struct Resources TaskResources;
        void         setPriority(int);
        int          getPriority();
        void         setStatus(int);
        int          getStatus();
        void         setPathLeft(char *);
        void         setPathRight(char *);
        char*        getPathLeft();
        char*        getPathRight();
        void         setName(char *);
        char*        getName();
        Task*        next;
        Task*        prev;
        bool         reset;
        bool         enable;
        bool         loadable;
        // the slot(s) the task can be loaded into
        int          slot;
private:
        int          priority;
```

```
        int         status;

        char        path_left[255];
        char        path_right[255];
        char        name[255];
        bool        check();
};
```

Field	Description
reset	Gives the initial value for the reset signal. This value is asserted immediately after downloading the task.
enable	Gives the initial value for the enable signal. This value is asserted immediately after downloading the task.
loadable	This flag indicates whether the task can be downloaded or not.
slot	This field indicates the possible slot(s) the task can be loaded into. Possible values are: BOTH_SLOTS, LEFT_SLOT and RIGHT_SLOT.
priority	This value holds the priority of the task and defines in which TaskQueue this task is enqueued. Possible values are: HIGH_PRIORITY and NORMAL_PRIORITY.
status	This integer stores the current status of the task. Possible values are: TASK_WAITING (for being scheduled), TASK_READY (to be scheduled), TASK_RUNNING and TASK_FINISHED.
path_left	This string holds the path of the bitfile for the left task_slot.
path_right	This string holds the path of the bitfile for the right task_slot.
name	This string identifies the task by its name.

Table 5.2: The fields of class Task

There are a set of functions provided by this class Task, too. These functions are used the alter the information (or fields) of the current task. For example, **SetStatus** alters the status value of the current task.

There are two pointers inside the class: **next** and **prev**. These pointers are used to implement a double-linked list, called a TaskQueue (see next subsection).

5.2.2 Class TaskQueue

As mentioned above, the application maintains a Task-Pool where all available task are put into. This Task-Pool itself consists of one or more TaskQueues with different priorities. In the current application, two TaskQueues are supported, one HIGH_PRIORITY and one NORMAL_PRIORITY queue. Tasks that have a HIGH_PRIORITY are enqueued into the first TaskQueue and tasks with a lower, or 'normal', priority are enqueued in the latter one. The code inserted below shows the API of the class **TaskQueue**:

```
class TaskQueue
{
public:
        TaskQueue();
        virtual ~TaskQueue();
        int                  enqueue(Task *);
        Task*                dequeue();

        // for the scheduler
        Task*                getNextFeasibleTask(Task *, int);
        void                 setPriority(int);
        int                  getPriority();
        bool                 queueContains(Task *);
        void                 deleteTaskFromQueue(Task *, bool);

        Task*                getTaskByName(char *);

        int                  getCount();
        // only for debugging purpose
        void                 printQueue();
        TaskQueue            *prev, *next;
private:
        bool                 checkFeasibility(Task *, Task *, int);
        Task*                first;
        Task*                last;
        int                  priority;

        int                  count;
};
```

Field	Description
priority	This value is set to either set to HIGH_PRIORITY or NORMAL_PRIORITY. When a task is to be enqueued into the current queue, the task's priority is compared against the priority of the current queue. If this comparison fails, the task is not enqueued and an error is reported.
count	This values indicates the number of tasks that are enqueued in the current queue.

Table 5.3: The fields of class TaskQueue

This API provides some functions, that are used by the scheduler. These are the following:

getNextFeasibleTask Imagine the following situation: The scheduler has been invoked and it notices, that the task in the left slot has finished its work and can be replaced by another one whereas the task in the right slot is still running. The task in the right slot has some resources assigned to it. In this

example, getNextFeasibleTask takes the following two parameters: a pointer to the task that is currently running in the right slot and an integer that specifies the slot the feasible task should be loaded into (in this case, the left slot). getNextFeasibleTask goes through the list of tasks and compares the resources of each entry to the resources of the task in the right slot. If one entry has been found that does not cause any resource conflict with the currently running task in the right slot, this task is returned. If no task could have been found, NULL is returned.

getTaskByName This function takes a string parameter which holds the name of the task that should be looked for. This function goes through the list and compares the name value of each entry to the parameter. If one of the entries matches, the matching task is returned, otherwise, NULL is returned.

checkFeasibility This function takes three arguments: Two tasks and a integer value which depicts a slot. The two tasks are checked against each other for possible resource conflicts. If there is such a conflict, the function returns FALSE (this indicates, that the tasks cannot run in parallel). Otherwise, if no conflicts may occur, the slot value of the first task argument (see Table 5.2 is examined: If the task can be loaded into both slots (BOTH_SLOTS) the function returns TRUE and everything is ok. Otherwise, if the slot value of the first task parameter (either LEFT_SLOT or RIGHT_SLOT) does not match the integer parameter, the function returns FALSE. This function is invoked every time a user manually chooses a task to be downloaded (see Fig. 5.1). For example, if the left task_slot is occupied by a task that requires the Audio Codec, then the user is not allowed to choose a task for the right task_slot that would require the same resource.

5.2.3 Class TaskManager

This API is something like the control-center of the Task-Pool. It maintains a variety of TaskQueues, flags and states. This API implements the scheduling algorithm described in 5.1.2.1. New tasks can be added to or removed from the Task-Pool via this API. The code inserted below shows the definiton of the API and Table 5.4 describes the most important fields in detail:

```
class TaskManager
{
public:
        TaskManager(void *, int);
        virtual ~TaskManager();

        void                addNewTask(...);
        int                 requeueTask(Task *);
        void                delTask(Task *);
        void                delTask(char *);
```

```
        void                removeTaskFromQueue(Task *);
        void                schedule();
        void                reloadFinishedTasks();
        void                readTaskStates(int, int);
        void                loadTask(char *, int);

        Task*               searchTaskByName(char *);
private:
        bool                checkFeasibility(Task *, Task *, int);
        Task*               running_left;
        Task*               running_right;
        // the next queues are used for scheduling purpose
        TaskQueue*          ready;
        TaskQueue*          finished;
        TaskQueue*          waiting;        // not used yet!!!

        int                 nofReadyQueues;
        // store the pointer to the application
        void*               app;
        int                 PCI_Card_ID;
};
```

Field	Description
running_left	This fields points to the task that is currently running in the left task_slot. If no task is running inside the left slot, this value is set to NULL.
running_right	This field points to the task that is currently running in the right task_slot. Again, this value is set to NULL if no task is currently running in the right slot.
ready	This value points to the first TaskQueue that holds some task with status TASK_READY. In the current version, the pointer is set to the HIGH_PRIORITY TaskQueue. The HIGH_PRIORITY TaskQueue then points to the next TaskQueue, in this version the NORMAL_PRIORITY TaskQueue. Thus, with this field, all TaskQueues that contain TASK_READY tasks can be reached.
finished	If one task has finished its work, the scheduler is invoked. The scheduler then puts the finished task into the finished-TaskQueue. Finished task are, thus, not put into the ready queue immediately, they are first enqueued in the finished-TaskQueue.
waiting	This field is not used in the current version.

Table 5.4: The fields of class TaskManager

Again, this API provides a set of functions to manage the Task-Pool. The ap-

plication should take actions on the Task-Pool only via the TaskManager API. The most important functions are described below:

addNewTask This function requires quite a lot of arguments. When the user wants to add a new task to the Task-Pool, the window in Fig. 5.2 appears and the user must specify the task's properties. If the user then confirms his values, addNewTask is invoked. This function creates a new class Task object and sets the parameters specified by the user. Finally, the new task is enqueued into the proper ready queue and can be scheduled.

schedule This function implements the scheduling algorithm described above (see 5.1.2.1).

reloadFinishedTasks This function empties the finished queue and the enqueues the tasks in the corresponding ready queue again.

readTaskStates This function is invoked periodically by a timer. Since the PCI7200 I/O Card does not support interrupt handling, the task states must be read manually. This function reads the states of both tasks and, when one task has finished its work, changes the status of that task to TASK_FINISHED. Each time, a task has finished its work and this is detected by readTaskStates, the scheduler is invoked in order to replace the finished task by a new one.

loadTask This function takes two arguments: The first one is the name of the task that should be loaded and the second one indicates the slot this task should be loaded into. When the resources have been checked, this function starts the partial reconfiguration of the FPGA (see 3) and loads the task.

5.2.4 Summary

The scheduler is implemented in one single function that follows the algorithm depicted in 5.1.2.1. The Resource-Manager is splitted into two functions: getNextFeasibleTask (invoked by the scheduler) implemented in the TaskQueue API and checkFeasibility(...) implemented in the TaskManager API. This has been implemented this way in order to distinct between the two operation modes of the application: the automatic and manual scheduling. Automatic scheduling demands proper queue handling and, thus, the Resource-Manager should be part of the queues, whereas manual scheduling does not require the queueing mechanisms of the TaskManager API.

5.3 Communication between Host-PC and Hardware: The PC-Interface

The PC-Interface has been implemented to establish the communication between the Host-PC and the FPGA. It works as follows: The application sends a command to the FPGA (and optional data) and the PC-Interface on the FPGA interprets this command and performs the desired action. Currently, two commands are supported (see 5.3.2. The next subsection describes the in- and outputs and their specific meaning.

5.3.1 The In- and Outputs of the PC-Interface

First, let's have a look at the Inputs and Outputs of the PC-Interface. Fig. 5.3 depicts the inputs (on the left hand) and outputs (on the right hand). The meaning of each signal is described in detail in table 5.5.

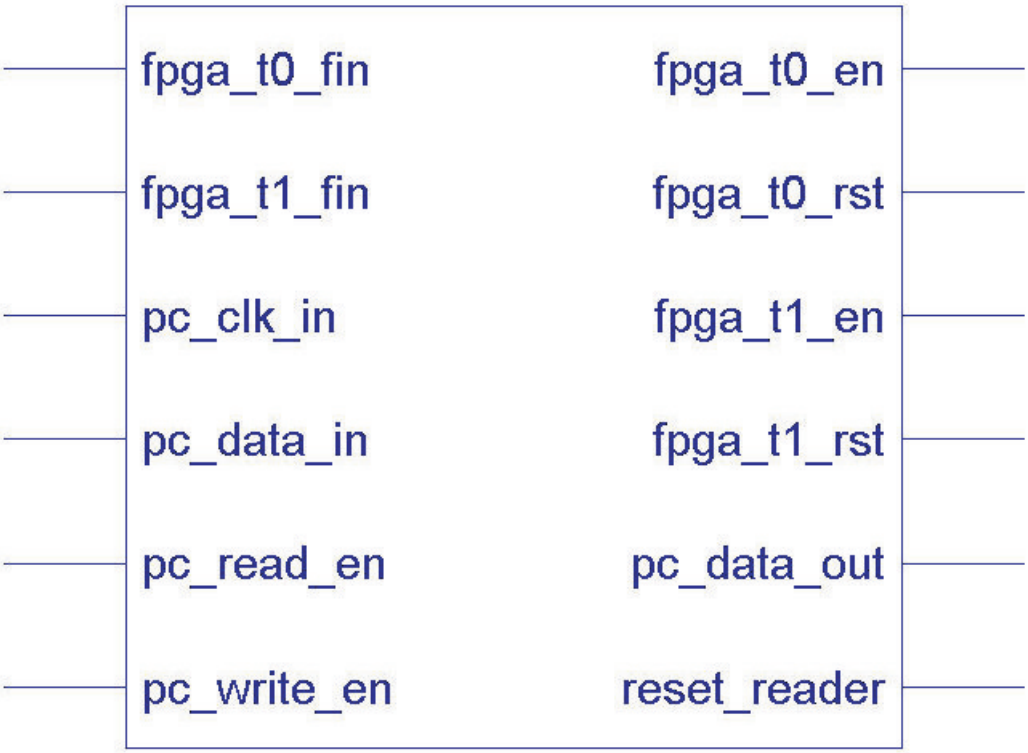

Figure 5.3: The Inputs (on the left hand) and Outputs (on the right hand) of the PC-Interface

The PC-Interface consists of six inputs and six outputs. The inputs fpga_t0_fin and fpga_t1_fin come from the tasks and are, thus, internal signals (the t_finished

Signal	Description
fpga_t0_fin	This input indicates whether the task in the left task_slot has finished its work or not.
fpga_t1_fin	This input indicates whether the task in the right task_slot has finished its work or not.
pc_clk_in	This is the Master Clock of the PC-Interface. Each data transmission is synchronized to this clock. This signal is generated by the Host-PC.
pc_data_in	This is the data line from the Host-PC to the FPGA. All data is transmitted serially and synchronous to the Master Clock.
pc_read_en	Read-Enable is asserted high by the Host-PC, when data is read from the Host-PC.
pc_write_en	Write-Enable is asserted high by the Host-PC, when data is written from the Host-PC to the FPGA.
fpga_t0_en	This is the enable signal of the STI for task left.
fpga_t0_rst	This is the reset signal of the STI for task left.
fpga_t1_en	This is the enable signal of the STI for task right.
fpga_t1_rst	This is the reset signal of the STI for task right.
pc_data_out	This is data from the FPGA to the Host-PC. All data is transitted serially and synchronous to the Master Clock.
reset_reader	This signal is described later in this text.

Table 5.5: The Inputs and Outputs of the PC-Interface

signal of the left task is connected to fpga_t0_fin and the t_finished signal of the right task is connected to fpga_t1_fin). The following signals come from the Host-PC: pc_clk_in, pc_data_in, pc_read_en and pc_write_en. Table 5.6 shows this inputs signals and their corresponding FPGA-PADs.

Signal	FPGA Pin	Expansion Header Pin
pc_clk_in	86	D12 (right)
pc_data_in	93	D14 (right)
pc_read_en	209	D6 (left)
pc_write_en	87	D13 (right)

Table 5.6: The inputs of the PC-Interface

The six outputs are divided into external and internal signals, too. There's only one external signal, namely pc_data_out. Table 5.7 shows the PIN assignment for this external signal. The other outputs are internal signals. The signals fpga_t0_en, fpga_t0_rst, fpga_t1_en and fpga_t1_rst satisfy the STI in the meaning that they deliver the tasks with the control signals t_enable and t_reset (t0 corresponds to

the left task and t1 corresponds to the right task).

Signal	FPGA Pin	Expansion Header Pin
pc_data_out	94	D15 (right)

Table 5.7: The output of the PC-Interface

5.3.2 The Control Commands

The current version of the PC-Interface supports two commands: First, a 'Config' command and, second, a 'ResetReader' command. A command consists of 8-bit wide code. The codes for the two supported commands are depicted in table 5.8.

Command	8-bit Code
Config	00000001
ResetReader	00000010

Table 5.8: The supported commands and their corresponding 8-bit Code

5.3.2.1 The 'Config' Command

The 'Config' command has been implemented to set the values for the control signals, t_enable and t_reset, of the tasks. A new task-configuration is written to the FPGA in the following way:

1. Assert pc_write_en high prior to a rising clock edge of the pc_clk_in.
2. At each following active clock edge, one bit is read. Note: The first bit is already read when the pc_write_en signal is asserted high.
3. When the command code 00000001 is read, write the task-configuration bits to the FPGA (Note: pc_write_en remains high.
4. When 8 bits are read, de-assert pc_write_en. After an additional clock cycle, the new task-configuration is passed to the internal circuitry of the OS-Frame and the tasks obtain the new configuration.

Fig. 5.4 shows the bitstream that has to be written sequentially to the FPGA in order to change the task-configuration. The outputs fpga_t0_en, fpga_t0_rst, fpga_t1_en and fpga_t1_rst are then set corresponding to the configuration bits of the bitstream. Table 5.9 shows how the control signals are generated out of the configuration bits of the bitstream.

The Enable control signals are assigned immediately, whereas the Reset control signals are generated out of two configuration bits. The Reset control signal is an

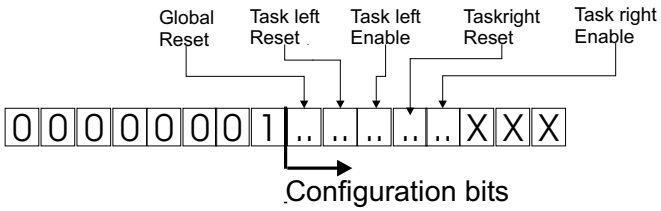

Figure 5.4: The Structure of a 'Config' Command

Control Signal	Generation
fpga_t0_en	"Task left Enable"
fpga_t0_rst	"Global Reset" AND "Task left Reset"
fpga_t1_en	"Task right Enable"
fpga_t1_rst	"Global Reset" AND "Task right Reset"

Table 5.9: Generation of the Control Signals out of the configuration bits

asynchronous, active low reset. Thus, both configuration bits, Global Reset and Task left/right Reset must be '1' in order the de-assert the Reset control signal (that's why the corresponding configuration bits are ANDed).

Fig. 5.5 shows a 'Config' command sequence. As one can see, the first bit is written at the same time the pc_write_en signal is asserted high (it is '1', since the bits must be written in the reverse order, that means LSB first). Then, seven '0's follow and after the command header, the configuration bits follow (again 8 bits, however the last 3 bits are unused). Then, pc_write_en is de-asserted and with the last clock cycle, the new configuration is published to the FPGA circuitry.

Figure 5.5: A typical 'Config' command sequence

5.3.2.2 The 'ResetReader' Command

The second command that is supported by the PC-Interface is the 'ResetReader' command. When one of the tasks finishes its work, it publishes this by asserting high the t_finished control signal of the STI. When the PC-Interface detects a high finished signal, it informs the Host-PC about the new situation with the pc_event signal. The Host-PC then reads both task states and, thus, finds out which task has finished its work. The reading of the task states is done in the following way:

1. Assert pc_read_en high.
2. Read one bit at each active clock edge of the pc_clk_in Master Clock. This happens twice, once for the t_finished signal of task left and once for the t_finished signal of task right.
3. De-assert pc_read_en.

Fig. 5.6 depicts the header for the 'ResetReader' command. As shown in table 5.8, this command consists of an 8-bit wide code: 00000010. Fig. 5.7 shows the timing diagram for this command: First, pc_write_en is asserted high and the first bit of the command code is applied. Then, for the next seven clock cycles, each bit is read serially. Then, after reading the 8th bit, pc_write_en is de-asserted and as the result of this command, the reset_reader output of the PC-Interface goes high. The last, 9th, clock cycle finishes the 'ResetReader' command and pulls the reset_reader ouptut down again.

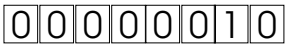

Figure 5.6: The Structure of the 'ResetReader' command Header

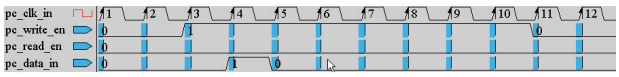

Figure 5.7: The Timing-Diagram for the 'ResetReader' Command

The reset_reader output must be high for one clock cycle (from the 8th to th 9th) in order to reset the pc_event signal that has previously informed the Host-PC about the finishing of a task. This must be done each time a task finishes its work. If this was not done immediately, no other events could be traced and this would result in serious functional bugs.

5.4 Summary

This chapter has described the software that has been written for the OS-Frame. Furthermore, the in- and outputs of the PC-Interface as well as the supported commands have been introduced.

Chapter 6

The Modular Design Flow

This chapter covers a special VHDL/Verilog design flow, called Modular Design. This design flow has some advantages over the common XST VHDL/Verilog design flow: Modular Design provides independent work to the developers of a group. Each programmer can develop his module independent from other developers, and, in the end of the design phasis, the different modules can be merged into one, full design which provides the desired functionality. Furthermore, Modular Design offers a quite simple way to make a module partially reconfigurable and that's the main reason, why we've chosen this special design flow. The first section covers Modular Design Entry and Synthesis, the second section deals with Modular Design Implementation, the third section introduces the specialities one has to take into account when programming modules for partial reconfiguration, the last section provides a cookbook for this Design Flow illustrated with an example.

6.1 Modular Design Entry and Synthesis

Again, imagine a group of developers who want to build a hardware system that provides a certain functionality. This system can now be divided into several pieces, called 'modules', and then be merged into one FPGA design. Each developer can independentely work on 'his' module without affecting the work of other members of the team. In the beginning of the design flow, the team leader defines a top level design. This top level design specifies the in- and outputs of the whole system, the number of modules the system is seperated into (including the interfaces of the different modules) and the way these modules are interconnected. Modules are instantiated as 'black boxes' defined by their interface.

Then, the developers create the modules by using either VHDL or Verilog and synthesize their code to obtain a netlist. The netlist of each module is needed for the further Modular Design flow steps.

6.2 Modular Design Implementation

Modular Design Implementation includes three steps which are described in this section. The first step is called Initial Budgeting Phase (6.2.1), Active Module Implementation (6.2.2) and Assembly Phase (6.2.3). At the end of the last phase, a bitstream which can be used to program the FPGA is generated.

6.2.1 Initial Budgeting Phase

In this phase, the team leader has to do the following work:

- Position all global logic
- Size and position each module
- Position the input and output ports

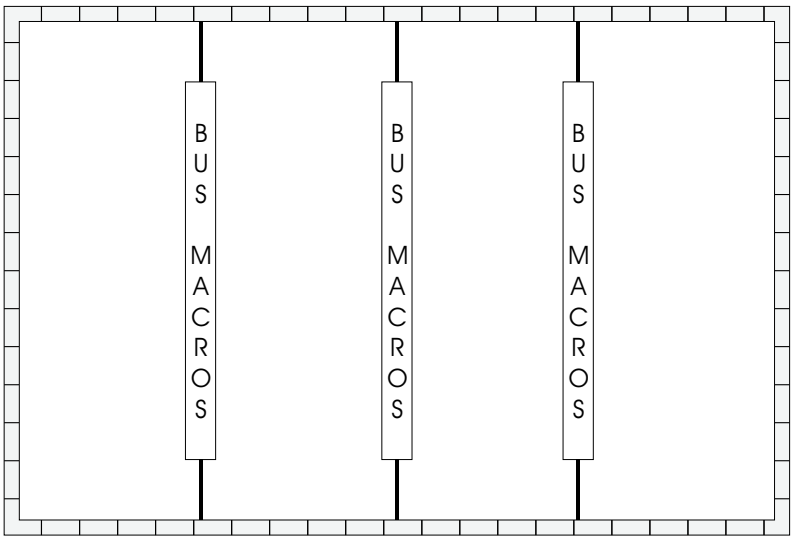

Figure 6.1: The Initial Budgeting Phase

Global logic is implemented in the top-level HDL file and should, hence, be positioned in the global *User Constraints File*. The size and the position of each module are parts of the top-level User Constraints File, as well. The input and output ports are 'glued' together with PAD's in the top-level .ucf file, again.

Having synthesized the top-level logic, this netlist has to be translated into a Xilinx file format using the following command:

ngdbuild -modular initial design_name.ngc

The files, that should be in the directory to perform this translation, are the following: a netlist (either an EDIF netlist or a NGC netlist) and the top-level User Constraints File which contains the positions of all global logic, the positions and sizes of the modules and the PIN assignments. See Fig. 6.2 for a sample User Constraints File containing an example of a module positioning and sizing and a PIN assignment.

```
– module positioning
INST "task_left_inst" AREA_GROUP = "task_left";
AREA_GROUP "task_left" RANGE = "CLB_R1C9:CLB_R56C36";
AREA_GROUP "task_left" ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE;

... – position all modules

– the positions of all busmacros

INST "bm_task_left_inst0" LOC = "TBUF_R52C5.0";
INST "bm_task_left_inst1" LOC = "TBUF_R53C5.0";
INST "bm_task_left_inst2" LOC = "TBUF_R54C5.0";
INST "bm_task_left_inst3" LOC = "TBUF_R55C5.0";
INST "bm_task_left_inst4" LOC = "TBUF_R56C5.0";

... – other busmacros

– PIN assignment

NET "rs232_tx" LOC = "P144";
NET "rs232_rx" LOC = "P141";

... – all PIN assignments
```

Figure 6.2: Sample User Constraints File (excerpt)

After issuing the command mentioned above, the Initial Budgeting Phase is finished and the team can start with the next phase called Active Module Implementation.

6.2.2 Active Module Implementation

During this phase, the developers implement the top-level design with only the "active" module expanded. Active refers to the module the teams is currently working on. There are some files needed to perform Active Module Implementation: The netlist (either EDIF netlist or NGC netlist) of the active module and the top-level User Constraints File[1]. To translate your active module, issue the following command:

ngdbuild -modular module -active module_name
top_level_design_directory_path/design_name.ngd

[1] In the Xilinx Development System Reference Guide, see [6], it is recommended that the top-level UCF is renamend from *top-level*.ucf into *module-name*.ucf, but we've figured out, that the translation works without renaming the UCF, too. In the context of this report, renaming the UCF is not applied and the showns commands are changed accordingly.

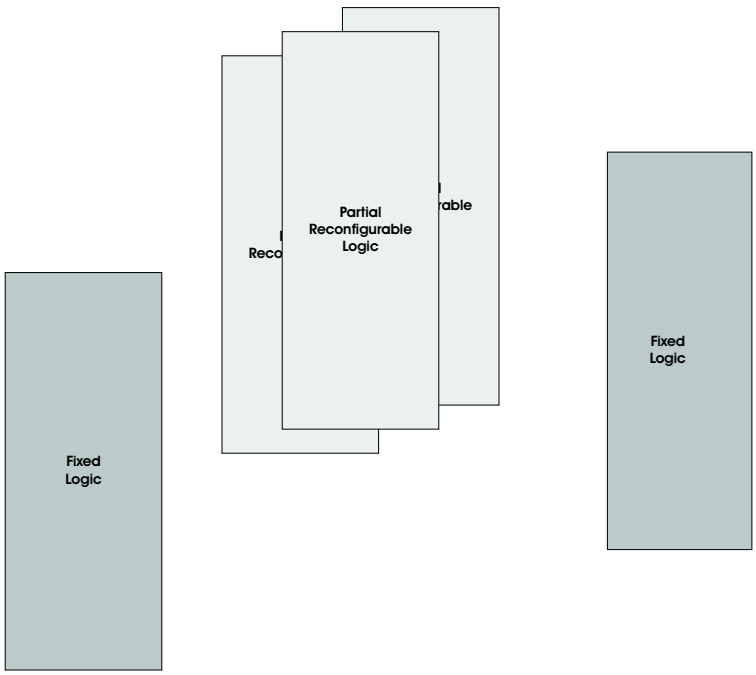

Figure 6.3: The Active Module Implementation Phase

The next step one has to perform is to issue the next command. This command maps the logic of the top-level design with the active module expanded:

map design_name.ngd

The mapped design is stored in the file called *design_name.ncd*. The next step the developers have to do is placing and routing the design. This is done by issuing the following command:

par -w design_name.ncd design_name_routed.ncd

The target file is stored in *design_name_routed.ncd* and, hence, does not overwrite the mapped design. The -w option ensures that any previous version of the *design_name_routed.ncd* are replaced by the new design.

Running the TRACE command on the implemented design verifies whether the top-level timing constraints are met or not. Issue the following command:

trce design_name**_routed.ncd**

As a last step, the implemented module file has to be published to centrally local PIMs directory set up by the team leader:

pimcreate pim_directory_path **-ncd** design_name_**routed.ncd**

This command copies the implemented module files (NGO, NGM and NCD) into the specified PIMs directory. The PIMs directory is used for Assembly Phase of the Modular Design Flow. The Assembly Phase merges all active modules into one final design which can be used to generate a bitstream. More details on this Assembly Phase are introduced during the next subsection.

6.2.3 Assembling the Modules

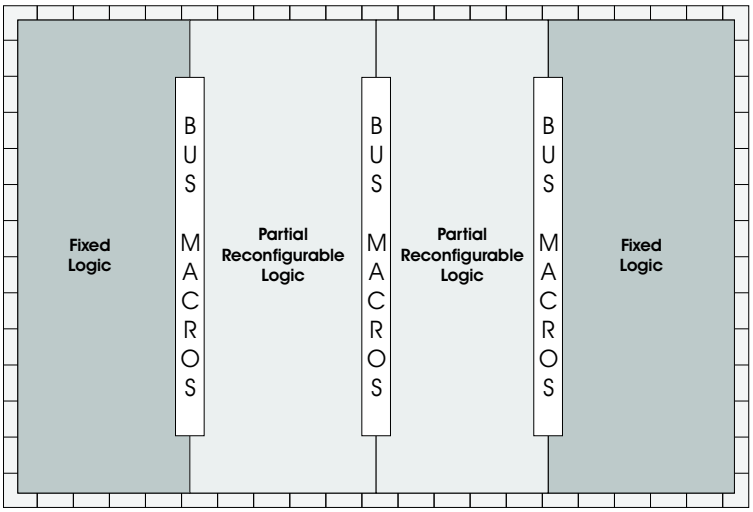

Figure 6.4: The Assembly Phase

In this last phase of the Modular Design Flow, the team leader assembles the previously implemented modules into one design. To do so, one has to change to the top-level design directory (/top/assemble) and, again, translate the whole design with the following command:

ngdbuild -modular assemble -pimpath pim_directory_path design_name.ngd

Ngdbuild generates and NGD file from the top-level UCF file (must be in this directory, too), the top-level design's NGD file and each PIM's NGD file. The next step, the team leader has to perform is to map the logic of the full design:

map design_name.ngd

Then, the whole design must be placed and routed by issuing the following command:

par -w design_name.**ncd** design_name_**routed.ncd**

The are no Modular Design specific options required, since the information is encoded in each PIM's NCD file. To chech the timing constraint (if applied), the team leader has to invoke the TRACE command:

trce design_name_**routed.ncd**

Then, implementation is finished and the design is ready to be translated into a Virtex conform bitstream which then can be downloaded to the device. The bitstream is generated by issuing the following command:

bitgen -f bitgen.ut design_name_routed.ncd

This command needs some special options that are written inside *bitgen.ut*. Fig. 6.5 shows the options file we've used for the OS-Frame.

```
-w
-l
-m
-g ReadBack
-g DebugBitstream:No
-g ConfigRate:4
-g CclkPin:PullUp
-g M0Pin:PullUp
-g M1Pin:PullUp
-g M2Pin:PullUp
-g ProgPin:PullUp
-g DonePin:PullUp
-g DriveDone:No
-g PowerdownPin:PullUp
-g TckPin:PullUp
-g TdiPin:PullUp
-g TdoPin:PullNone
-g TmsPin:PullUp
-g UnusedPin:PullDown
-g UserID:0xFFFFFFFF
-g StartUpClk:Cclk
-g DONE_cycle:4
-g GTS_cycle:5
-g GWE_cycle:6
-g GSR_cycle:6
-g LCK_cycle:NoWait
-g Security:None
-g Persist:Yes
-g DonePipe:No
-g Binary:no
```

Figure 6.5: Options file for the OS-Frame

6.3 Modules for Partial Reconfiguration

This section shows the specialities the programmer has to account for. Otherwise, Module Implementation will fail:

Module Width should always be $n \times 4$ (this is because of the BusMacro) CLBs wide. The size of the module is defined by the AREA_GROUP constraint specified in the User Constraints File.

Module Height Since the smallest configuration unit is the frame and frames always extend the full height of the device, all modules should extend the full height of the device, too. Again, this is specified in the User Constraints File.

IOBs The IOBs on the top and the bottom are part of the module. All adjacent IOBs are reconfigured when a module is downloaded.

These are the most important guidelines that always should be followed in order to obtain a working circuitry. The first two points show that the area that is to be reconfigured must be a rectangular with one side being $n \times 4$ CLBs long and the the other side extending the full height of the device.

6.4 Modular Design: A Cookbook

This section is intended to provide a cookbook for the Modular Design Flow. With the steps below, modules for partial reconfiguration can be obtained (NOTE: "Normal" modules can be implemented in the same way, however, the commands are slightly different.[2]).

6.4.1 Preparations

Before starting with Modular Design Entry and Synthesis (see above), the directory structure depicted in Fig. 6.6 should be created.

hdl This directory contains all source code (in this case VHDL code).

ise This directory contains the project files (ISE project) of the top level logic.

modules This directory only contains sub-directories, one per module.

pims This directory contains the files required for assembling the full design. **pim-create** automatically creates the sub-directories and copies the recommended files of each module in its corresponding sub-directory.

top This directory, again, only contains two sub-directories called **initial** and **assemble**. The 'initial' sub-directory is used for the Initial Budgeting and contains the top-level files required for the translation (the **top-level**.ngc and the **top-level**.ucf User Constraints File). The 'assemble' sub-directory contains the same files as the 'initial' sub-directory.

[2]The standard design flow is depicted in the "Development System Reference Guide-ISE 5" provided by XILINX®

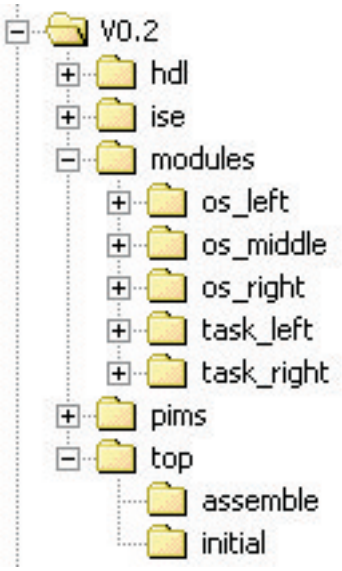

Figure 6.6: The recommended directory structure

6.4.2 Example Flow: The Initial Budgeting, Active Module Implementation and the Assembling Phases

As an example, we take two modules called *module_1* and *module_2* and the top-level design called *top*. Fig. 6.7 shows the resulting directory structure.

The 'hdl' directory contains the following files: *module_1*.vhd, *module_2*.vhd and *top*.vhd. These are the source code files. The 'ise' directory contains the XILINX ISE project called *top*. The next step is the synthesis of the top-level design. This top-level design contains all global logic and two component instantiations, called *module_1_inst* and *module_2_inst*. These two modules communicate with each other. As depicted in chapter 4, to guarantee the communication the BusMacro has to be used. In this example, there is one BusMacro required for the communication between the modules. Fig. 6.8 shows the User Constraints File for this example. It contains the positions and sizes of the modules, the BusMacro location and a few PIN assignments.

The modules are both 16 CLBs wide and extend the full height of the Virtex XCV800 FPGA (56 rows). The BusMacro is placed in the middle of the modules. Since the BusMacro requires 8 TBUFs while four of them belong to the left module and four of them belong to the right module, and the location of the BusMacro is specified by the position of the leftmost TBUF, we have to substract 4 (because of the four TBUFs belonging to the left module) of the right frontier of the left module (in this example, the right frontier is the 16th CLB column) and this calculates to 13 (16 included). Thus, the TBUF column is the 13th. The row needn't be calculated, it can be freely chosen, for example 5. Thus, the location is specified by

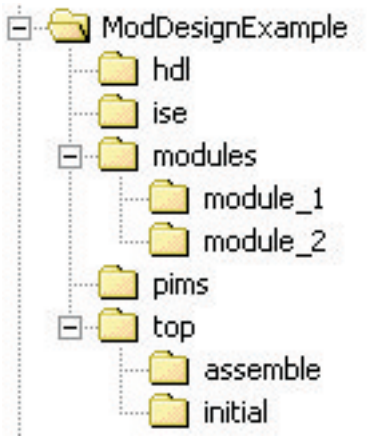

Figure 6.7: The recommended directory structure for the example

```
– The sample User Constraints File

– Size and positions of the modules
INST “module_1_inst” AREA_GROUP = “module_1_group” ;
AREA_GROUP “module_1_group” RANGE = “CLB_R1C1:CLB_R56C16” ;
AREA_GROUP “module_1_group” ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE ;

INST “module_2_inst” AREA_GROUP = “module_2_group” ;
AREA_GROUP “module_2_group” RANGE = “CLB_R1C17:CLB_R56C32” ;
AREA_GROUP “module_2_group” ROUTE_AREA = RECONFIG DISALLOW_BOUNDARY_CROSSING RECONFIG_MODE ;

– BusMacro location
INST “bus_macro_inst” LOC = “TBUF_R5C13.0” ;

– PIN assignment
NET “sample_signal” LOC = “Pxxx” ;
...
```

Figure 6.8: The User Constraints File for the Example

TBUF_R5C13**.0** (the **.0** comes because of the two slices, and two TBUFs, a CLB consists of. **.0** chooses the left TBUF).

When the top-level design is synthesized, the source code files of the modules mustn't be included in the project. This has the effect, that the components, or modules, are treated as black boxes and remain unexpanded. Having synthesized the top-level design, the *top.ngc* has to be copied in the /top/initial directory. With this .ngc file, the User Constraints File *top.ucf* and the BusMacro file *bm_4b.nmc* (provided on the XILINX® homepage) the translation of the top-level design can be done with the following command:

ngdbuild -p xcv800-hq240-5 -modular **initial top.ngc**

By issueing this command, the Initial Budgeting Phase is complete. The next phase is the Active Module Implementation. First, copy *top.ucf* and *bm_4b.nmc* into both module sub-directories. Furthermore, the options file *bitgen_par.ut* depicted in Fig. 6.5 must be copied into each sub-directory. Then, both modules have

to be synthesized seperately and the resulting two files *module_1.ngc* and *module_2.ngc* must be copied into their appropriate module sub-directory. Then, to implement the module, the following command sequence has to be issued (NOTE: The implementation of module_2 is done in the same way):

ngdbuild -p xcv800-hq240-5 -modular module -active **module_1**
..\..\top\initial\top.ngc
map -pr b top.ngd -o top_map.ncd top.pcf
par -w -ol 5 -n 1 -s 1 top_map.ncd mppr.dir top.pcf
copy mppr.dir\5_5_1.ncd top.ncd
bitgen -d -f **bitgen_par.ut** -g ActiveReconfig:yes top.ncd
trce top.ncd top.pcf
pimcreate -ncd top.ncd -ngm top_map.ngm ..\..\pims

Having done this, the Active Module Implementation phase is finished. The bitfiles for partial reconfiguration are in the modules' sub-directories and called *top.bit*. They can now be renamed into a more appropriate one. Thus, the last step, the Assembly phase, can take place. To do so, first copy the following files into the /top/assemble directory: *top.ngc*, *top.ucf*, *bm_4b.nmc* and *bitgen_par.ut*. Having copied these files, the following command sequence must be performed:

ngdbuild -p xcv800-hq240-5 -modular **assemble**
-pimpath ..\..\pims **top.ngc**
map -pr b top.ngd -o top_map.ncd top.pcf
par -w top_map.ncd top.ncd top.pcf
bitgen -f bitgen_par.ut top.ncd
trce top.ncd top.pcf

When these commands have been worked through, the top-level bitfile is created and ready to be downloaded on the device. The bitfile is called *top.bit* and for a Virtex XCV800 about 576 KBytes big, whereas the bitfiles of the modules are considerably smaller (about 50-100 KBytes, depending on the size of the module).

Chapter 7

The Testbed, Conclusions and Related Work

This chapter gives a summary of the achieved goals. The first section shows how the Host-PC and the XESS Board were interconnected during the tests and the second section draws some conclusions.

7.1 The Testbed

Fig. 7.1 depicts the testbed that has been used during this thesis. First, there's the Host-PC with the running application and, second, there's the XESS Prototyping Board (see chapter 2). The Host-PC and the XESS Board are connected to each other. The PCI7200 I/O Card provides the main in- and outputs on the PC's side: the Parallel Port of the XESS Board and the Controlling Signals to the Expansion Headers are feeded with signals coming from (or going to) this PCI7200 I/O Card. The serial port of the Host-PC is connected to the serial port of the XESS Board directly (directly means, that these signals do not pass the PCI7200 I/O Interface).

7.2 Conclusions

With the application described in chapter 5, the OS-Frame and the Tasks (described in chapter 4) the tests could have begun. The tests have revealed that the application runs stable, that means the software has never crashed. Furthermore, the two modes (manual and automatic scheduling) do their corresponding work properly. This implies that the Resource-Manager detects resource-conflicts whenever they occur and handles them properly.

The main goal of this thesis was to implement and test a Hardware OS-Frame and this has been reached. This thesis shows that it is possible to implement a

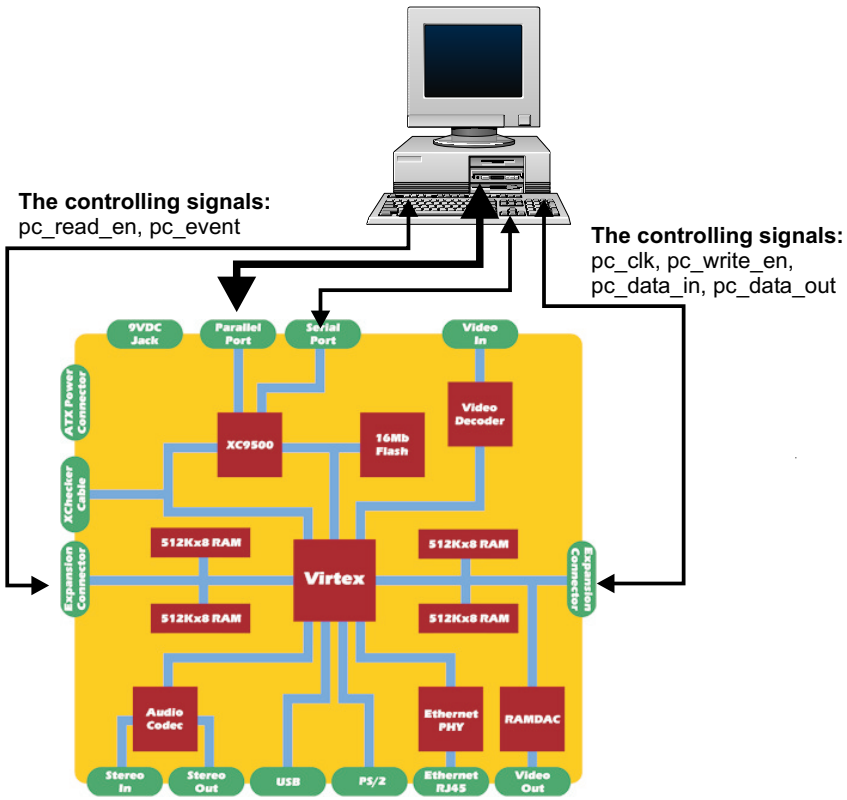

Figure 7.1: The Testbed

general platform for Hardware Tasks when accepting some restrictions. The restrictions come from the XESS Board and the limited resources such an FPGA provides. With the introduction of the Standard Task Interface (STI), task programmers are bound to the in- and output limitations this STI brings along.

7.3 Related Work

The current OS-Frame only supports stand-alone Tasks. That means, the Tasks cannot communicate with each other. In further versions, this InterTask-Communication should be implemented with FIFO-Buffers or similar techniques[1]. Apart from that, InterTask-Communication can also be implemented in software, but this would only satisfy non-time-critical communication.

By using a different Development Board, some severe restrictions that come along when using the XESS Board could be eliminated (it would be best to develop a board which is custom-made to the demands of such a Hardware OS-Frame).

[1] This has already been implemented in another diploma thesis (see [10]). They have exented the STI and implemented the FIFO-Buffer approach

Acknowledgements

I would like to thank a couple of people who have made this diploma thesis possible. First, I would like to thank my advisor Herbert Walder for the many hours of introduction to this research topic, for the valuable discussions on the concepts and for helping me with VHDL and C++ programming difficulties. Second, I would like to thank Matthias Dyer for his additional help on VHDL coding. I would like to thank Andres Erni and Stefan Reichmuth for the coorporation with the definition of the STI and, last but not least, I would like to thank Michael Lerjen for his previous work. He has implemented the main C++ routines for configuring the FPGA via the PCI I/O card.

Finally, I would like to thank Prof. Dr. Lothar Thiele and the Computer Engineering and Networks Laboratory (TIK) for providing the premises and materials used during the thesis. Thank you all.

Bibliography

[1] XSV Board Manual, V1.1 9/21/2001

[2] Virtex FPGA Series Configuration and Readback, XILINX® XAPP138 v2.3, October 4, 2000

[3] Virtex FPGA Series Configuration Architecture User Guide, XILINX® XAPP151 v1.5, September 27, 2000

[4] Two Flows for Partial Reconfiguration: Module Based or Small Bit Manipulations, XILINX® XAPP290 v1.0, May 17, 2002

[5] Virtex™ 2.5 V Field Programmable Gate Arrays, XILINX® DS003-2, v2.7, July 19, 2000

[6] Development System Reference Guide, © Copyright 1994-2002 Xilinx, Inc.

[7] Michael Lerjen, Chris Zbinden: Reconfigurable Bluetooth/Ethernet Bridge

[8] Matthias Dyer et al: Partially Reconfigurable Cores for Xilinx Virtex

[9] Matthias Dyer, Marco Wirz: Reconfigurable System on an FPGA

[10] Andres Erni, Stefan Reichmuth: Inter-Task-Communication in Reconfigurable Operating Systems