Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Samuel Nobs

Reconfigurable Hardware OS Prototype - Part CPU

Masters Thesis DA-2004-04
Winter Term 2003/2004

Tutor: Herbert Walder

Supervisor:
Prof. Dr. Lothar Thiele

30.4.2004

XFBOARD

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Preface

This text is the report of the development work done in the framework of my master's thesis at the Computer Engineering and Networks Lab (TIK) at the ETH Zürich. The duration of this thesis was 6 months including report generation. This time span allows for a deep insight into the topic at hand and a lot of work to be done. Therefore this document is rather lengthy, however, it does not include all details.

The subject of this thesis was the development of an operating system that could be used with reconfigurable hardware based on FPGAs. This development both includes software and hardware design. Reconfigurable hardware operating systems are a new field of research and a subgroup of the subject matter *Embedded Systems*.

Chapter 1 of this report offers insight into the background and motivation for this thesis. The goals to be achieved are mentioned also there.

A *very* short discussion of related work is done in **chapter 2**, looking both at the hardware and software approaches.

An overview of the system and its components is given in **chapter 3**. Both the hardware and software components are mentioned and placed in the big picture there.

Chapter 4 explains the scheduling approach used in the system to allow for time sharing of the CPU. A description of the processes and their statuses is given there too.

The usage of the various memory technologies on the board is discussed in **chapter 5**. Memory allocation and protection are two other topics of this chapter.

OS elements that are not complex enough to be dedicated their own chapters are compiled in **chapter 6**: the user interface, the OS bridge and the configuration of the second FPGA in the system.

The documentation of all hardware developed for this system is made available in **chapter 7**, a detailed explanation of the software is given in **chapter 8**.

The knowledge needed to start as fast as possible developing your own hard- and software is transferred in **chapter 9**.

A rich appendix finally gives you additional background information to on the system at hand.

Throughout this text, you may encounter the following signs which are meant to give additional information to the user or any developer continuing my work:

Notes point out things that might be interesting to explore further.

Warnings indicate potential pitfalls and errors which should be avoided by any means.

Samuel Nobs
Zürich, April 30, 2004

Contents

List of Figures

List of Tables

1 Introduction

In this introduction, the background and motivation for this work is mentioned. The assignment of this thesis is discussed too, and some notes on the development environment are given.

1.1 Background and Motivation

Silicon process technologies used for FPGA design have been constantly improved over years: layout densities and clock frequencies have been significantly increased and, in the meantime, reached a level where a 32 bit CPU including controllers and peripherals fits into such an FPGA without even using all available resources. For example, the XILINX Virtex-II family is built on a 0.15 micron, 8-layer metal process with high speed 0.12 micron transistors. The largest device in this family, the XC2V8000, offers 8 million system gates[1] in 23'296 configurable logic blocks (CLB), and the highest speed grade is suited for clock frequencies above 200 MHz.

Furthermore, current FPGA technologies allow for partial reconfiguration. This allows a device being altered in certain areas at runtime, leaving other areas untouched. So FPGAs are in a position now to be used as dynamically allocatable resources. Computationally complex hardware tasks that might be inefficiently treated by a general purpose microprocessor can be implemented in dedicated hardware and loaded or unloaded on demand, boosting performance by orders of magnitude if the time lapse needed for (re-)configuration can be kept short.

The special forms of resource allocation needed in the above-mentioned application of FPGAs ask for a reconfigurable hardware operating system (RHWOS) providing an abstraction from the underlying technology by offering services like device drivers for I/O components (Ethernet, Audio), doing the bookkeeping about free user space on the FPGA and assign this space to HW-Tasks to be loaded, and

[1] System gates are a combination of logic, memory, and custom circuit resources that would be utilized in a typical design. This term is used as a measure of FPGA density

managing task requests to internal (block RAM, FiFos)and external memory. For more details about RHWOS' you might want to refer to [23].

Configurable boards with FPGAs and CPLDs have become an important means for rapid prototyping and system development. A huge number of manufacturers (XESS, BURCHED, XESYS, MEMEC DESIGN, SUNDANCE, etc.) are exploiting this market and offer a broad range of such prototype boards. None of the products found in this range fulfills the special board architecture demands of a RHWOS.

Due to the limitation of partial reconfiguration of the XILINX FPGAs to chip scanning[2] and the desired topology of the architecture inside the FPGA, all I/O devices should be connected to an OS frame on the left and the right side of the FPGA. These OS frames will be left untouched during the reconfiguration process, so the connections to the I/O devices will persist. None of the commercially available boards respects this constraint.

To maximise the usability of such a board, all I/O devices and memory modules should be addressable independently, a requirement that is not consequently met on the board currently at hand, the XSV Board by XESS Corp.[25]: e.g. to use the LED bar, you have do disable the FlashRAM on the board.

That is where the XFBOARD[10][19][20] comes into play, which has been designed to avoid the disadvantages being inherent to all other boards available so far. This board, being the development platform for this thesis, is being looked at closer in section 1.3.

Once the hardware is done, an operating system (OS) doing the bookkeeping about the resources available on the reconfigurable entity and performing the scheduling of the tasks is needed. Such operating systems exist, but as this type of OS' need to be perfectly adapted to the underlying architecture, it seems to be more promising and instructional to design an OS from scratch than to dig into the depths of an existing OS and adapt it to the architecture. Additionally, these OS' are generally oversized due to their huge amount of features.

1.2 Thesis Assignment

In this thesis, I was advised to figure out a concept for a RHWOS that runs on the XFBOARD. Elements and services made available by the CPU and its peripherals should be implemented. The software representing the OS kernel must be designed. These are the subtasks to be performed:

- Define which type of applications should be realized on this platform. Which are the key data of the applications? How do they behave?
- Specify the RHWOS services. How do they cooperate? Operating sequence: power-on-reset → system and OS boot → application start → user interaction → task management ...
- Perform a partitioning of the OS-Elements. What is going to be implemented in hardware, what in software?

[2] Column-wise reconfiguration

- Specify and design system and OS elements. Define interfaces an the communication between CPU and the reconfigurable resource.

- Evaluate the performance of applications and the operating system.

1.3 Development Platform

The hardware platform used for the development process is the XFBOARD I designed in a term thesis[10]. Due to the fact that I was the one that made this hardware, I have enough in-depth knowledge about the hardware to make the OS tightly fit into the constraints imposed by the components present on the XFBOARD. The Board consists of two FPGAs, peripherals and memory modules. A block diagram of the board can be seen in figure 1-1, a photograph is provided in figure 1-2.

1.3.1 FPGAs

Instead of a standard CPU (e.g. ARM, PIC, MCore), a XILINX Virtex-II XC2V1000 FPGA is present to implement a CPU soft core on it. This brings in much more flexibility since these cores can be fine tuned, and their detailed implementation may evolve during the lifetime of the board. Using a soft CPU, the hardware drivers can be designed on-chip. XILINX offers a 32 bit RISC processor that fits on the FPGA, the μBlaze soft processor core [27]. The CPU will be implemented on this FPGA to execute the software part of the OS.

The reconfigurable resource on the board is represented by a XILINX Virtex-II XC2V3000 FPGA, which is called R-FPGA. This FPGA is offered the same 50 MHz clock as the CPU FPGA (C-FPGA) can derive 4 additional clock signals from the system clock, which are fed to the R-FPGA.

1.3.2 Peripherals

For the communication from the external host to the FGPAs, mainly used for the transmission of configuration data (C-FPGA) and for streaming and networking applications (R-FPGA), two 100 Mbps fast ethernet transceivers with an RJ45 connector are installed.

To download the bit stream to the configuration PROM, a JTAG test access port is present. The C-FPGA is also connected to this JTAG chain to allow for an emergency configuration if all other means failed. For the same reason, the R-FPGA is accessible through a separate JTAG port.

A general purpose I/O (GPIO) bus is provided to enable communication between the two FPGAs. This GPIO is 40 bits in width, e.g. for 32 bit data and 8 control signals. Additional 32 bits, the optional general purpose I/O (OGPIO), may be used if partial reconfiguration of the R-FPGA is not needed. Moreover, two 16 pin expansion headers are connected to the OGPIO bus that may be used by the C-FPGA even when partial reconfiguration of the R-FPGA is used.

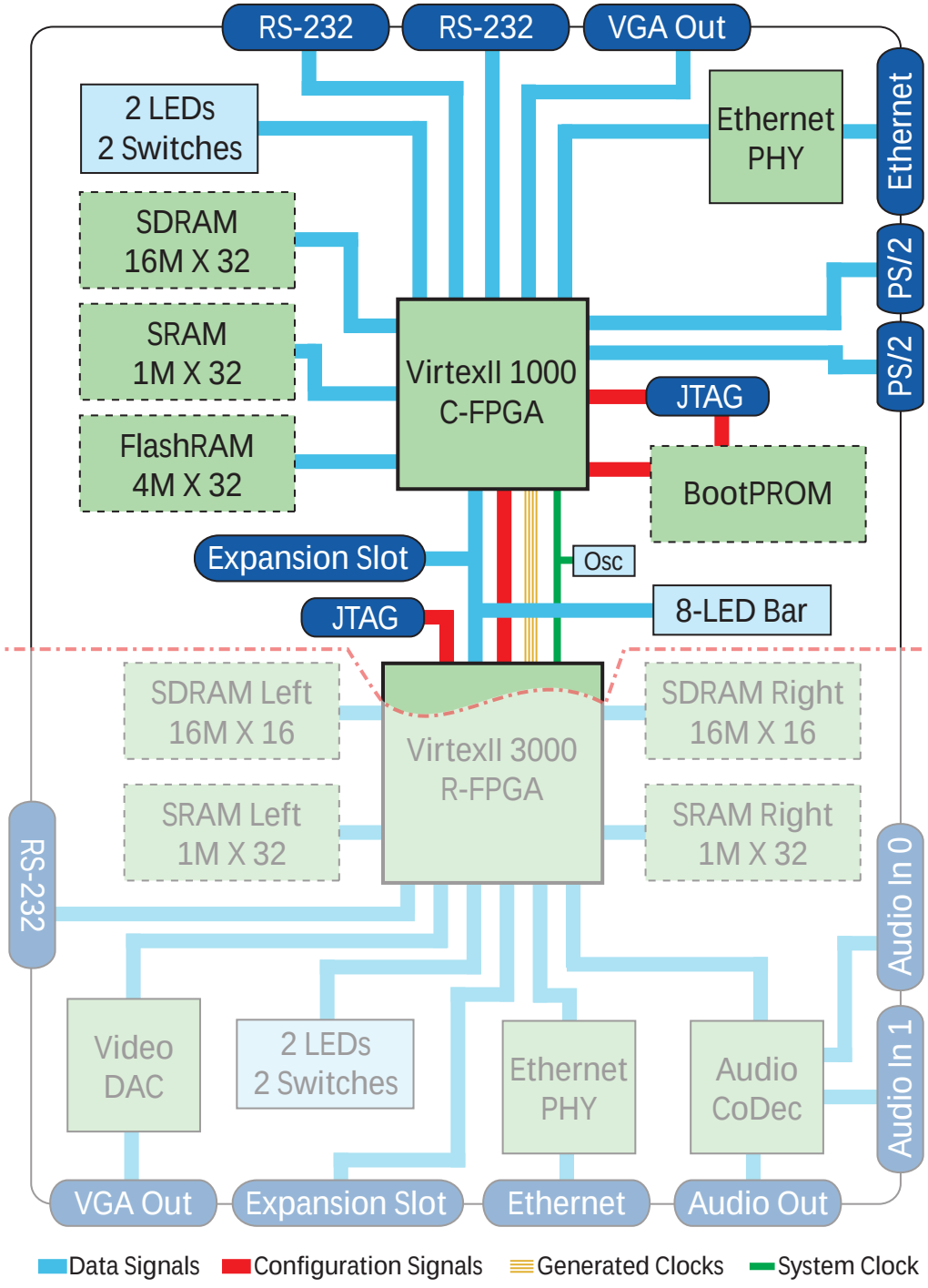


Figure 1-1: Block Diagram of the XF-Board. *The components not being subject of this work are below the dash-dotted line and drawn in pale colours.*

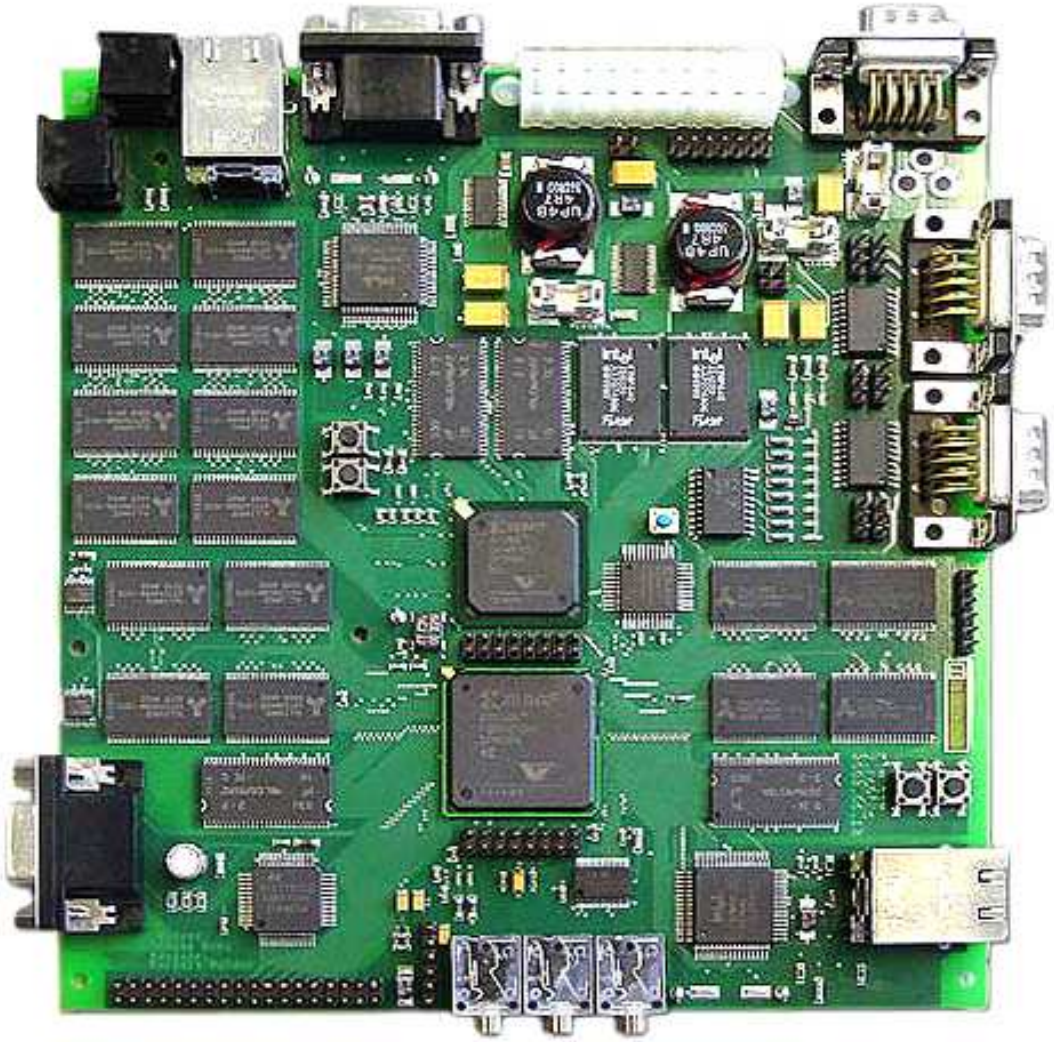


***Figure 1-2: Photo of the XF-Board**. The C-FPGA is in the center of the board, the R-FPGA below. Most of the board is covered with RAM modules of different technologies. The board is 174mm × 160mm in size.*

The following debugging channels are provided: for a very basic input and output, two push-buttons and two LEDs are attached to each FPGA. For advanced CPU debugging, two PS/2 connectors for a mouse and a keyboard and a simple 8 color VGA Output are connected to the C-FPGA. For debugging using a host PC, two RS-232 ports can be accessed using the C-FPGA, and an additional port is accessible via the R-FPGA. An 8-LED bar monitors the level of 8 out of the 40 GPIO signals; this LED bar may be used for visual feedback. The two 16-pin headers connected to the OGPIO are available for debugging purposes, too.

For video and audio applications, the R-FPGA is equipped with a video DAC being capable of displaying 24 bit true-color on a VGA monitor, and an audio CoDec featuring two inputs and one output.

Temperature sensors to monitor the core temperature of both FPGAs are installed, and a 36 pin header is provided to extend the R-FPGA by custom peripherals.

Power can be fed using either a standard PC power supply attached to the board's ATX connector or a +5 V supply. Additional voltages are derived on-board.

1.3.3 Memory

There are four memory technologies present in the system. The fastest memory available on the board are be the FPGA's built-in BlockRAM blocks. The fact that these RAMs are dual-ported is responsible for a substantial speed-up. The available amount of BlockRAM in the C-FPGA is large enough to contain a simple OS with basic services on the C-FPGA. On the R-FPGA, a few high-speed but not overly memory consuming tasks can be served.

To allow for fast tasks with slightly higher memory requirements, the R-FPGA has access to 4 MB of 16 bit wide SRAM memory. To store larger amounts of data, 64 MB of 16 bit wide SDRAM memory are present. The memory width of 16 bits is consistent with most signal processing applications.

As there will be the need to store additional program code and various bit streams in memory, the C-FPGA is equipped with the same amounts of SRAM memory and SDRAM memory, but organized in 32 bits due to the μBlaze's architecture. FlashRAM is present as non-volatile storage.

1.4 Development Environment

This section does offer an overview of the software being used to develop the system intended to run on the platform described above in section 1.3. To generate the software and the bit streams to be downloaded to the XFBOARD, the following software tools provided by XILINX have been used:

Platform Studio / EDK 6.1.2. The EDK is a tool suite that contains the tools to generate a bit stream from VHDL. The hardware of a system is being described in the microprocessor hardware specification file `*.mhs`. The systems designed with EDK commonly consist of a μBlaze CPU with some peripheral devices, the so called cores. After being defined in the `*.mhs` file, the VHDL or net-list descriptions of the cores and the μBlaze are combined using a wrapper and then syn-

thesized and converted into a bit stream. Now the software is being generated. The software libraries are described in the `*.mss` microprocessor software specification file and accordingly compiled. Then the user's source code for the program gets compiled and then linked with the libraries. The executable code obtained so far is merged into the bit stream, which is then ready to be downloaded to the target FPGA.

To compile, assemble and link the software, a port of the GNU C compiler `gcc`, in EDK called `mb-gcc`, and a port of the so called binutils, are used.

EDK also includes a GUI to compose such systems, which is a great help to get familiar with the environment, but does not suffice the needs of more advanced users.

XILINX also ships a large library of cores such as memory controllers and peripheral drivers. This library can be extended by user-defined cores which have to be written in VHDL alongside with a bunch of files that contain the information needed by EDK to include these cores in a microprocessor system.

Project Navigator / ISE 6.1. The is a tool suite used to synthesize and implement designs for the XILINX FPGAs and CPLDs. I used this tool suite to implement designs that did not fit into the EDK design flow.

The Project Navigator is a GUI that merges the various tools of the ISE into one front end.

In addition to these commercial tool suites, a number of freely available programs have been used. To download data via ethernet, PERL has been used. JAVA has been used to design a simple graphical OS configuration tool. The editor of choice was the almighty EMACS.

2 Related Work

A short note on work related to this thesis shall be given in this chapter.

2.1 Previous Work at the Computer Engineering and Networks Lab

A top-down approach to reconfigurable hardware OSs has been demonstrated in [22]. Design concepts have been described, and OS services have been defined in a device-independent way. An application case study has been delivered alongside these concepts.

A **reconfigurable OS prototype** has been developped in [14]. This OS was not implemented as a standalone system, because only one FPGA being the reconfigurable resource was used. The role that is played by the C-FPGA on the XFBOARD was assigned a host PC attached to the board.

Propositions of an online scheduling system allocating tasks to a block-partitioned reconfigurable device are discussed in [21]. Several scheduling approaches and placement strategies have been explored. Placement and partitioning algorithms are presented in [24]. The interdepence of scheduling and placement is discussed in[17].

2.2 Linux Ports

There exist various projects which have ported LINUX to run on reconfigurable hardware. Two of them shall be mentioned here.

MEMEC[7] and **MIND**[9] have ported Linux and RedBoot[12] to the MEMEC Virtex-II Pro platform. RedBoot is a complete bootstrap environment for embedded systems and allows download and execu-

tion of embedded applications via serial or ethernet.

uClinux[1] is a derivative of the Linux kernel intended for microcontrollers without memory management units. The **Microblaze uClinux** project has succeeded in porting this small kernel to the XILINX μBlaze. They even succeeded in performing device self-reconfiguration using the ICAP internal configuration access port by defining a device in the filesystem which can be written bitstreams to.

2.3 Stretch S5000 Family

The approach made by STRETCH INC.[18] is interesting from the hardware point of view. Their new device family, the S5000, is a software-configurable processor which embeds programmable logic within the processor. This is in contrast to the `Xilinx`' approach to include processors within an FPGA (Virtex-II Pro). Unfortunately, at the time writing only a press release on this device family existed, and this section has been written based on that information. Therefore, you are kindly invited to read these lines with reservations.

Developers using C/C++ not only program the CPU but also create new instructions ideally matched to their applications' needs. So called *hot spots*, which are sequences of operations that must be repeated many times, are reduced into single instructions. On conventional processors such as DSPs, optimization of hot spots is usually done by a programmer using low-level assembly code, which directly represents the sequence of processor operations one by one. Compilers automate this task, but only with a significant loss in performance. Further, because each operation is very simple, tens to hundreds of assembly instructions are needed to implement each hot spot.

When working with these S5000 processors, the software developer identifies such hot spots using a profiling tool. The source code from these hot spots is automatically compiled into a so called ISEF configuration. The ISEF is a software-configurable data-path based on proprietary programmable logic. This configuration is the used to create a single custon instruction that implements the entire hot spot.

This approach pairs the advantages of a general purpose processor with the speed gains yielded by specialized and dedicated hardware.

3 System Overview

In this chapter, an overview of the hardware and the software elements implemented in the C-FPGA shall be given. Some components have been placed at the disposal by XILINX, but a huge number of cores were to be designed by myself. A more detailed discussion of these elements is due in the following chapters. Large and important elements will be dedicated their own chapters, smaller parts will be combined into one chapter.

First, a summary of the hardware elements of the OS being implemented in the C-FPGA is given, then the software components are discussed shortly. The partition of hardware and software is intrinsic to the system: generally, device drivers are implemented in hardware, the rest is implemented in software.

3.1 Hardware Elements

The hardware of the system mainly consists of a XILINX μBlaze[29] with a large number of peripherals, like I/O drivers, and memory controllers. Due to the fact that an FPGA is used for the CPU, all these peripherals can be implemented on-chip side by side with the μBlaze CPU.

3.1.1 MicroBlaze CPU

As stated above, the central element of the system is a CPU soft core, the μBlaze. The key information about the architecture and the conventions of this processor shall be given in this section. For more detailed insights into the μBlaze please refer to [29].

This RISC processor has a 32 bit big-endian architecture and is optimised for the use with XILINX FPGAs. Have a look at figure 3-1 on how the bits are ordered. This endianness brings in a lot of potential pitfalls at the borderline between the μBlaze and eventual user cores, because most VHDL designers are used to have the most significant bit at position 0, which contrasts with the actual MSB

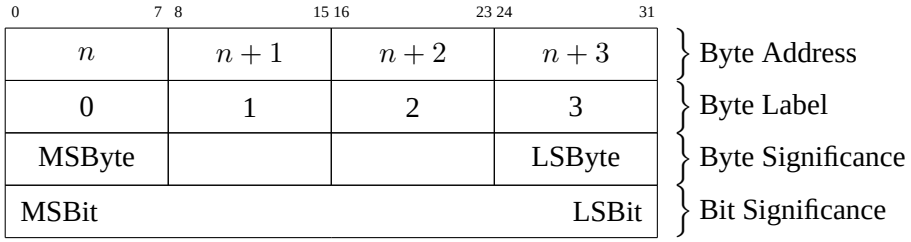


Figure 3-1: ***μBlaze Data Endianness****. The μBlaze is organised in a 32 bit big-endian architecture. In contrast to the bit order most VHDL programmers are used to, the most significant bit is bit 31.*

position 31.

The μBlaze has a 32×32 bits register file plus 2 special registers, and a 3-stage pipeline; each stage is active on each clock cycle, so three instructions can be executed simultaneously. The pipeline effectively completes (in general) one instruction per clock cycle; instructions performing accesses to external devices normally take a longer time lapse. The start of the program code is expected to be at address 0x0. However, if no valid instruction is found there, the processor's program counter is incremented until an executable instruction is found. The stack grows towards lower memory addresses.

One interrupt input is present, which forces the μBlaze to jump to a fixed address where a branch to the interrupt handler is expected to be found. If more than one interrupt is to be used, an interrupt controller has to be implemented. In addition to that interrupt input, signals that offer information about the μBlaze's internal state are available, e.g. the register contents can be read.

The processor's two local memory bus (LMB) ports are used to access on-chip BlockRAM memory. The instruction side LMB port is used to read instructions from memory and is therefore a read-only port, the data side LMB port is used to read and write data to the memory. The BlockRAM is a dual-ported memory technology, so the same memory cells can be accessed from the data side port and the instruction side port. The VGA driver's text memory is also connected to the LMB.

The μBlaze also features an on-chip peripheral bus (OPB), which is used to access external memory using the appropriate memory controllers, and to communicate with the drivers for the various I/O peripherals present on the XFBOARD. Although it is an on-chip bus, it is mainly used here to communicate with drivers for off-chip peripherals which results in higher bus-turnaround times.

3.2 Peripherals and Peripheral Drivers

To access the various memory technologies, the system incorporates the memory controllers needed. Two LMB BlockRAM interface controllers are present, one for the instruction side LMB and one for the data side LMB. On the instruction side, the *LMB BRAM Interface Controller*[28] placed at the disposal by XILINX was useful for this system. The controller for the data side LMB is a custom

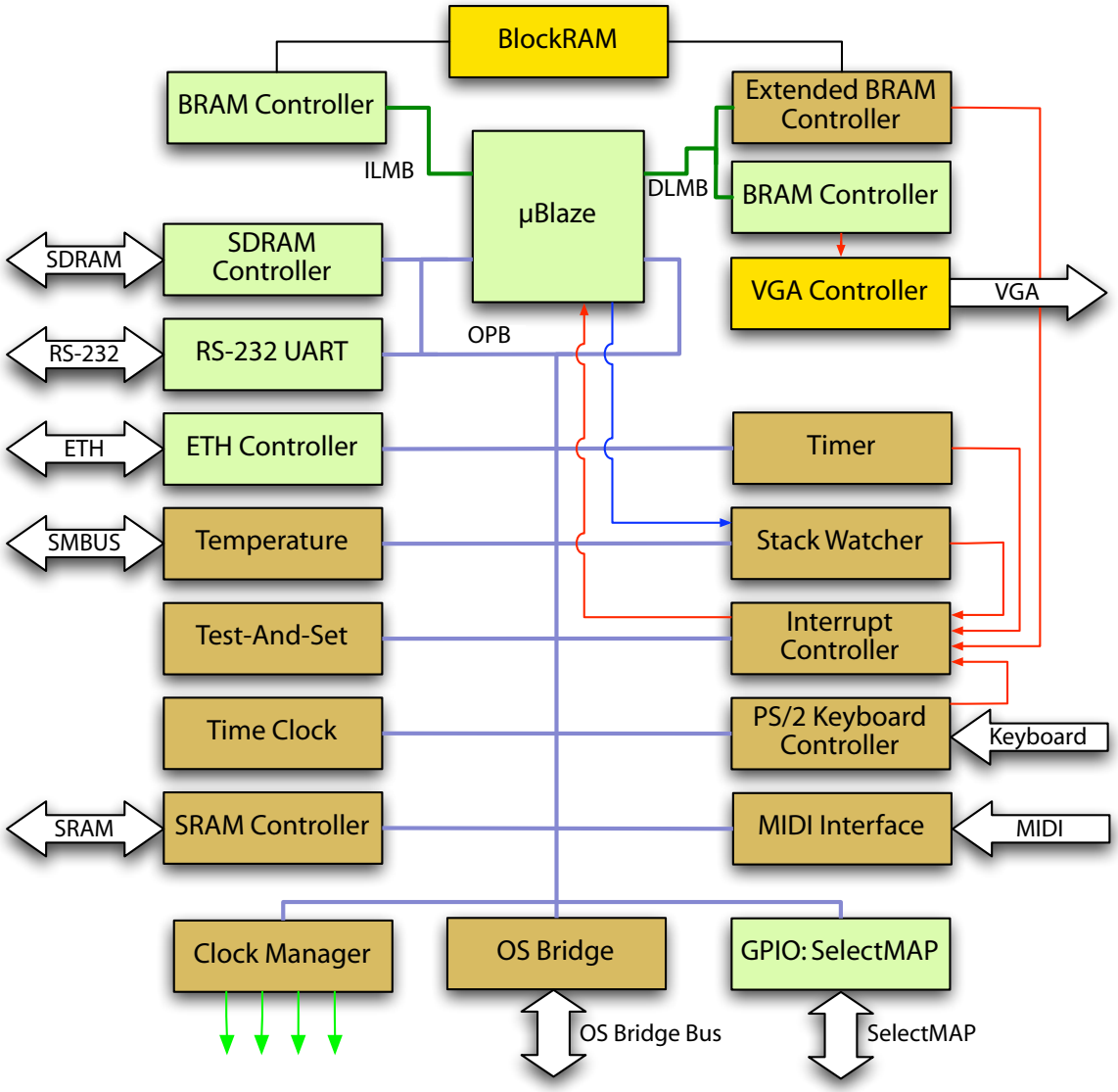


***Figure 3-2: MicroBlaze System Hardware**. This figure offers an overview of the hardware implemented in the R-FPGA. The data side and the instruction side OPB are connected to used the same memory modules for data and instructions.*
Legend:

- *Open Peripheral Bus (OPB).*
- *Local Memory Bus (LMB).*
- *Clock signals fed to the R-FPGA.*
- *Interrupt signals.*
- *Register Contents.*
- *Cores provided by XILINX.*
- *BlockRAM. The VGA controller is seen as a BlockRAM module.*
- *Custom Cores*

extended version of this controller to include an interrupt output which informs the system about intended write accesses to the read-only section at the beginning of the memory. The SRAM memory modules are controlled using a core made available by XILINX. The SRAM memory modules are accessed using a custom made SRAM controller. BlockRAM memory is used to store program code. SRAM memory is also used for program memory, and the stack and dynamically allocated memory is stored here. The main use of the SRAM is to store bit streams. A detailed memory map and a rationale for the partition of the memory technologies is given in chapter 5; there you also get information on the memory protection mechanisms in the system, namely the stack watcher that protects the system from stack overflows, and the above-mentioned extensions to the LMB interface controller.

The RS/232 UART has been inserted in earlier stages of the development process. The UART core also was available in the XILINX library. In the beginning, when no other means for communication where implemented yet and the VGA display was not put into service, the UART was handy to communicate with the system, because read and write operation to its interface were easy to implement, and most of the software routines are delivered with the core. Debugging messages have been printed to a terminal application on the host PC, and data have been uploaded through this rather slow and inefficient channel. Due to the presence of faster communication facilities, the UART has almost become obsolete. It is still included in the system because it does not use a lot of resources and therefore does not compromise the rest of the system at all. In addition to that, there still may exist applications where an UART is welcome due to its simple protocol.

To send and receive data via ethernet, an ethernet MAC provided by XILINX, the *OPB Ethernet Lite Media Access Controller*[30], has been included in the system. The ethernet port is used to transfer large amounts of data from and to the board, e.g. executable program code or bit streams are sent to the board. As soon as the ethernet interface has been introduced in the system, it took over the role that has been played by the UART before.

A VGA controller offers the services to write text to a display. Thanks to a custom adapted character set, simple graphical elements can be drawn. The VGA controller is basically accessed by writing to a memory location used as text memory. This text memory is implemented using BlockRAM which is accessed via a second BlockRAM interface controller attached to the data side LMB. The VGA controller is capable of displaying 1024×768 pixels in 8 colors; however, only 3 colors can be used at a time: a background color, and two foreground colors. The VGA controller is a custom core. The core documentation can be found in section 7.2.

A PS/2 keyboard driver is available to get textual user input. Most of the system is controlled using the keyboard. The controller generates an interrupt when the escape key is hit on the keyboard; an explanation of the special use of the escape key is given at the end of section 6.1.1. To allow for sound applications, a MIDI interface is also present. This interface implements only an input port; the board is not able (yet) to send MIDI data. As no audio output is connected to the C-FPGA, the MIDI data most probably will be forwarded to the R-FPGA and be processed there. Both the MIDI controller and the PS/2 keyboard controller are custom designs. The documentation of the two cores can be found in sections 7.5 and 7.7, respectively.

As a means to configure the R-FPGA with bit streams and eventually reconfigure it partially and

dynamically[1], access to its SelectMAP port is provided using a GPIO port. Due to the fact that a GPIO has been used, the whole protocol was implemented in software. This software solution therefore does not run at full speed. In addition to that, the CPU is being occupied for the execution of a simple protocol, which is not desirable. Refer to chapter 6.3 for more details.

To exchange data with the R-FPGA, a so called OS bridge controller has been implemented. The OS bridge controller sends and receives data using a simple command interface to a receiver entity in the R-FPGA, which then forwards the commands or data to the element being responsible to handle the command. The C-FPGA and the R-FPGA counterparts of the OS bridge have been designed together and are described in chapter 6.2. The OPB core on the C-FPGA side is also discussed in the core documentation in section 7.6.

In the designs configured into the R-FPGA, it most probably is not going to suffice to have only one clock frequency available. Generating the clock signals in the R-FPGA is possible, but their frequency must be set at synthesis time, so the simplest approach is to derive them from the system clock in the C-FPGA using counters and to forward them to the R-FPGA to allow for run-time configuration of these clocks. Four such clock signals are offered by the custom designed clock manager documented in section 7.3.

An interrupt controller has been designed that merges the interrupt signals present in the system into one global interrupt signal fed to the μBlaze. To get knowledge about the interrupts that actually have been triggered, the μBlaze can poll a register in the interrupt controller. Interrupts can also be masked using this controller. The interrupt controller core is documented in section 7.12

Last but not least three more trivial cores are to be mentioned. A test-and-set variable is available in the core documented in section 7.4, the timer needed for the process scheduler is documented in section 7.11, and a simple clock period counter used to count the clock periods completed since the last system reset with more details available in section 7.13.

3.3 Software Components

The basic operating system includes only the most important software components, as only very limited memory space is available in the BlockRAM. These basic elements need to reside in the BlockRAM, as this is the only memory that can be guaranteed to contain valid program code when power is applied.

The XFBOARD OS is a multi-tasked system. A number of processes in a process list are scheduled following a simple round-robin scheme. Each process is assigned a certain number of time quanta during which it may execute on the CPU before being pre-empted. Processes may yield the CPU at any time, activating the process being next in the list. The functions needed to manage the processes are also implemented: processes can be killed, suspended, assigned a different number of time quanta, and the process list can be displayed.

[1] actually, the development on the R-FPGA side has not yet reached the point where partial reconfiguration with complex tasks can be performed.

Command User Interface To launch processes and configure system parameters, a command user interface, the so called shell, is provided. The shell implements basic features such as tab completion[2] and a command history. In general, all keyboard input is catched by the shell, but certain commands are able to suspend the shell and process the keyboard input themselves. The shell also is able to start processes whose code is stored in external memory. The software functions representing the shell are discussed in section 8.2.4.

Ethernet Daemon An ethernet daemon constantly checks for received packets and dispatches them to the according process. It responds to pings and implements a UDP stack. Additionally, software routines are offered to processes which can be used to send packets. The network software has been taken from [5], adapted and extended by additional functionality. Refer to section 8.2.19.

Graphics Functions to comfortably print to the VGA displays are implemented (see section 8.2.6). The display is vertically divided into a text section in the left half, which is mainly used for the shell, and a graphics section, which is used for displaying ASCII-art graphics. The graphic display is multi paged. The graphic manager bothers with the graphical elements to be displayed and is responsible to allocate pages in the graphics section for the information to be displayed there.

Memory Management A memory manager cares about the allocation of stack memory to the processes to be executed. Using the memory manager, a process can dynamically allocate additional memory. A graphical representation of the memory map is shown in the display's graphics column. The memory manager is discussed thoroughly in chapter 5, its functions are described in section 8.2.13.

Configuration To configure and erase the R-FPGA using the SelectMAP port, a collection of functions is offered by the system (section 8.2.24). Communication with the tasks in the R-FPGA can be achieved using the macros defined to read from and write to the OS bridge. The routines to configure the MMU in the R-FPGA and to exchange data with this MMU, e.g. read or write FiFos, make extensively use of these OS bridge macros.

The functions or macros to access all other hardware peripherals not explicitly mentioned here are made available by the operating system, e.g. macros to write to the OS bridge. Refer to the OS code documentation in chapter 8 for the remaining functions and macros.

3.4 System Startup

A the beginning of the code, a short sequence performing the system bring-up is executed. In this phase, all services mentioned above are initialised, and a short welcome message is printed to the screen. Also, some parameters playing a key role in memory protection are set. The process list gets

[2]The completion of a partially entered command string into the full command when the tab key is hit is termed *tab completion*

populated with some services that will be running in the background during the whole system up-time: the ethernet daemon, the shell and the graphic manager. Then, all interrupts are demasked and the full control is handed over to the round-robin scheduler.

4 Scheduler

As the system is planned to be able to execute multiple processes at the same time, a process scheduler is included. I first want to summarize the various scheduling approaches commonly used in operating systems, and then discuss the scheduling scheme used in the XFBOARD OS.

4.1 CPU Scheduling in General

Having some processes running at all times is the objective of multitasking. The CPU should be switched among processes so frequently that users can interact with each program while it is running. If you have, let's say, a shell running, it seems (or at least should seem) like this shell is able to react at any time on your keyboard input. Actually, the shell is not running at any time because other processes are assigned the CPU rather frequently. The interval between the moments where the shell is indeed running on the CPU are kept so short that you won't even notice that the shell was not active when you input some data using the keyboard; the shell reads this input when it is active the next time, i.e. some 10 to 100 milliseconds in general, depending on the CPU load[1]. This time lapse is short enough not to be perceived by the user.

All processes to be executed are put into a job queue, or, to say it in other words, added to a process list. This list needs to hold information about the process, which is commonly referred to as the process control block (PCB). This information is mainly needed to (re)activate processes which are not currently running on the CPU. Examples for this information would be

- state of the process; is the process new, running, halted, etc. ?
- program counter; at which location does the process start/continue when launched / reactivated?
- CPU registers; which data was currently being treated when the process was taken the CPU?

[1] the CPU load is nearly proportional to the number of processes the CPU is switched among

- CPU scheduling information; what priority does this process have, or how long is it allowed to be active on the CPU?

The information being present in such PCBs heavily depends on other OS features and on the scheduling scheme used in this OS, therefore the contents of a generic PCB cannot be listed exhaustively here.

When a process terminates, voluntarily recedes from the CPU or is forcibly removed from the CPU by the scheduler, the contents of this process' PCB need to be updated. Then, the next process to be assigned the CPU is selected from the process list using one of various selection schemes. A variety of such scheduling algorithms shall be mentioned here:

first-come, first-served scheduling is the simplest scheduling approach. The process that entered the process list first assigned the CPU first. The process is not removed forcibly from the CPU and can stay there as long as it wants to. This scheduling approach is called non-preemptive. A process could possibly occupy the CPU forever.

shortest-job-first scheduling is an approach that selects the process with the shortest execution time to be assigned the CPU. This algorithm is probably optimal as it minimizes the average waiting time for a given set of processes. However, it is very difficult to estimate the duration of the execution time of a process. This scheduling scheme can be preemptive or non-preemptive.

priority scheduling selects the processes according to their priority. This method brings in the advantage, that important processes are executed first. The problem to define a measure of priority arises. Why is a process more important than another? This scheduling scheme can be preemptive or non-preemptive.

round-robin scheduling really shares the CPU among various processes. A process is assigned the CPU for a small unit of time, called a time quantum, and then the CPU is switched to the next process. When the scheduler arrives at the end of the process list, it wraps around and starts again from the beginning of the list. This scheduling algorithm is preemptive.

Both *shortest-job-first* and *priority* scheduling can be implemented in a non-preemptive or preemptive manner. They are preemptive when the currently executing process is removed from the CPU as soon as a shorter process or one with a higher priority arrives. Long processes or processes with low priority can possibly be blocked indefinitely.

Refer to [15] for a detailed discussion of processes and their scheduling.

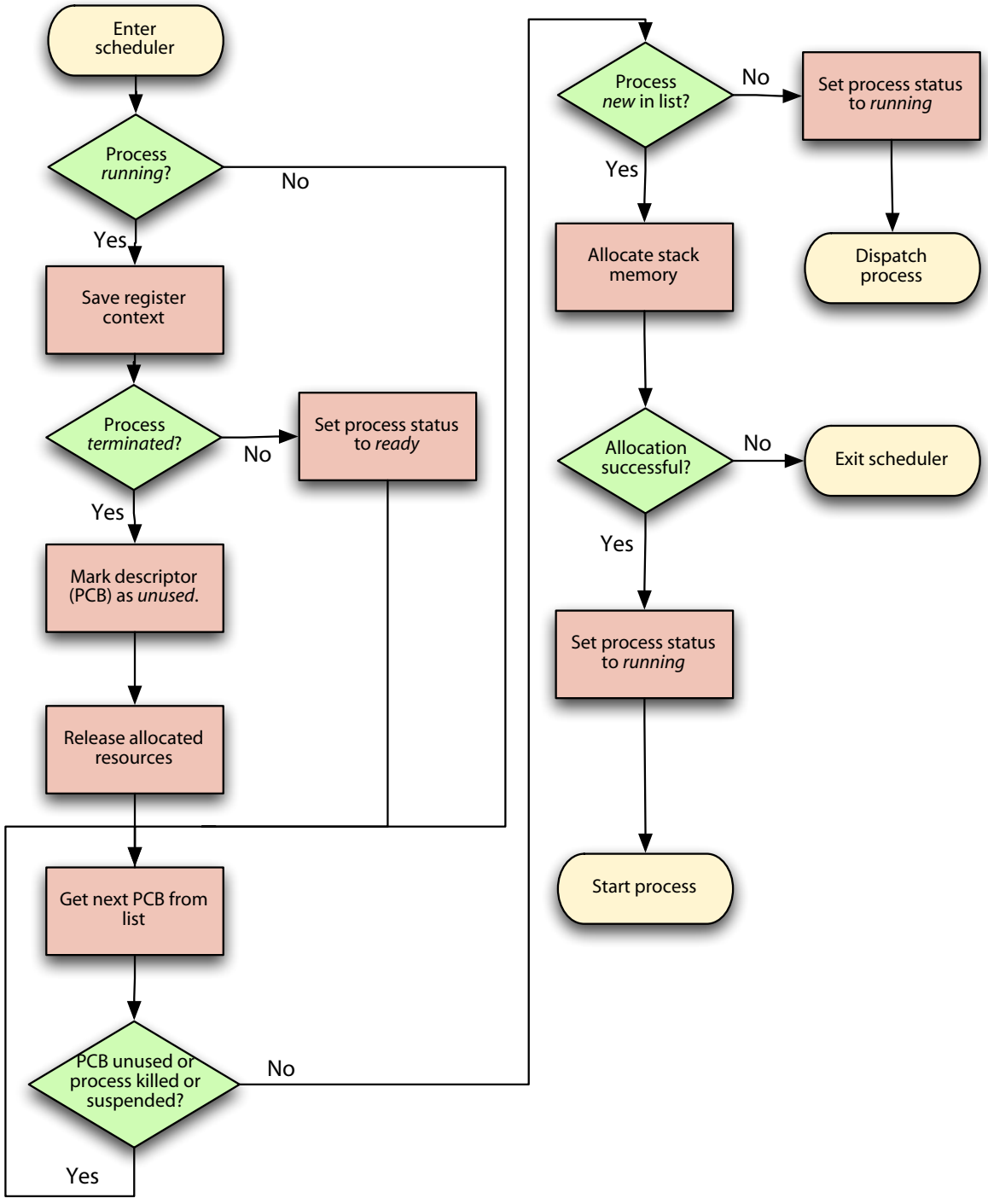


Figure 4-1: Scheduler Flow Chart*. This flow charts shows the actions performed when the scheduler is called upon time slice completion of a process. The current process' context is saved, and the next process ready to execute is selected from the list.*

4.2 CPU Scheduling in the XFBOARD OS

4.2.1 Round-Robin Scheduler

The scheduling scheme selected for the XFBOARD OS is a round-robin scheduler, because this is the simplest implementation of a time sharing system. As stated in the introduction to this chapter, more than one process at a time needs to be executed on the CPU, so a time sharing and preemptive approach is indeed mandatory. The pool of processes available to be scheduled is represented by a process list of fixed size, i.e. an upper limit for the number of process that can share the CPU is given. This upper limit is set to 15 in the current build of the OS. The process list is represented by an array of PCBs (see section 4.2.2 for details on these PCBs). The index of a process' PCB is, at the same time, the process ID by which the process is referenced.

4.2.1.1 Scheduler Flow

In a round-robin scheduler, the CPU is switched between a number of process in fixed intervals. The shorter these intervals are, the higher is the degree of process concurreny perceived by the user. For very short intervals, the time actually needed to perform the switch between two processes is dominant. Therefore, a tradeoff between short switching intervals and low switching overhead exists.

Each process in the XFBOARD OS is assigned a number of time quanta to execute on the CPU. One such quantum is a time slice of approximately 1.3 milliseconds, which is the minimum value making any sense, because the time needed for the context switch is in the same order of magnitude. As soon as this number of time quanta is up, the timer generates an interrupt and an interrupt handler gets executed. This interrupt can also be forced by the process; such a forced interrupt can be achieved by letting the timer expire. This is done when a process finishes execution or yields the CPU, e.g. when it waits for I/O.

The interrupt handler stores the content of all CPU registers in its stack, and then the scheduler gets activated, performing the following steps:

1. The scheduler checks whether the actual process is still running. If this check is positive, the register context stored in the calling interrupt handler's stack is copied to the PCB of this process. If the check is negative, which means that the process is either killed, suspended or blocked, the context in the PCB is either unused[2] or unchanged since the last context save, so the flow is continued with step 3.

2. If the process has not yet finished and is in the ready status, the flow is continued at point 3. If the process has finished, which means that the program counter stored in the register context points to the location where the exit function[3] resides in memory, the PCB is marked as unused

[2] The kill function which is used to put a process in the killed status has already deallocated all resources, including the stack memory.

[3] The exit function is implemented as an empty endless loop.

and returned to the pool of process list entries that can be assigned to new processes added to the list.

3. The scheduler now looks at the PCB with the next higher index in the list. If the process described by this PCB is unused, killed or suspended, it cannot be assigned the CPU. Therefore step 3 is performed again.

4. If the process described by the PCB is new, step 5 is performed. Otherwise, the process is put into the running status, and the dispatcher is called. The dispatcher writes the contents of the register context back to the originating registers; as soon as the program counter is restored too, the execution of this process is continued and the scheduler is **exited**.

5. As the process described by the current PCB is new, stack memory needs to be allocated (see section 5.2.2). If this memory allocation succeeds, the arguments submitted with the process are moved to the argument registers, and the program counter is set to the process entry point stored in the PCB and. So the process' execution is initiated and the scheduler is **exited**. If stack memory allocation failed, the scheduler is activated again, which means that the next process ready to execute is searched for. As soon as some memory large enough for the previously failed stack allocation has been freed, the process can be executed.

4.2.1.2 Adding a New Process

When a new process is to be launched, an unused PCB entry (or the entry of a killed process) needs to be found in the process list. The list is searched from the beginning at index 1, because index 0 is occupied by the system idle process[4]. As soon as a PCB is found, the addition of this process is successfully completed. If no unused PCB can be found, the process cannot be started, and an error message is printed.

4.2.2 Process Control Blocks (PCB)

The processes in the XFBOARD OS are represented using a structure that holds all the information needed by the round-robin scheduler to correctly handle these processes (see figure 4-2). For every process, there exists an instance of such a structure in the process list. Each such instance makes the following information available:

Process entry point. The process entry point is a pointer to the address where the function representing this process starts in the code. This field of the PCB is only used for tasks that have newly been added to the process list and are waiting to be launched for the very first time.

Register context. The register context holds a snapshot of the contents of all relevant registers. The snapshot is taken when the process gets removed from the CPU. Therefore, the register context

[4] A system idle process is always needed because the process list must not be empty as the CPU always needs to be fed with valid instructions. The idle process executes dummy instructions.

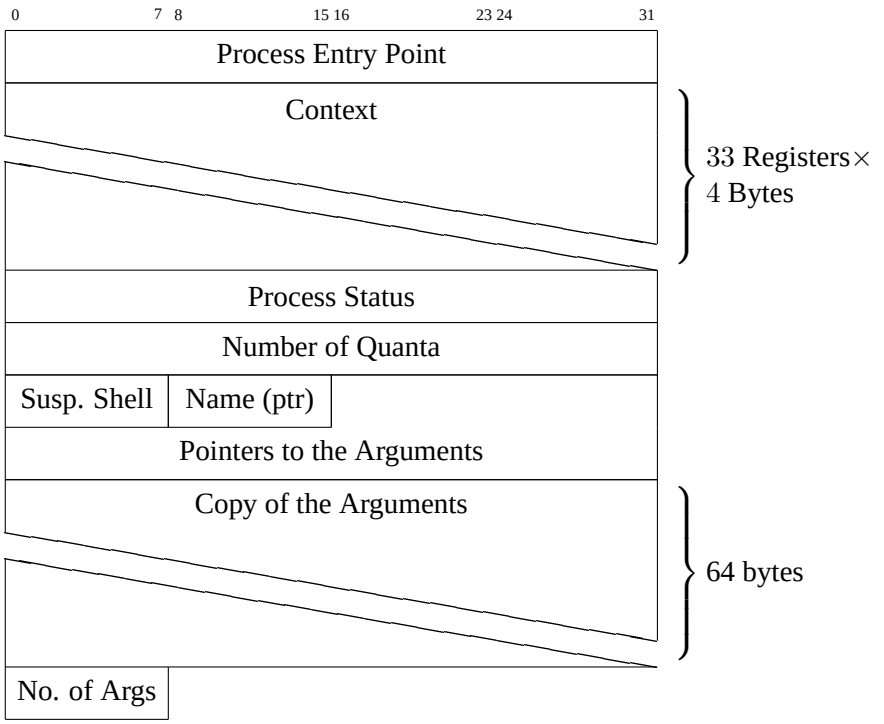


Figure 4-2: Process Control Block. *A process control block in the XFBOARD OS is 213 bytes in size. To optimize the memory requirements, changing the way the arguments are stored is most promising. The amount of memory used by the register contents in the context is mandatory and cannot be reduced.*

is empty for new processes that have never been active on the CPU before. The register context is needed to let the process continue exactly at the same point where it has been interrupted.

Stack information. The information about the stack size and its location in memory is needed by the scheduler only the first time a process is launched, because at that moment the memory for the stack needs to be allocated. Afterwards, the only information used by the scheduler is the value of the stack pointer, which is stored in the register context. As the stack dimensions still might be informative for the user, this stack information is kept in the process list.

Process status. The scheduler needs to know the status of a process to take the according action when browsing the process list. A process can be in one of the following states:

- unused
- new
- running
- ready
- blocked
- suspended
- killed

The exact meaning of these states is discussed in section 4.2.3.

Number of quanta. The number of time quanta a process is allowed to have exclusive access to the CPU is needed by the scheduler to set the timer accordingly when the process is started or reanimated.

Suspend shell flag. When this flag is set, the corresponding process suspends the shell[5] to catch keyboard input which would otherwise be consumed by the shell. This information is not actually needed for scheduling purposes; it is mainly used to be displayed when the process list is requested by the user.

Name of the process. The string representing the name of the process only is used for displaying purposes.

Argument list. The argument list of a process is needed the first time when it gets activated by the scheduler, as the arguments need to be forwarded to the function to be executed. Afterwards they remain in the process list for the case the user wanted to display them.

As the process list wastes a respectable amount of memory when implemented that way, the memory usage of such a PCB could be reduced in future. To have a full copy of the arguments in the PCB is maybe not the best approach, but it was the easiest to implement at the time the scheduler was

[5] Actually, the shell process is not put into the *suspended* status. It is self-suspending because it checks for shell-suspending processes whenever it is *running* and voluntarily yields the CPU whenever such a process is present.

developped. The arguments need to be stored somewhere for the time span between adding a command to the process list and its actual execution on the CPU. If they are not stored in a statically allocated field in the PCB, the memory to keep them needs to be allocated dynamically, but dynamic memory allocation was introduced in the system later in the development process.

4.2.3 Process Statuses

Now a detailed description of the various process statuses is due. A diagram with the transitions possible between the statuses is to be found in figure 4-3. When a process enters either the unused or the killed state, its stack and memory gets deallocated and all other used resources are released.

Unused Process

The term *unused process* is not a real process status. In fact, this means that this PCB entry in the process list is empty and not currently in use. As a consequence, entries marked to be *unused* are in the pool of PCBs that can be assigned to new processes added to the process list. There are to ways a PCB entry can become unused:

- The PCB has never been used. On system startup, all process list entries are initialized to be in the *unused* status.
- The process has completed.

New Process

A process that has been newly added to the process list is in this status. Such a process has never been active on the CPU and therefore most fields of its PCB are empty. This status can only be entered and left due to scheduler activity. The only status that con follow the *new* status is the *running* status.

Running Process

A running process is currently executing on the CPU. Obviously, only one process at a time can be in this status. A process being *running* can change to the following statues:

ready: the ready status can be entered either voluntarily by the process by yielding the CPU or forcibly by the scheduler when the process' time quantum is over.

unused: the unused status is reached when the process finishes internally. This state is not entered when the process is killed!

blocked: the process is blocked when it calls LOCK() while another process is owning the lock.

suspended: the process may be suspended by the user. Suspension can be achieved by hitting the escape key on the keyboard. However, only processes suspending the shell can be suspended, as the suspension state is used as a means to escape to the shell from such processes.

killed: when the process is behaving ill, it can be killed by the OS to prevent the system from getting unstable.

Ready Process

A process that is ready to be executed on the CPU is in the ready state. Ready processes can only enter two status: they can be moved to the running status by the CPU, or they can be killed by user input.

Blocked Process

A blocked process also can be assigned the CPU, because only the process itself is allowed to perform the transition from *blocked* to *running*. To be able to perform this transition, the process must be executing on the CPU. If the process decides to keep staying in the blocked state, it immediately yields the CPU. A blocked process can be killed by the user.

Suspended Process

A suspended process is never assigned the CPU. The process is only allowed to leave the CPU when the user decides to resume the process. A resumed process is put into the ready status.

Killed Process

When a process is killed, its PCB is returned into the pool to be used for new processes.

4.3 Hardware Scheduling in the XFBOARD OS

As the XFBOARD OS is meant to be an OS for reconfigurable hardware, an additional scheduler that switches the R-FPGA's resources between various hardware tasks would be interesting. Such a scheduler is needed if the R-FPGA can be partially and dynamically reconfigured. This partial reconfiguration cannot be achieved yet, therefore no such scheduler has been designed.

For previous work on this topic, refer to section 2.1.

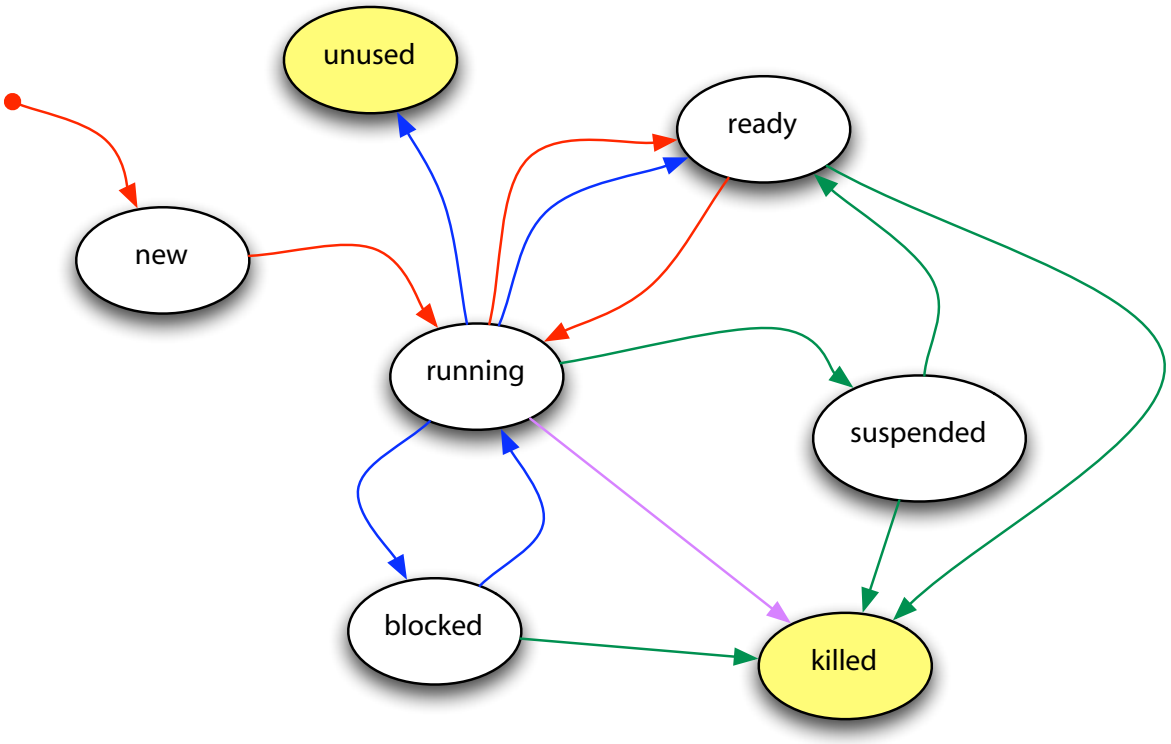


Figure 4-3: Process Status Transitions. *This graph shows all transitions that can occur between the various process statuses. The edges in the graph are colored; the colors indicate who can be responsible for the corresponding transition.*

Legend:

- *Scheduler and scheduling utility functions*
- *Running process*
- *User (by keyboard input)*
- *Operating system component other than scheduler*
- *Process states creating an empty and reusable entry in the process list*

5 Memory

5.1 Memory Layout

As stated in the introduction in section 1.3.3, there are 4 types of memory present on the board: BlockRAM, FlashRAM, SRAM and SDRAM. These memory technologies greatly vary in size and access speed, therefore each technology may serve better for some purpose while being inappropriate for another. This section helps you figure out which memory type is used for what purpose and offers an explanation for this partition. A memory map is drawn in figure 5-1. At the time writing, the FlashRAM has not been in use yet.

The numeric values given for the memory locations apply for the OS build I submitted with this documentation. These values may have changed due to future edits in the source code.

5.1.1 BlockRAM Memory

The BlockRAM memory is used store the program code for the operating system. The OS' variables are stored here, and the stacks of the functions being executed during system bring-up and of the interrupt handler and scheduler are placed here.

The main reason to have the OS code in BlockRAM is the speed of execution. OS functions should be executed as fast as possible to allow for efficient system operation, as OS function are likely to be used rather often. Similar arguments lead to the decision to have system variables and the various allocation tables and lists in the BlockRAM, in spite of the hard size constraints given by the BlockRAM's dimensions considered later in this section. Also, it is mandatory to execute the scheduler and the

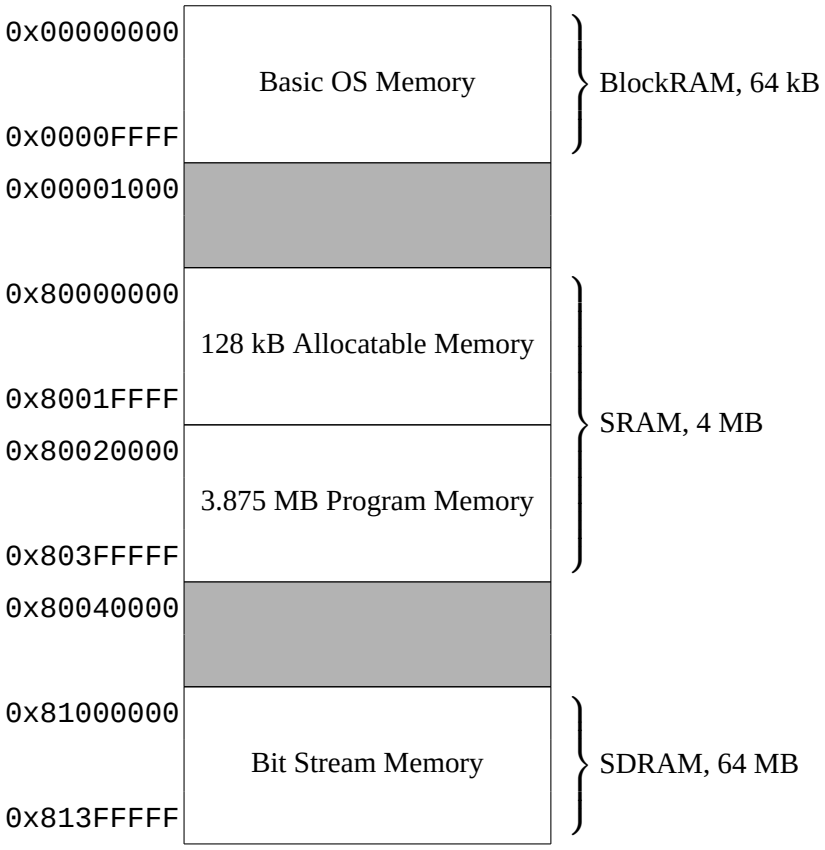


***Figure 5-1: Memory Map**. This memory map shows which memory types are used for what purpose and the physical addresses of the used regions. In the grayed regions, no memory is present; these regions are either unused or occupied by the LMB or OPB cores.*

interrupt handler at the maximum speed achievable, as they are called very frequently, so even the stacks of these functions are held in BlockRAM; the stacks of all other functions are kept in SRAM (see section 5.1.2).

BlockRAM is the fastest memory accessible in the system, because it is attached to the LMB (see section 3.1.1) which is dedicated to the data transfer between μBlaze and BlockRAM. As a consequence, there is no other traffic on the bus, in contrast to the OPB bus which is connected to a lot of peripheral components generating traffic on the system. In addition to that, BlockRAM is dual ported memory, one port of which is connected to the port the μBlaze fetches the instructions, and the other port is connected to the port it reads data from memory. So data can be read or written at the same time an instruction is fetched.

The second reason to keep the OS program code in the BlockRAM is the fact that the memory cells can be assigned a default value which is set in the bit stream. This means that the content of the BlockRAM is configured the same time the hardware is uploaded to the FPGA: once the hardware is ready, the software may start to be executed. If one of the other memory technologies were used to contain the OS, the program code would need to be uploaded in a separate step.

Although the BlockRAM brings in these significant advantages, it's limited size is an unpleasant drawback. The Virtex-II XC2V1000 offers 96 kB of BlockRAM memory, but due to the fact that the amount of memory to be used for program code has to be of the form 2^n as imposed by EDK, the maximum that can be instantiated for that purpose is 64 kB. Being in the know of this limit, it is obvious that only the most important parts of the OS can be stored in the BlockRAM.

5.1.2 SRAM Memory

As the amount of BlockRAM memory is unlikely to satisfy the need of space for more complex programs, the OS provides means to upload program code into SRAM. Extensions to the basic OS get stored in SRAM, as does the code of custom user programs. 3.875 MB of memory are reserved for this additional code[1]. External programs are uploaded in the MOTOROLA SREC format via ethernet; refer to section 9.2 and appendix B where this process is explained in more detail.

128 kB are used for storing huge amounts of data using `malloc()` and the stacks of the processes in the system, which also consume large memory chunks. This way to use the memory infers the need for allocation schemes which are discussed in section 5.2.

The SRAM does not support dual-ported access, thus requests for instructions and data get serialized. As a result, the code is executed much slower, and data residing in the SRAM cannot be accessed as fast as in the BlockRAM. Another factor that slows down the access to this memory is the traffic on the OPB bus, as the controller for the SRAM modules is attached to this bus.

Enabling the μBlaze's cache possibly reduces these effects. However, this feature has not been used yet, so no additional information can be offered on that topic.

[1] When speaking of code here, the global variables and initialized data are included too.

5.1.3 SDRAM Memory

SDRAM memory is used only to store bit streams to be configured into the R-FPGA. As these bit streams consume large amounts of memory (a full bit stream is 1.2 MB in size), the SDRAM is adequate for that purpose. SDRAM can be accessed in bursts, which is ideal to download configuration data to the R-FPGA.

The burst access feature of the SDRAM cannot be used with the actual configuration solution in software. This feature will become interesting as soon as a DMA controller is implemented which will perform the download to the R-FPGA

The bit streams to store in memory are sent over the ethernet, as explained in section 6.3.

5.2 Memory Allocation

5.2.1 `malloc` and `free`

When a process needs large amounts of memory, it is not practical to statically allocate the space needed by creating variables in the according size. Often, the exact size of the memory needed can not be predicted at compile time. This means that enough memory would be allocated to satisfy the maximum demands by the process, which is probably far more than an average pass of the process might need. It is much better to dynamically allocate memory in the size actually needed as soon as it is used and to deallocate it as when it is no longer needed. An that is exactly what the SRAM is used for in the XFBOARD OS.

The memory is divided into a number of blocks of equal and fixed size to simplify the allocation and management algorithms. When a process requests memory using `malloc`, a contiguous set of blocks is searched for which is large enough to contain memory in the requested size. In the XFBOARD OS, there are 512 blocks, each having a size of 256 bytes.

These dimensions only apply for the build I submitted with this documentation. Most probably, they will be adjusted in future to allow for more memory to be dynamically allocated.

If a memory chunk of 400 bytes is requested, 2 blocks are returned, representing 512 bytes. Assuming uniformly distributed sizes of the memory requests, this approach with fixed-sized blocks brings in the disadvantage of allocating too much memory: due to the round-up process, half of a block is wasted in average, a problem commonly termed *internal fragmentation*. The lower the block size, the less memory is wasted.

A table is managed which contains an entry for every block in that memory. Each entry holds the following information:

Owner PID: the system needs to which process a memory block is assigned. This information is needed for the cleanup when a process exits or is killed, as all memory used by this process needs to be deallocated.

Memory type: a memory block can either be free, used for memory allocated using `malloc`, or used as stack memory (see section 5.2.2.

Last block: This flag marks the last block of a contiguous region composed of a number of blocks.

This table is referred to as the memory allocation table (MAT). The size of memory used to hold this table is proportional to the number of blocks being managed. So choosing an appropriate block size the memory is divided into is a trade-off between minimal internal fragmentation and minimal management complexity.

When memory is requested by a process, the MAT is searched for a contiguous region, or hole, that fits the requested size. As discussed in [16], this search can be performed using one of the following approaches:

First fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the MAT or where the previous search ended. The search can be stopped as soon as a free hole being large enough is found.

Best fit: Allocate the smallest hole that is big enough. To find this hole, the whole MAT must be searched. This approach increases the search time, but produces the smallest leftover hole, but small holes eventually cannot be used anymore.

Worst fit: Instead of the smallest hole, the largest hole is searched for. This approach comes with the same search overhead as the *best fit*, but the large leftover holes are more likely to be reusable.

As first fit searches and best fit searches are shown to be comparable in terms of storage utilisation, the first fit approach is used in the XFBOARD OS because it is generally faster.

When memory is not used anymore, it can be deallocated by the process that owns this memory. Deallocation is performed by executing `free` on a pointer to the beginning of the memory region. The memory is then marked as free in the MAT up to the entry flagged as the last block of this region.

5.2.2 Stack Allocation and Management

Let's start thinking at the at the assembly language level: each time a subroutine gets called, the stack gets enlarged by the amount of stack memory this subroutine needs. When the subroutine is returned

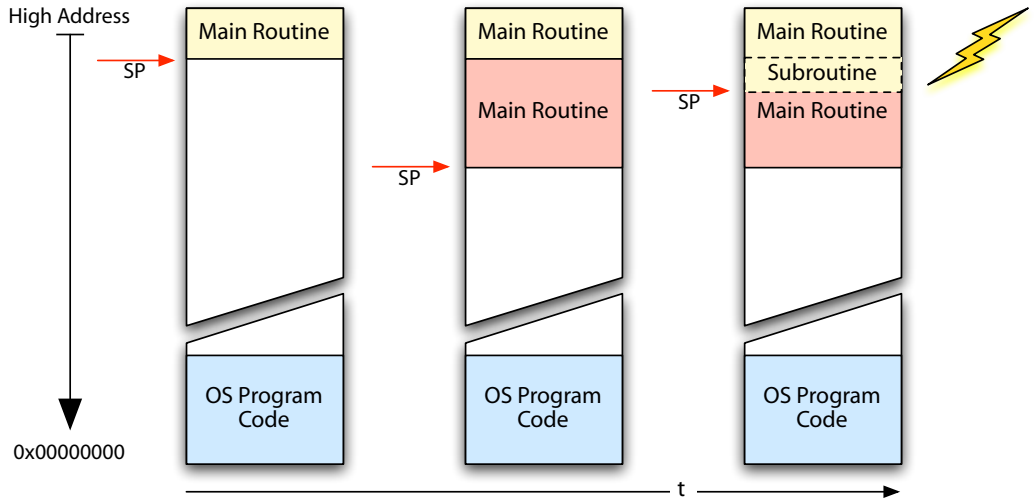


Figure 5-2: Stack Overflow Scenario 1*. A process is running, having stack allocated for the current function. Then, a new process is launched with its own stack. Later, the first process is on the CPU again and calls a subroutine. The stack pointer is decremented and grows into the stack of the second process.*

from, the stack is shrinked by the same amount of stack memory. These size adjustments are performed by changing the value of the stack pointer. Depending on the stack convention of the system, the stack pointer initially points to a low address and is incremented when the stack grows, or the stack pointer initially points to a high address and is decremented when the stack grows. The latter stack convention is the one used in the μBlaze.

This works well in an environment where the CPU is not shared among various processes, and where recursive function calls are guaranteed not to occur. But what happens when time sharing is performed? A number of processes are being scheduled, every one with its own stack, but none of them is informed about the position of the stack of the other processes. When a subroutine is called in a process, the stack pointer is decremented, and eventually grows into the stack of another process, unwantedly changing the content of this stack. Most probably, the system will show erroneous behavior.

Otherwise, if no such error occurs, terminated processes will leave unused memory where their stack resided. As the stack pointer is alway decremented for each new process or subroutine, it tends to indefinitely grow to lower memory. Sooner or later, the stack pointer reaches the program code which is located at the beginning of the memory, and overwrite this code with bogus data. The system will crash. Many more catastrophic scenarios could be thought of, as the system's behavior would be rather indeterministic. Two such scenarios are depicted in figures 5-2 and 5-3.

It is obvious, that stack memory must be managed. In the XFBOARD OS, every PCB contains information on the stack. This information contains the size of the stack needed for the process. When the process gets started by the scheduler, stack memory is allocated using the allocation scheme discussed

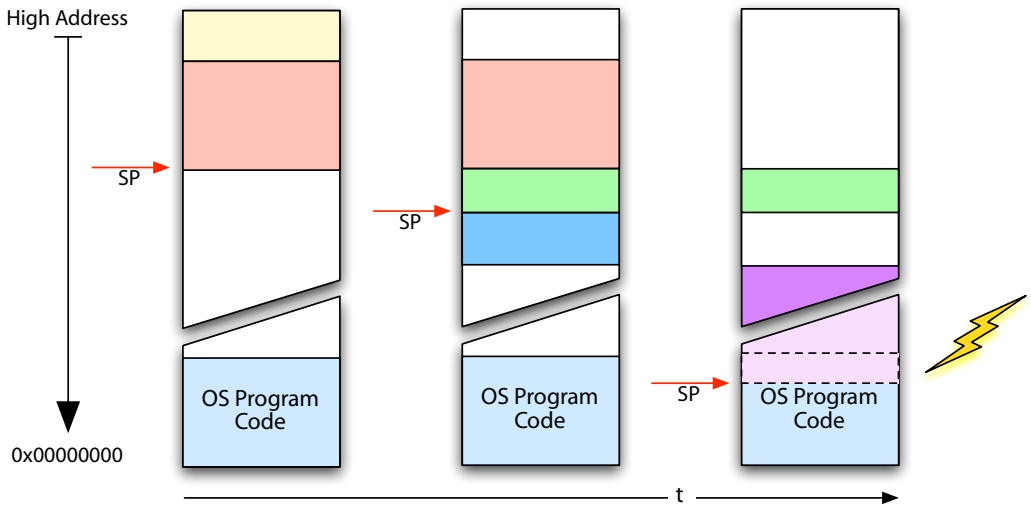


***Figure 5-3: Stack Overflow Scenario 2**. Some processes are running. Terminated processes eventually leave unused memory at the location their stack resided, as the stack pointer only can be decremented. It is only a question of time for the stack to grow into the program code*

in section 5.2.1. Assuming that the stack size given in the PCB is correct, stack overflows should not occur. Due to the fact that the stack memory is managed and clearly defined, the stack will not grow into the program code at the beginning of the memory.

Be sure to correctly calculate the amount of memory needed for the stack when implementing your own processes. If these numbers are not correct, the system might kill your process as described in the next section!

5.3 Memory Protection

Memory is not generally protected in the XFBOARD OS, as the CPU does not provide the hardware needed for this purpose. Any process can overwrite and read the data of any other process. Still, a minimum of protection can be done.

5.3.1 Stack Monitoring

In section 5.2.2 I mentioned that the stack size of each process must be known. Of course, the estimate of this size can be too small, or a subroutine could be called recursively, making the stack grow beyond its expected boundaries and therefor compromise other processes by overwriting their stack. As the

OS should not rely on these estimates which are made by the programmer of the process, a means to monitor the stack is mandatory.

Additional hardware has been developed to watch the location the stack pointer is pointing at. As the PCB contains the information where the stack of the process is located in memory, this core can be informed about the values the stack pointer is allowed to assume. This information is written to the core by the scheduler when a process gets activated. As soon as the stack pointer leaves the allowed region, an interrupt is raised and the process is killed, which prevents the other processes and the OS from being corrupted. The core that has been developed for that purpose is the `opb_xfcregwatch` described in section 7.8.

5.3.2 Read-only protection

As stated above, every process can write to the variables used by other processes. These erroneous accesses cannot be easily avoided, and sometimes processes need to communicate with each other using common variables.

Processes even have write access to the read-only memory section where the OS program code and the constants reside, which is fatal. This access could be forbidden by locking the according memory section for write accesses via the LMB bus. The address range to be locked is not practical to be statically configured into the system at hardware synthesis time, because the software may change, and therefore the limits of this read-only memory section are not fixed. Whenever these limits change, the hardware would need to be generated again.

This problem is solved by adding additional functionality to the memory controller connecting the LMB (see section 7.1) which protects the read-only section of the BlockRAM memory from write accesses. If such an erroneous write is about to occur, an interrupt is raised and the process performing this access is killed. As the instruction side LMB anyway supports the read direction only, this special controller must be inserted at the data side LMB only.

The region containing read-only code can be easily determined. It starts at address 0x0 in the program memory, and it ends at the position pointed at by the `_erodata`. This symbol gets assigned a value when the OS code is linked. When the system starts, this value is written to the BlockRAM controller.

As only the read-only section of the OS code and data is proteced from unwanted accesses, processes might write to other code sections unintentionally and compromise system stability. Be careful with those accesses!

6 Services

This chapter shall mention additional OS services and elements that are not interesting or complex (from a conceptual point of view) enough to be dedicated their own chapters.

6.1 User Interface

Making use of the text capabilities provided by the VGA core (section 7.2) and the functions to write to this display (section 8.2.32), a simple textual user interface has been implemented which executes commands entered using the keyboard. The keyboard driver is discussed in sections 7.7 and 8.2.10. The display is vertically divided into two columns. The left column is used to enter commands, the right column is used to display ASCII-art fashioned graphics.

6.1.1 Command User Interface: Shell

The left column of the text display is used to implement a command user interface commonly referred to as *shell*. Using this shell, the user can enter commands which start new processes or control the system behavior. The shell includes a history which contains the recently entered commands. This history can be accessed using the *up* and *down* keys on the keyboard. The history is a circular buffer of fixed size. The shell is also enabled to expand partially entered commands. This expansion can be accomplished by hitting the *tabulator* key. When nothing has been entered yet, hitting the *tabulator* key results in a list of all available commands being displayed. When some characters have been entered and then the *tabulator* key is hit, the shell tries to find commands beginning with the same substring. If more than one such command can be found, all possible completions are listed; if only one such command exists, the characters entered so far are expanded to the full command. Command completion does work not only with built-in commands, but also with user defined commands. Refer to section 9.2 on how to add your own commands to the system.

Be sure to define the commands to be added to the shell correctly and in alphabetical order. Unwanted behavior and even shell instability may occur if not doing so!

The left column implements a scrolling function which is activated as soon as the bottom of the screen is reached.

Whenever the shell is running on the CPU and waiting for input, it consumes all characters entered using the keyboard. Therefore, when a process wants to get keyboard input reliably, it must be defined to be able to suspend the shell. An explanation on how to do this is given in section 9.2.

The keyboard driver forwarding the characters to the shell implements the mapping of raw scan codes sent by the keyboard to the real characters. It also performs the handling of special keys:

F-Keys: commands including their arguments can be mapped to the F-Keys on the keyboard. When such an F-Key is hit, the corresponding command is executed. This is useful for often used commands.

Page up, Page down: these keys are used to browse through the pages of the graphics manager, see section 6.1.2.

Print Screen: this key calls a print screen function that dumps the content of the screen to the UART.

The escape key also experiences special treatment, but it is catched in the hardware of the keyboard driver. When a process is running that is suspending the shell, the escape key can be hit to generate an interrupt which suspends the process to enable to escape to the shell. The process that has been suspended using the escape key can then be resumed later, again suspending the shell.

The functions representing the shell are discussed in section 8.2.4.

6.1.2 Graphics Manager

As mentioned above, the right column of the display is used to display simple ASCII graphics. A graphics manager is implemented that cares about displaying these graphics. This graphics manager provides a number of pages to display these graphics. Using the *page up* and *page down* keys navigation through these pages is achieved.

For every graphics element to be displayed, a hook-up function must be written providing the number of lines needed in the display and the code to actually draw this element. This hook-up function must then be installed into the graphics manager.

Whenever a hook-up function is to be installed into the graphics manager, the graphics manager first tries to fit the graphics into the first page of the graphics display. If there is no room for this graphics element, it continues searching on page two, and so on. If no room is available on any of the pages, the graphics element cannot be installed.

When the graphics manager is running on the CPU, it subsequently calls all drawing functions installed in the currently active page. As soon as all elements are drawn, it yields the CPU.

6.2 OS Bridge

The OS bridge enables the communication with hardware OS elements present in the R-FPGA. Data can be transferred to and from the R-FPGA. Configuration data can be written to OS elements, and status information can be polled. Also, it can be used for debugging purposes.

The OS bridge provides an abstraction from the interfaces and protocols used by the OS elements to be communicated with. Instructions and data are written to a master controller in the R-FPGA which then distributes this information to the according slaves implementing the interface to the OS elements. For read accesses, the information is collected from these slaves.

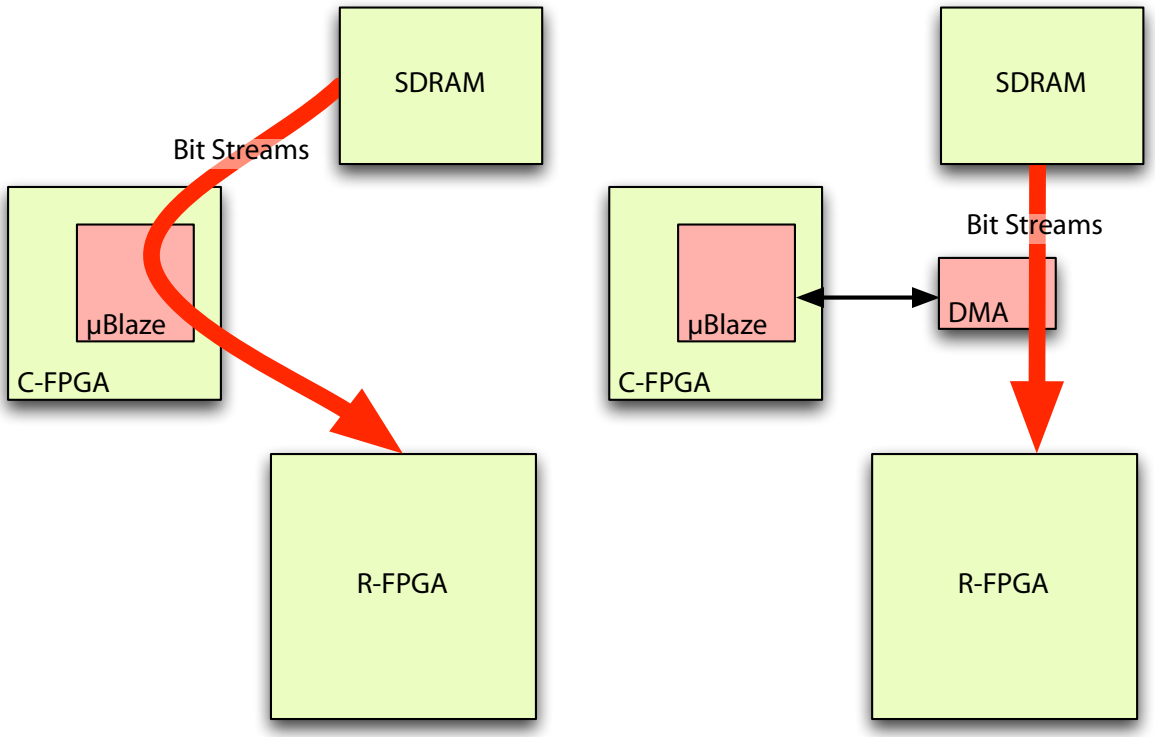
The OS bridge is documented in sections 7.14 and 7.6. As the documentation there is rather detailed, I don't dig any deeper into the OS Bridge's internals here.

6.3 Configuration of the R-FPGA

Bit streams to be configured into the R-FPGA can be uploaded to the system's SDRAM via ethernet. At the beginning of the SDRAM memory space, a table of contents holding information on all currently present full and partial bit streams is located.

The configuration process is currently implemented in software. The software reads a bit stream from the SDRAM and writes it to the R-FPGA using the SelectMAP port. This software approach is very slow: it takes about 2.5 seconds to configure a full bit stream. Furthermore, the CPU is loaded during the whole process. A solution to this problem is the use of a DMA controller which is delegated the configuration process. The DMA controller would directly read the bit stream from memory and write it to the R-FPGA autonomously. This controller is currently being developed in a further master's thesis. A graphical comparison of these two different approaches is given in figure 6-1.

The software routines performing the configuration are discussed in section 8.2.24.



(a) Software Approach

(b) DMA Approach

Figure 6-1: R-FPGA Configuration. *The current implementation of the R-FPGA configuration mechanism is a software approach with limited speed (a). A better and faster solution would be a DMA controller which is delegated the configuration process. Furthermore, the CPU is not loaded during the configuration.*

7 Hardware Documentation

Having discussed the conceptual ideas behind the various peripheral hardware components being attached to the process, this chapter shall present the hardware components actually designed for the XFBOARD OS. Sections 7.1 to 7.13 describe hardware components attached to the μBlaze in the C-FPGA, i.e. LMB and OPB cores, and section 7.14 describes hardware designed for the R-FGPA.

7.1 LMB BlockRAM Interface Controller

Core Name: `lmb_xfbram_if_cntlr`

7.1.1 Introduction

This core extends the core `lmb_bram_if_cntlr` by a function that traps all LMB write accesses that point to a location below a lower limit. These access violations raise an interrupt.

7.1.2 Parameters

All parameters are inherited from the core `lmb_bram_if_cntlr`. For your convenience, these parameters are still explained here.

The range specified by C_BASEADDR and C_HIGHADDR must comprise a complete, contiguous power-of-two range, such that range = 2^n, and the n least significant bits of C_BASEADDR must be zero. The decode mask determines which bits are used by the LMB decode logic to decode a valid access to LMB. If using EDK, this bit can be automatically set by `platgen` and users do not need to setup the value. The address mask indicates which bits are used in the LMB decode to decode that a

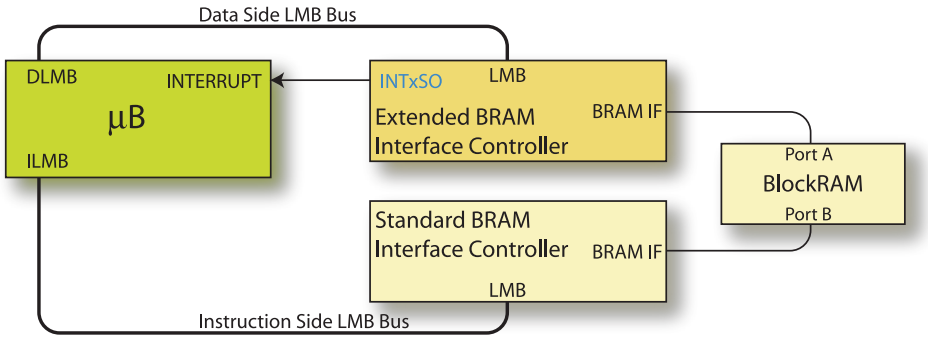


Figure 7-1: Example Connection Scheme. *This is an example on how to connect the µBlaze's two LMB busses to two BlockRAM interface controllers. In this setup, the data side LMB bus is protected from illegal write accesses to the BlockRAM*

valid address is present on the LMB. Any bits that are set to 1 in the mask indicate that the address bit in that position is used to decode a valid LMD access. All other address bits are considered don't cares for the purpose of decoding LMB accesses. The platform generation tool may limit the user's choice for the address mask: the most restrictive case is that only a single bit may be set in the mask. Consult the platform generation tool documentation and informational messages for more information.

7.1.3 I/O Signals

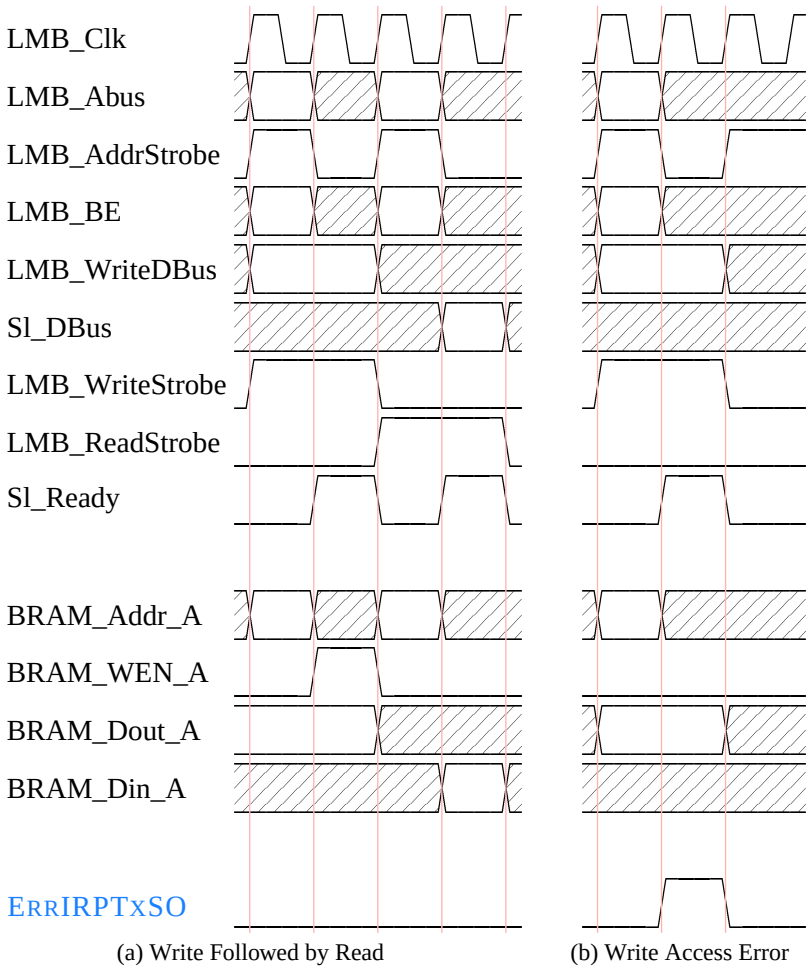
The added functionality of this interface controller implies that the pool of the LMB signals is extended by an interrupt line, the ERRIRPTXSO. This is a high active interrupt that is asserted high for one clock period whenever a LMB write access falls below a lower address limit. For sake of clarity, an example setup involving the data side and the instruction side of the µBlaze's LMB is displayed in figure 7-1. As the instruction side only performs read accesses, the standard controller is used for this side.

7.1.4 Core Operation

Read and write accesses are treated exactly the same way as with the core `lmb_bram_if_cntlr` as long as the address for write accesses don't fall under the lower address limit. This lower address limit can be written via LMB bus to the register in figure 7-3. This register can be written only once to prevent its content from being changed unwantedly.

As soon as a write access violation is detected, the interrupt is raised for one clock cycle, and the write enable line to the BlockRAM is forced to 0. The address that was to be written during this forbidden access is stored in a register that can be read from the software (figure 7-3).

Timing diagrams for allowed and forbidden accesses are depicted in figure 7-2.



(a) Write Followed by Read (b) Write Access Error

Figure 7-2: Example Waveforms. *On the left, a write access is followed by a read access. On the right, a write access error occurs and an the interrupt is triggered. The upper of signals are connected to the LMB bus, the lower group is connected to the BlockRAM.*

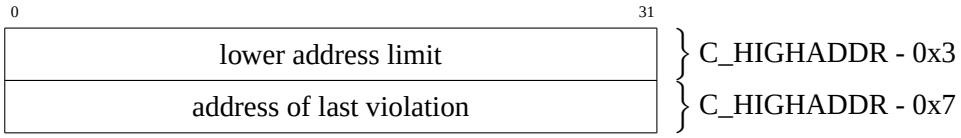


***Figure 7-3: Core Registers**. These registers can be read via the OPB bus. The address limit can be written only once, the address of the last violation is read-only-*

7.1.5 Driver

To assign an interrupt handler to the interrupt line, a driver supporting this feature is needed, e.g. `timerint`.

7.1.6 Software

In your program, you possibly want to protect the read-only section at the beginning from any write access. This can be done using this code snippet:

```
// at the beginning of your code, write the
// high address of the read-only section to the
// register containing the lower address limit
*((Xuint32*)(XPAR_DLMB_BRAM_HIGHADDR-3)) = (Xuint32)&_erodata;

// later, e.g. in an interrupt handler, you might
// want to print the address of the last violation
printf("Last access violation occurred at address 0x%08x",
     *((Xuint32*)(XPAR_DLMB_BRAM_HIGHADDR-7));
```

XPAR_DLMB_BRAM_HIGHADDR is assumed to be the high LMB address of this interface controller. The symbol `_erodata` gets defined at link time and points to the end of read-only data.

description	parameter name	allowable values	default value	VHDL type
LMB BRAM base address	C_BASEADDR	valid address	none	std_logic_vector
LMB BRAM high address	C_HIGHADDR	valid address	none	std_logic_vector
LMB data bus width	C_LMB_DWIDTH	32	32	integer
LMB address bus width	C_LMB_AWIDTH	32	32	integer
LMB decode mask	C_MASK	valid decode mask	0x008000000	std_logic_vector

Table 7-1: BlockRAM Interface Controller Parameters*. These are the parameters that can be set for the core. The data and address withs cannot be actually changed as they must fit the LMB bus architecture.*

7.2 LMB Text Display Driver

Core Name: `lmb_xfcvga`

7.2.1 Introduction

This VGA text display core is a custom-tailored LMB core for the C-FPGA being part of the X-Forces project. Using this core, text and simple ASCII graphic output can be displayed with 8 colors on a monitor. A display with these capabilities is adequate for a debugging terminal and a simple user interface to control and monitor the system's state. To be able to display an acceptably large amount of information, the trade-off between many colors and many pixels has been settled in favor of a high resolution.

7.2.2 Parameters

You may tailor this core by adjusting parameters listed in table 7-2. The default values for the scanning parameters defining the video mode (C_H_FRONT_PORCH through C_V_RIGHT_BORDER in table 7-2) are correct for the XGA mode described in section 7.2.6. The settings for the clock divider are adequate for a 50 MHz system clock. To support another video mode, you may change these parameters; however, the core has been tested with the default values only.

The parameters printed in dimmed colors are present only to let the core appear as a normal BlockRAM module (again, see section 7.2.6) and should be left untouched.

7.2.3 Insertion of the Core

As this core is attached to the LMB bus and shall be seen as a BRAM block, some additional remarks are due. To connect this core to the LMB, an instance of the LMB controller (lmb_bram_if_cntlr) is needed. The controller must be attached to the DLMB port of the processor. The address range of the controller is restricted to be a power of 2 and must be at least 0x1800 in size (see section 7.2.6 for a rationale). If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. For the high address, the n least significant bits must be 1, so the size of 0x1800 has to be expanded to the next valid value. A valid configuration would be 0x00700000 for the base address and 0x00701FFF for the high address.

As soon as both the LMB controller and the VGA core are inserted into your design, some manual adjustments to the project's `*.mhs` file have to be performed to attach the VGA core to the LMB controller.

1. open the `system.mhs` in an editor and find the instance of the LMB controller
2. add the line `BUS_INTERFACE BRAM_PORT = conn_vga` to the instance.

3. find the corresponding instance of the VGA core and add the line `BUS_INTERFACE TEXTMEM = conn_vga`. The string `conn_vga` is arbitrary, but it must be unique within the `*.mhs` file.

As the `system.mhs` has been edited outside the Platform Studio, you have to close your project and reopen it to make these changes active.

7.2.4 I/O Signals

The I/O signals for the VGA core are listed in table 7-3. Only five signals are needed to drive a CRT or TFT display. Please note the recommended ranges for the signals as they exactly fit to the number of pins available in the user constraints file for the C-FPGA!

7.2.5 Driver

No special driver is recommended, the driver `generic` may completely satisfy your needs.

7.2.6 VGA Core Operation

As mentioned above, the VGA core is designed to be seen as a BlockRAM module by the MicroBlaze. Therefore, this core is commonly connected to the DLMB of the CPU. When doing so, printing to the screen becomes as simple as assigning a variable a value in the C code. For a reference have a look at figure 7-4 where the display driver is used in a generic MicroBlaze environment; unnecessary details have been omitted in the drawing.

The core supports the XGA standard, which is an extension to the commonly known VGA:

pixels per line	:	1024
lines	:	768
pixel clock	:	75.0 MHz
horizontal frequency	:	56.48 kHz
vertical frequency	:	70.1 Hz

As this VGA core is meant to be driving the VGA connector attached to the C-FPGA on the XF-Board, only 8 colors can be displayed[1]. These colors are red, green, blue, cyan, magenta, yellow plus black and white.

To save BlockRAM, the core is designed to support text output only instead of giving access to each and every pixel on the screen. It consists of a text memory and a character lookup table. The text memory holds a 9 bit value for each character: 8 bits for the ASCII representation of the character,

[1] $8=2^3$: three 1-bit lines represent the color information

description	parameter name	allowable values	default	VHDL type
horizontal front porch	C_H_FRONT_PORCH	any pixel count	24	natural
horizontal sync pulse	C_H_SYNC	any pixel count	136	natural
horizontal back porch	C_H_BACK_PORCH	any pixel count	144	natural
horizontal left border	C_H_LEFT_BORDER	any pixel count	0	natural
horizontal active region	C_H_ACTIVE	any pixel count	1024	natural
horizontal right border	C_H_RIGHT_BORDER	any pixel count	0	natural
vertical front porch	C_V_FRONT_PORCH	any line count	3	natural
vertical sync pulse	C_V_SYNC	any line count	6	natural
vertical back porch	C_V_BACK_PORCH	any line count	29	natural
vertical left border	C_V_LEFT_BORDER	any line count	0	natural
vertical active region	C_V_ACTIVE	any line count	768	natural
vertical right border	C_V_RIGHT_BORDER	any line count	0	natural
total characters per line	C_CHARS_PER_LINE	$\leq$ C_H_ACTIVE / 8	128	natural
characters in text column	C_CHARS_IN_TEXT	note 1	64	natural
characters in graphics column	C_CHARS_IN_GRAPH	note 1	63	natural
number of text lines	C_NUMBER_OF_LINES	$\leq$ C_V_ACTIVE / 16	48	natural
clock divisor	C_CLK_DIVIDE	note 2	2	integer
clock multiplicator	C_CLK_MULTIPLY	note 2	3	integer
BlockRAM size	C_MEMSIZE	don't alter	8092	integer
BlockRAM data width	C_PORT_DWIDTH	don't alter	32	integer
BlockRAM address width	C_PORT_AWIDTH	don't alter	32	integer
BlockRAM No. of write enables	C_NUM_WE	don't alter	4	integer

note 1: C_CHARS_IN_TEXT + C_CHARS_IN_GRAPH = C_CHARS_PER_LINE - 1

note 2: the range of valid values highly depends on the system clock frequency

Table 7-2: Core Parameters. *These are the parameters that can be adjusted for this core. However, be sure that the parameter combination represents a valid display standard. The greyed-out parameters should be left untouched as they need to perfectly fit the LMB bus architecture, which is fixed.*

signal name	I/O	recommended width	description
RxDO	O	1	red color
GxDO	O	1	green color
BxDO	O	1	blue color
HSYNCxSBO	O	1	horizontal synchronization, active low
VSYNCxSBO	O	1	vertical synchronization, active low
RSTxRI	I	1	reset input

***Table 7-3: I/O Signals**. Except for the reset input, all signals are outputs that can be directly fed to the VGA / XGA display*

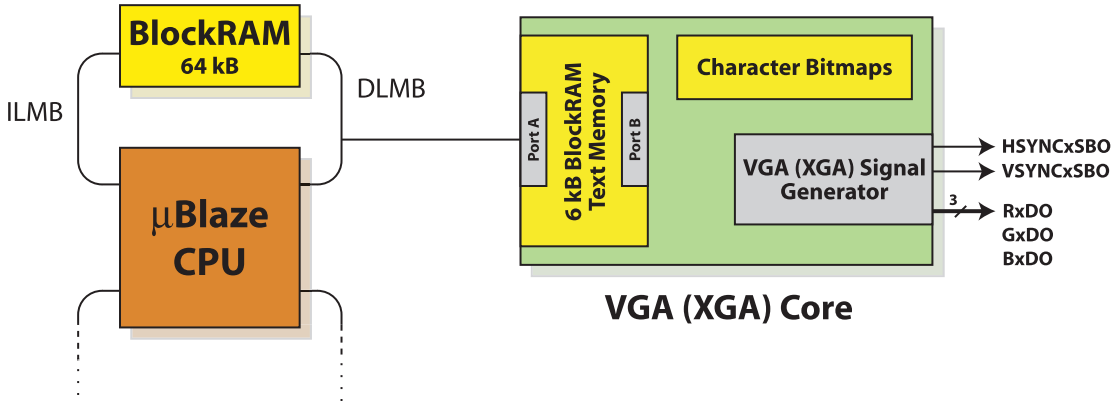


***Figure 7-4: Example Connection Scheme**. In this example, the VGA core's text memory is connected to the data side LMB bus of the µBlaze. The other signals are fed to a VGA connector.*

and an additional bit to choose either the primary or the secondary text color. In the character lookup table, the bitmaps for the characters are stored. As there are many control characters in the 8 bit ASCII character set, these control characters are replaced by custom bitmaps to be used for simple ASCII-art drawing, i.e. for graphical control and monitoring elements.

One character is 8 pixels in width and 16 pixels in height, which results in 128 characters per line and 48 text lines. As a consequence, a total of 6144 characters can be displayed on the screen. Given the coordinates (x,y)[2] of a pixel, its value can be calculated and looked up the following way:

- get index $i_{textmem}$ into text memory

[2](0,0) is the top left corner

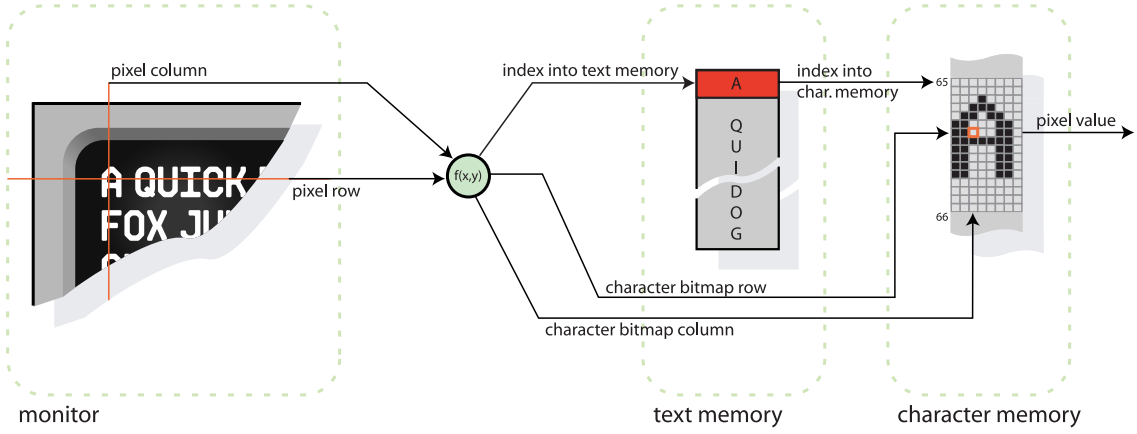


***Figure 7-5: Core Internals**. This figure gives an overview of the VGA core's internals. Given the pixel coordinates, an index into text memory is calculated. The pixel value for these coordinates is then read from the bitmap for the character being stored at this index.*

$$i_{textmem} = \left\lfloor \frac{x}{8} \right\rfloor + 128 \cdot \left\lfloor \frac{y}{16} \right\rfloor$$

- the ASCII number of the character to display is stored in the text memory at this index
- then the pixel value at row r and column c in the bitmap representing this ASCII number is returned

$$r = x \bmod 8$$

$$c = y \bmod 16$$

Refer to figure 7-5 for an illustration of these steps.

As mentioned above, this approach reduces memory usage. To have full control over all pixels on the screen,

$$3 \text{ bits} \cdot 1024 \text{ pixels} \cdot 768 \text{ lines} = 2'359'296 \text{ bits} = 288 \text{ kB}$$

of memory would be needed. Splitting up into text memory and character bitmap memory, this number shrinks down to a mere

$$6144 \text{ characters} \cdot 9 \text{ bits} + 256 \text{ characters} \cdot 8 \text{ columns} \cdot 16 \text{ rows} = 88'064 \text{ bits} = 10.75 \text{ kB}$$

of memory and LUT resources.

offset	description
0	color configuration: $b_8 \ldots b_6$: background color $b_5 \ldots b_3$: secondary text color $b_2 \ldots b_0$: primary text color
4	scrolling offset
8	cursor position, lower bits
12	cursor position, upper bits

***Table 7-4: Core Registers**. These are the registers that can be configure via the LMB bus. The registers are appended to the end of the address range of the text memory, so their offsets are relative to the end of this range.*

To have two more or less independent screen partitions for text and graphics, the display is split into two columns. The left side of the screen is used for text display, therefore a simple scrolling mechanism is implemented for this column. Scrolling is performed automatically whenever the text reaches the bottom of the screen. Assuming the default values for the parameters in table 7-2, the characters for the text column are written to the first $64 \cdot 48 = 3072$ positions in the text memory, i.e. the address range from C_BASEADDR to C_BASEADDR$+4\cdot$(C_CHARS_IN_TEXT$\cdot$C_NUMBER_OF_LINES-1); the factor 4 is inserted due to the MicroBlaze's 32-bit addressing scheme. This memory section is written circularly: whenever the end of the memory section is reached, it starts being overwritten at the beginning. To emulate scrolling, the memory section is then displayed with an offset. There is no scrollback buffer, so only the text currently displayed remains in the memory. In the text column, a blinking cursor can be displayed. The cursor's position must be updated in the software.

The right side of the screen is used for graphics display. As this core is character oriented, graphics must be composed from ASCII characters. For the graphics column, no scrolling is implemented. The blinking cursor can be moved to the graphics side of the display too. The characters for the graphics column are located in the address range C_BASEADDR $+ 4 \cdot$ (C_CHARS_IN_TEXT$\cdot$C_NUMBER_OF_LINES) to C_BASEADDR$+4\cdot$((C_CHARS_IN_TEXT$+$ C_CHARS_IN_GRAPH) $\cdot$ C_NUMBER_OF_LINES $- 1$).

One character column is left empty to separate the text partition from the graphics partition. As a consequence, C_NUMBER_OF_LINES memory entries at the end of the memory remain empty. These locations are used to store some configuration data. The memory locations for the configuration are listed in table 7-4. The offset is relative to the address C_BASEADDR $+ 4 \cdot$ (C_CHARS_IN_TEXT $+$ C_CHARS_IN_GRAPH) $\cdot$ C_NUMBER_OF_LINES. The configuration settings may be altered at runtime writing to the corresponding address from the C program.

7.2.7 Software

Some basic functions to write to the screen and set up configuration data are given in the following listing. The code in this listing is working, but you may want to extend it as it only covers the most trivial aspects of a text display:

- initializing the display
- printing a character
- printing newlines
- updating the cursor
- setting up colors

Some functions are referred to in the code without being implemented. They are marked with the keyword *tbi*[3] in the comment.

```
#include "ctype.h"
#include "xbasic_types.h"
#include "xparameters.h"

#define CHARS_PER_LINE 128
#define CHARS_IN_TEXT 64
#define CHARS_IN_GRAPH 63
#define NUMBER_OF_LINES 48
#define TEXT_MEM_HIGHADDR XPAR_VGA_BASEADDR + NUMBER_OF_LINES * \
                    (CHARS_IN_TEXT + CHARS_IN_GRAPH)*4
#define COL_REG     TEXT_MEM_HIGHADDR
#define LIN_REG     TEXT_MEM_HIGHADDR + 4
#define CUR_REG     TEXT_MEM_HIGHADDR + 8

volatile Xuint32 *textMemory;
Xuint32 currentLine, colorMask, textMemLineOffset = 0;
Xboolean incrementLineOffset = XFALSE;

void vga_updatecursor(){
  volatile Xuint32 *memAddr;
  memAddr = (Xuint32*)(CUR_REG); // here the cursor position is stored
  *memAddr = ((Xuint32)textMemory - XPAR_VGA_BASEADDR) / 4; // write lower bits
  memAddr++;
  *memAddr = (((Xuint32)textMemory - XPAR_VGA_BASEADDR) / 4) >> 9; // write upper bits
}

void vga_newline(){
  Xuint32 *memAddr;
  currentLine++;
```

[3] to be implemented

```
30   currentLine %= NUMBER_OF_LINES; // keep inside memory range
31   if(currentLine == textMemLineOffset) // we reached the bottom of the screen
32     incrementLineOffset = XTRUE; // activate scrolling
33   currentLine %= NUMBER_OF_LINES; // keep inside memory range
34   if (incrementLineOffset){     // we are scrolling
35     textMemLineOffset++;        // increment scroll offset
36     vga_clearline();            // erase the line (tbi)
37     memAddr = (Xuint32*)(LIN_REG); // here the scroll offset is located
38     *memAddr = textMemLineOffset; // write offset to offset register
39   }
40   textMemory = (Xuint32*)(XPAR_VGA_BASEADDR + 4 * currentLine * CHARS_IN_TEXT);
41 }
42
43 void vga_init(){
44   textMemory = (Xuint32*)XPAR_VGA_BASEADDR;
45   while((Xuint32)textMemory < XPAR_VGA_BASEADDR +
46      4 * NUMBER_OF_LINES * CHARS_PER_LINE){
47     *textMemory=(Xuint32)0;           // init memory with 0
48      textMemory++;
49   }
50   textMemory = (Xuint32*)XPAR_VGA_BASEADDR; // reset text insertion point
51   currentLine = 0;                  // set up line counter
52   colorMask = 0;                    // reset color mask
53   vga_setupColors(0x7,0x2,0x0);     // setup colors : white , green , black
54 }
55
56 void vga_setupColors(Xuint32 textcol1, Xuint32 textcol2, Xuint32 bgcol){
57   volatile Xuint32 *memAddr;
58   Xuint32 color;
59   memAddr = (Xuint32*)(COL_REG);          // here the color config is stored
60   color  = textcol1 + (textcol2 << 3) + (bgcol << 6); // compose value
61   *memAddr = color;                       // save value
62 }
63
64 void vga_putc(unsigned char c){
65   if (c == '\n' || c == '\r')          // treat newlines and backspaces
66      vga_newline();
67    else if (c == '\b')
68      vga_backspace(); // tbi
69    else{
70      *textMemory = (Xuint32)c + colorMask; // write character to screen
71      textMemory++;
72      if((Xuint32)textMemory % (CHARS_IN_TEXT * 4) == 0)// treat line-wrapping correctly
73        vga_newline();
74    }
75    vga_updatecursor();
76 }
77
78 void vga_setColor(int col){
79   if(col == 0)
80     colorMask = 0; // primary color
81   else
82     colorMask = 256; // secondary color, set 9th bit
83 }
```

7.2.8 Outlook

In a future release, additional configuration possibilities may be introduced, i.e. separate color setup for the graphics column. Moving the character bitmap to BlockRAM instead of keeping a hard-wired LUT might be interesting too. An option for installing custom fonts might be implemented then.

7.3 OPB Clock Generator

Core Name: `opb_xfcclockman`

7.3.1 Introduction

This clock generator derives four additional clocks from the system clock and delivers them to the R-FPGA. The clock frequencies can be set in software, and the clock signals can be disabled.

7.3.2 Parameters

You can tailor this core by adjusting the parameters listed in table 7-5. Most of the parameters are described in more detail in section 7.3.4. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. As there are only 4 registers present in the system to hold the configuration data, only 4 addresses need to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0xF is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF00000F.

7.3.3 I/O Signals

There is only one output signal: the four clock lines are combined to `CLKxCO`, which is defined on the entity as `std_logic_vector(0 to 3)`.

7.3.4 Core Operation

To generate the 4 clock signals, the basic clock is divided using a counter. Deriving the new clock signal using a counter does not violate the basic principles of synchronous, single-edge triggered design as the signal is not used as a trigger for any flip-flop inside the R-FPGA. It is fed to a clock input of the C-FPGA where a clock buffer is available to create a valid clock net.

As an example, the calculation of the frequency of one such clock is done next. Assuming a base clock frequency f_{sys} and a divisor value d_0 stored in the register, the frequency f_{out0} output at the pin for clock 0 is

$$f_{out0} = \frac{f_{sys}}{2 \cdot (d_0 + 1)}$$

The counter counts up to d_0 and then restarts with zero; therefore d_0 must be increased by 1 to get the divisor. An additional factor 2 is included because output signal gets toggled each time the counter reaches d_0. To get a full clock period, the signal must be toggled twice. The offset to d_0 gets

description	parameter name	allowable values	default value	VHDL type
SRAM memory base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
SRAM memory high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector
Maximum width for divider Constants	C_DIV_MAX_WIDTH	any number between (and including) 1 and 31	5	integer
Initial divider value	C_DEFAULT_DIV	any value that can be represented with C_DIV_MAX_WIDTH bits	5	integer
Defaults for enable bits	C_DEFAULT_ENA	any value from 0000 to 1111	1111	std_logic_vector
System clock multiplication factor	C_BASE_MULTIPLY	depends on the combination of base frequency, multiplicator and divisor values. Refer to Xilinx' DCM documentation.	5	integer
System clock divisor	C_BASE_MULTIPLY	depends on the combination of base frequency, multiplicator and divisor values. Refer to Xilinx' DCM documentation.	1	integer
System clock period	C_PERIOD	the period in nanoseconds	20.0	real
Include DCM flag	C_INCLUDE_DCM		true	boolean

***Table 7-5: Core Parameters**. This is a list of all parameters that may be adjusted for the clock generator core. For details on the valid ranges of the divisor and the multiplication factor refer to XILINX' documentation of the DCMs in the Virtex-II user guide[32].*

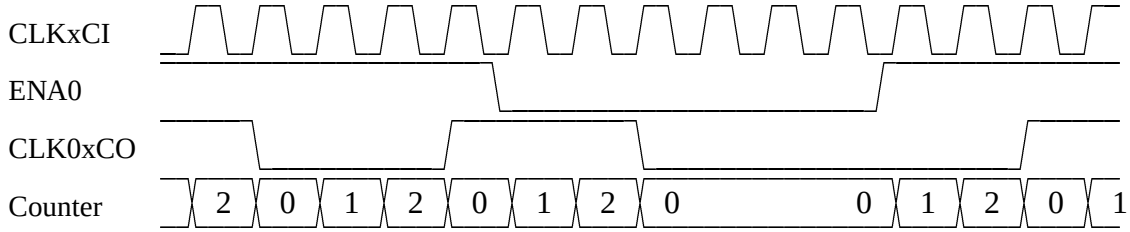


***Figure 7-6: Example Waveforms**. This is an example for clock 0, which is configured to divide the system clock by $2 \cdot 3$, which means that $d_0 = 2$. The divided clock is being disabled and then enabled again. The current period is being terminated before actually being disabled.*

compensated when being set via software, but the compensation does not apply for the initial value C_DEFAULT_DIV. If you want to have the clock divided by $2 \cdot 6$, you need to set C_DEFAULT_DIV to 5; if you want to set the value using the software, you need to write a 6 to the register.

If you know that you are not going to use large divisor values, you may want to use a smaller value for C_DIV_MAX_WIDTH which basically defines the counter width and thus significantly affects the combinational path from the output of the counter registers to its input. The larger C_DIV_MAX_WIDTH is, the more logic levels are used for the adder and the compare logic, which results in a lower maximum operation frequency of the clock generator.

To get clock frequencies above the system clock frequency and to have a better frequency in lower frequencies, a digital clock manager (DCM) can be used by setting C_INCLUDE_DCM to *true*. Using a DCM, the base frequency can by multiplied by C_BASE_MULTIPLY and divided by C_BASE_DIVIDE. Assuming a multiplicator n and a divisor d the above equation can be rewritten:

$$f_{O0} = f_{sys} \cdot \frac{n}{d} \cdot \frac{1}{2 \cdot (d_0 + 1)}$$

As mentioned in the introduction, the 4 clock signals can be enabled and disabled separately. To guarantee valid clock signals, the current clock period is being finished when a clock is changed from the enabled to the disabled state. An example is given in figure 7-6, where clock 0 is being disabled and then enabled again.

7.3.5 Software

To configure the clock generator, you simply have to write a 32 bit word to the OPB bus. Bit 0 (the MSB in the μBlaze architecture) is the enable flag, the remaining bits represent the clock divisor as shown in figure 7-7.

```
*((Xuint32*)(XPAR_CLOCK_BASEADDR + 4)) = 0x80000000 || 9 ;
```

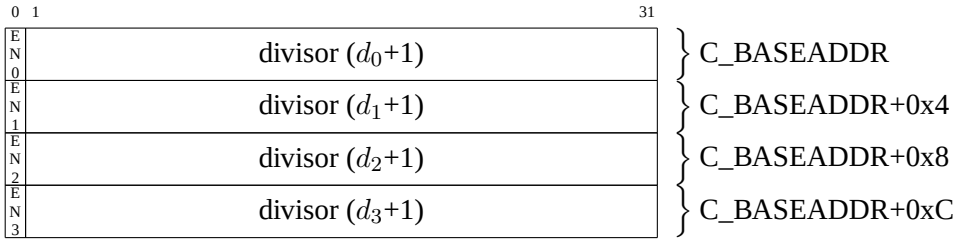


***Figure 7-7: Core Registers**. These configuration registers hold the divisor constants and the enable flags for all four clock outputs. The registers can be read from and written to via the OPB bus from within the software.*

is an example code line that sets the clock generator for clock output 1 to divide the frequency output by the DCM by $2 \cdot 9$. The generator is enabled by setting bit 0 which can be achieved by OR'ing the value 9 with 0x80000000. Disabling the clock, but keeping the divisor 9 in the register (for whatever reason...) would have resulted in a line similar to

```
*((Xuint32*)(XPAR_CLOCK_BASEADDR + 4)) = 9 ;
```

Writing these code examples, I assumed that XPAR_CLOCK_BASEADDR is the base address of the clock generator.

7.3.6 Outlook

This version of the clock generator only includes very basic features. Extending the generator to support enabling clocks for a certain number of periods and then have them disabled again by the generator would probably be interesting and useful.

7.4 OPB Test-And-Set Lock

Core Name: `opb_xfclock`

7.4.1 Introduction

This core implements a hardware structure that can be tested and set atomically. Such structures are often used in systems where one or more critical code section exist; in such systems it must be guaranteed that only one process at a time is allowed to have access to such a critical section.

7.4.2 Parameters

The only parameters that must be set for this core are those listed in table 7-6. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. As there is only 1 register which represents the lock variable, only 1 address needs to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0x4 is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF000003. Detailed documentation on the MIDI protocol can be found in [8].

description	parameter name	allowable values	default value	VHDL type
Test-and-set core base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
Test-and-Set core high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector

***Table 7-6: Core Parameters**. The only parameters that must be set define the test-and-set core's OPB address range*

7.4.3 I/O Signals

This is an internally used core; therefore no I/O signals are used.

7.4.4 Core Operation

The lock variable is implemented using a register that can be read from and written to. Reading from the register performs a test-and-set access, writing to the register results in a clear access. When the core is written to, two scenarios may be encountered:

- The lock is not set

  If the lock is not set at the moment the register is being read, a 0 is returned, and at the same time, the register's content is set to 1, which means that the lock is set.

- The lock is set

  If the lock is set at the moment the register is being read, a 1 is returned and the register's content is not altered

To unlock, the register can be written with a any value, which sets its content to 0.

7.4.5 Outlook

In a future version, this core may implement more than one lock variable. Also, it might be useful not to restrict the range of values of the register's content to 0 and 1 only, e.g. to store the PID of the process requesting the lock.

7.5 OPB MIDI Interface

Core Name: `opb_xfcmidi`

7.5.1 Introduction

This OPB Midi interface simply reads in serial MIDI data and writes them parallel into a FiFo which can be read from the OPB bus. It does not perform any data filtering, therefore all MIDI bytes for all 16 MIDI channels are catched. Any filtering has to be performed in software. This MIDI interface is based on the MIDI synthesizer designed in [11].

7.5.2 Parameters

The only parameters that must be set for this core are those listed in table 7-7. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. As this core is sort of a FiFo which can be read, only one address needs to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0x4 is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF000003.

The FiFo address width C_FIFO_AWIDTH governs the number n of entries that can be stored in the FiFo:

$$n = 2^{\text{C\_FIFO\_AWIDTH}} - 1$$

The clock frequency C_CLOCK_FREQ is needed to synchronize the receiver to the clock used in the MIDI standard, which is 31.25 kHz.

description	parameter name	allowable values	default value	VHDL type
MIDI core base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
MIDI core high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector
FiFo address width	C_FIFO_AWIDTH	unbounded; only limited by the maximum number of registers available	6	natural
System Clock Frequency	C_CLOCK_FREQ	the clock frequency in Hz	50000000	integer

Table 7-7: Core Parameters. *These are the parameters that must be set for the MIDI core.*

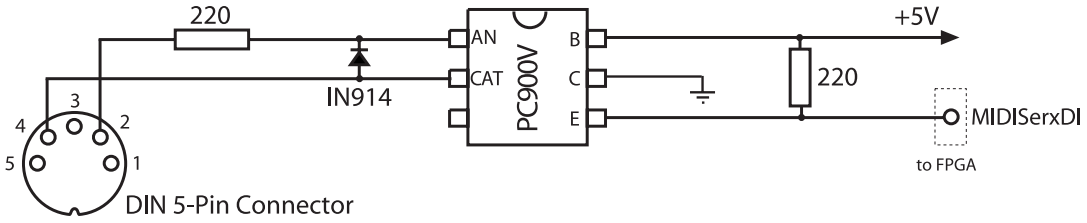


***Figure 7-8: MIDI Schematic**. This schematic is a suggestion on how to connect the MIDI cable to the FPGA pin. The PC900V is an optocoupler used as a current-to-voltage converter. It also galvanically isolates the MIDI system from the FPGA.*

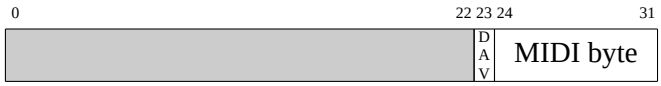


***Figure 7-9: Core Register**. The data word that can be read via OPB bus contains the actual MIDI byte plus a data-valid flag.*

7.5.3 I/O Signals

The MIDI interface uses only one external input signal that feeds the MIDI signal, MIDISERXDI, which is of `std_logic`. The MIDI signal cannot be directly fed to the FPGA; instead, an optocoupler is needed to convert the corrent of about 5 mA delivered by the MIDI cable into a voltage. Also, the optocoupler serves to galvanically isolate the MIDI cable from the FPGA to prevent possible electrical problems. An example circuit is given in figure 7-8.

7.5.4 Core Operation

MIDI data being fed serially to the core is converted to bytes using a shift register and then written to a FiFo. If the FiFo is full, newly received bytes are discarded. When the core gets an OPB read access, it tries to pop a value off the FiFo. If the FiFo is not empty, this value gets returned in bits 24 to 31 of the data bus, and bit 23, being the data valid flag, is set. If the FiFo is empty, bits 24 to 31 are cleared; the data valid flag is cleared too. This flag is needed due to the fact that 0x00 is a valid MIDI byte too. Bits 0 to 22 are never used, as shown in figure 7-9.

7.5.5 Software

Reading the FiFo from within the software is easy. The code snippet below waits for MIDI data and prints the received bytes. In the code, the core is assumed to be assigned the base address XPAR_MIDI_BASEADDR.

```
Xuint32 data;
Xuint8 midibyte;
while(1){
  do{ // loop until valid MIDI data are available
    data = *((Xuint32*)XPAR_MIDI_BASEADDR);
  }while(!(data & 0x100)); // the data valid flag: bit 23
  // strip the data valid flag
  midibyte = data & 0xFF;
  // print the MIDI byte
  printf("Received MIDI byte: 0x%02x\n",midibyte);
}
```

7.6 OPB OS Bridge

Core Name: `opb_xfcosbridge`

7.6.1 Introduction

The OS Bridge is meant as a communication interface between C-FPGA and R-FPGA. Using a simple instruction set, data and configurations can be written to and read from hardware modules on the R-FPGA, e.g. FiFos could be monitored and dumped. This OPB core represents the C-FPGA part of the OS Bridge. Documentation on the R-FPGA part can be found in section 7.14. The OS bridge has been introduced in [19].

7.6.2 Parameters

The only parameters that must be set for this core are those listed in table 7-8. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. The core needs to be assigned an address range of at least 256 addresses (see section 7.6.4 for a rationale), which means that $n = 8$. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF0000FF.

description	parameter name	allowable values	default value	VHDL type
OS bridge base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
OS bridge high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector

Table 7-8: Core Parameters. *An address range of minimal size 0x100 must be set for the OS Bridge.*

7.6.3 I/O Signals

The signals needed by this core are listed in table 7-9. OSBxDIO is a bidirectional signal used to write instructions and data to the R-FPGA and read data from the R-FPGA. When instructions or data are written, OSBWxEO is asserted to tell the OS bridge counterpart present in the R-FPGA that data is being written. To prevent this counterpart from driving the data bus, the tristate signal OSBTrstexTO connected to the R-FPGA's output buffers is asserted too. When data is being read, OSBTrstexTO is deasserted to allow the R-FPGA to drive the bus. The R-FPGA then asserts OSBRxEI as soon as the data is ready to be fetched from the bus.

signal name	I/O	recommended width	description
OSBxDIO	IO	16	data / instruction bus
OSBTRSTEXTO	O	1	tristate control signal
OSBWxEO	O	1	write enable, active high
OSBRxEI	I	1	read enable, active high

***Table 7-9: Core Signals**. All I/O signals of the OS bridge are directly connected to the R-FPGA via the system's GPIO bus. The CPU is the master and therefore controls the tristate signal.*

7.6.4 Core Operation

The core basically translates the 32 bit accesses over the OPB bus into 16 bit accesses over the OS bridge. These 16 bit accesses are performed in a sequential manner.

For writing to the R-FPGA, there are 3 command types which only differ in the amount of data being written. The opcodes used are 8 bits in length. Given the bus width of 16 bits, there are 8 bits remaining which can be used to transfer data. If more than 8 bits need to be written, one or more additional 16-bit words are transferred to the R-FPGA. The 3 command types (see figure 7-10) are as following:

type 0 commands consist of a single word sent over the OS bridge. The upper 8 bits represent the opcode, the lower 8 bits are the data payload.

type 1 commands are used to transfer up to 24 bits of data. Type 1 commands are transmitted using 2 OS bridge words. The upper 8 bits of the first word again represent the opcode, the lower 8 bits are the most significant byte of the 24 bits of data, and the second word consists of the two least significant bytes.

type 2 commands can write up to 32 bits of data, which is the maximum amount of date that can be written using a single instruction. Type 2 commands result in 3 OS bridge words being written to the bus. The upper 8 bits of the first word contain the opcode, the lower 8 bits are not used. Word 2 and word 3 contain the upper halfword and the lower halfword of the data being transmitted, respectively.

The command type is encoded in the two most significant bits of the opcode, i.e. `00`XXXXXX for type 0 commands, `01`XXXXXX for type 1 commands, and `10`XXXXXX for type 2 commands.

The timing diagram for a type 2 write command is depicted in figure 7-12. Here, a base address of 0xF0000000 is assumed, and the opcode 0x80 is executed, writing the random 32 bit word 0x0600C0C3 to the OS bridge bus.

For reading from the R-FPGA, there is only one command that can get up to a maximum of 16 bits

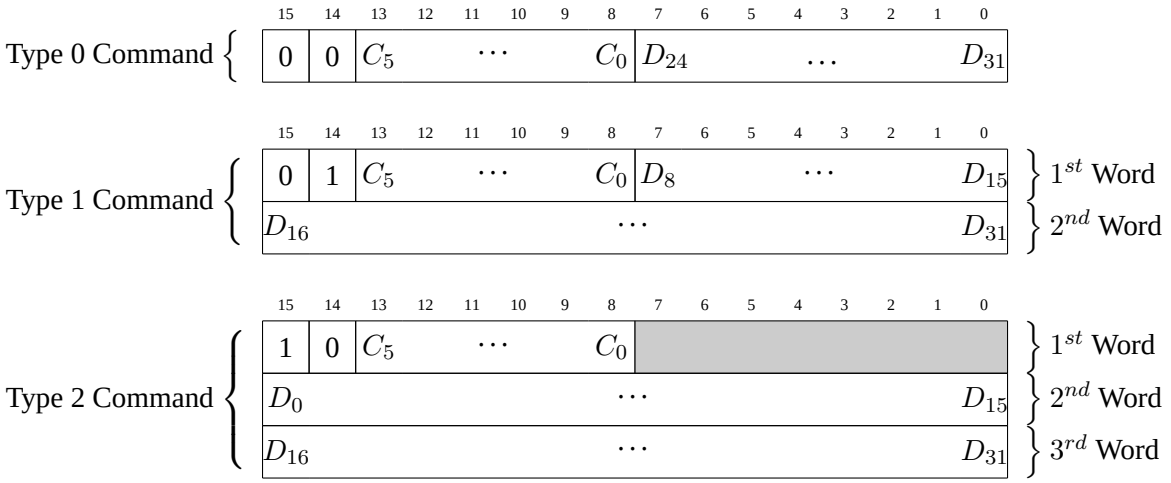


***Figure 7-10: OSB Write Comands**. There are 3 types of commands to write data to the R-FPGA using the OS bridge. The difference between the commands is the number of words being written. The type is deducted from the 2 most significant bits of the command's opcode.*

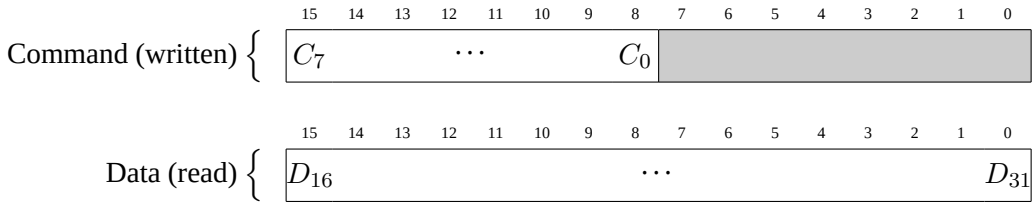


***Figure 7-11: OSB Read Command**. Only one command is defined to read data from the R-FPGA using the OS bridge. With this command, 16 bits of data can be read.*

of data. First, the 8-bit opcode is written to the bus, then the core waits for the R-FPGA's read enable signal and then reads 16 bits of data from the bus.

A timing diagram for a read command is given in figure 7-13, again with a base address 0xF0000000. The opcode is 0x81, and 0xAF is an example how the R-FPGA might answer this request.

Now let's focus on a basic problem. A write access to the OPB is done using a variable assignment in C, which then is translated into a `sw` instruction in assembly language. Similarly, a read access is translated into a `lw` instruction, which does not take any arguments except the address being read and the register being written with the value read at this address. The specification described above states that an opcode must be written to the bus before the actual data can be read. To avoid an additional write instruction that writes the opcode to the bus before every read instruction, a simple trick is used: the instruction being written in a read access is encoded in the address to be read. The OPB address A_{OPB} being read can be calculated using

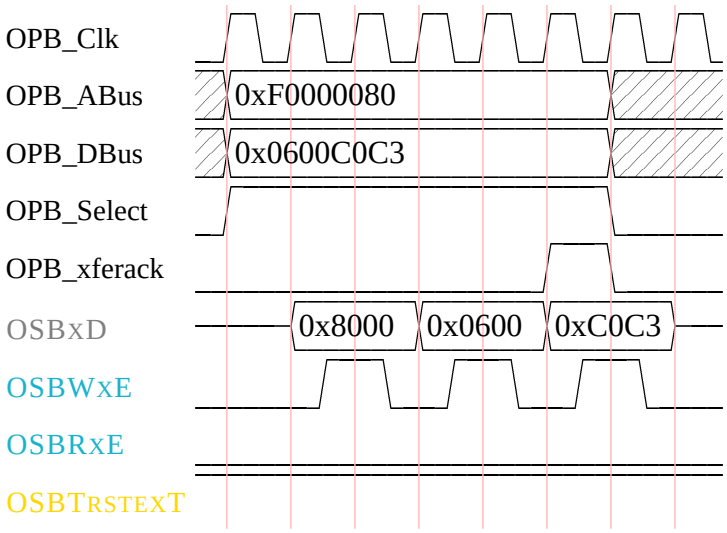


***Figure 7-12: Write Waveforms**. Example waveforms of a OS bridge write cycle using a type 2 command is given here. A 16-bit word containing the opcode is sent, then two such words containing data are written to the bus.*

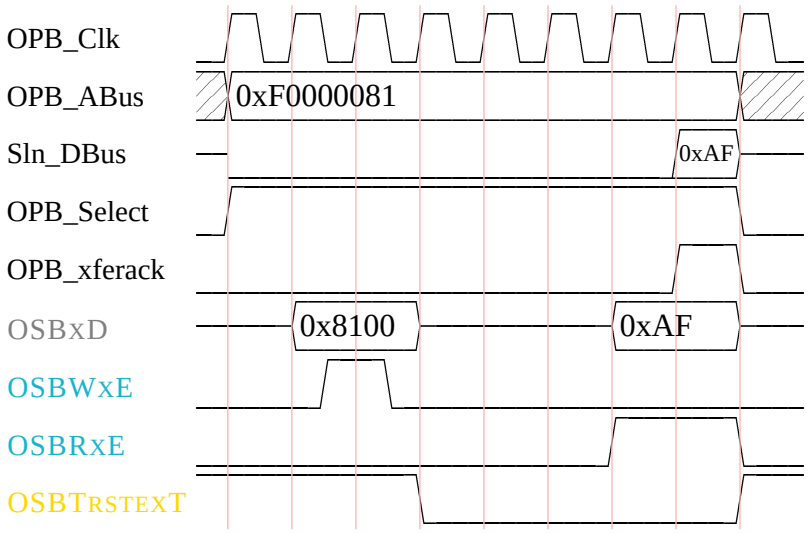


***Figure 7-13: Read Waveforms**. 16 bits of data are being read. First, the opcode is sent, then the bus is set into read direction, and the data from the R-FPGA are being awaited.*

$$A_{OPB} = A_{base} + opcode$$

with A_{base} being the base address and $opcode$ being the opcode. Thanks to the fact that in section 7.6.2 it has been stated that the 8 least significant bits of the base address need to be 0, the address can be calculated more easily (easy is here meant in terms of hardware complexity)

$$A_{OPB} = A_{base}(0 \text{ to } 23) \ \& \ opcode(0 \text{ to } 7)$$

written in pseudo-HDL with $\&$ being the concatenation operator and $(a \text{ to } b)$ being the bit selection operator extracting bits a to b. This simple trick also improves write accesses to the OS bridge because now accesses with an 8-bit opcode *and* 32 (instead of 24) bits of data are feasible.

7.6.5 Software

Writing to and reading from the R-FPGA using the OS bridge is as simple as writing to or reading from a single OPB address. One possibility is to use two simple macros for reading and writing:

```
1 // three example opcodes
2 #define OPCODE_WRITE_SRAM 0x81 // type 2 opcode
3 #define OPCODE_WRITE_GPIO 0x32 // type 0 opcode
4 #define OPCODE_READ_GPIO 0x32 // read opcode
5 // macros for reading and writing
6 #define OSBRIDGE_WRITE(opcode, value) \
7       *((Xuint32*)(XPAR_OSBRIDGE_BASEADDR | opcode) = value
8 #define OSBRIDGE_READ(opcode) \
9       *((Xuint32*)(XPAR_OSBRIDGE_BASEADDR | opcode)
```

These example macros are given with the assumption that XPAR_OSBRIDGE_BASEADDR holds the base address of the OS bridge OPB core.

7.6.6 Outlook

Currently, the communication over the OS bridge is rather slow: 6 clock cycles are needed to transfer an opcode and 32 bits of data. With some changes in the timing of this core and its R-FPGA counterpart, this number might be reduced to 3 clock cycles.

7.7 OPB PS/2 Keyboard Driver

Core Name: `opb_xfcps2key`

7.7.1 Introduction

This OPB keyboard driver simply reads in serial PS/2 scan codes and writes them parallel into a FiFo which can be read from the OPB bus. Apart from that, it generates a level-sensitive, high active interrupt when the scan code for the escape key has been detected.

7.7.2 Parameters

description	parameter name	allowable values	default value	VHDL type
Keyboard core base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
Keyboard core high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector
FiFo address width	C_FIFO_AWIDTH	unbounded; only limited by the maximum number of registers available	6	natural
FSM reset period	C_RESET_PERIOD	the number of clock periods after which the FSM is reset	8192	integer

***Table 7-10: Core Parameters**. These are the parameters that must be set for the PS/2 keyboard interface.*

The only parameters that must be set for this core are those listed in table 7-10. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. As this core is sort of a FiFo which can be read, only one address needs to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0x4 is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF000003.

The FiFo address width C_FIFO_AWIDTH governs the number n of entries that can be stored in the FiFo:

$$n = 2^{\text{C\_FIFO\_AWIDTH}} - 1$$

signal name	I/O	recommended width	description
PS2CLKxCI	I	1	PS/2 clock input
PS2DATxDI	I	1	PS/2 data
INTERRUPTxSO	O	1	interrupt line, level-sensitive, active high
INTACKxSI	I	1	interrupt acknowledge

***Table 7-11: Core Signals**. The core has 2 PS/2 signals. The interrupt line and its acknowledge signal are used to inform the µBlaze about the escape key being pressed.*

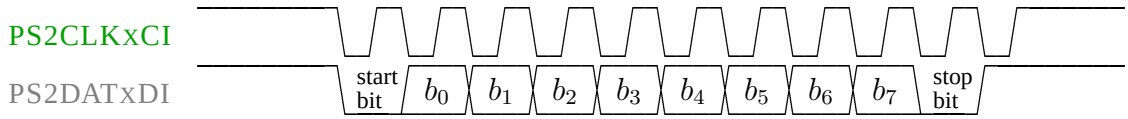


***Figure 7-14: PS/2 Waveforms**. As we only want to read data from the keyboard, the protocol is really simple: 8 bits with a start and stop bit are sent over the data line, alongside with a synchronisation clock only being active during the data transfer.*

7.7.3 I/O Signals

The PS2KBD interface uses two external input signals that feed the PS/2 data signal and the PS/2 clock. Additionally, internal signals for interrupt handling are available. These signals are listed in table 7-11.

7.7.4 Core Operation

PS/2 scan codes being fed serially to the core are sampled and then converted to bytes using a shift register implemented as an FSM that waits for the start bit and then shifts in the following 8 bits. These bytes are then written to a FiFo. When the FiFo is full, newly received scan codes are discarded. As there exists no scan code with the value 0x00, this value is returned whenever the FiFo is empty.

As the PS/2 keyboard sends a clock along with the data, no special synchronization has to be done; the core works with all clocks that are significantly faster than the PS/2 clock which is at 10.9 - 16.7 kHz. A timing diagram explaining how the data is sent using the PS/2 protocol is drawn in figure 7-14. According to the PS/2 standard, the devices at both ends of the PS/2 lines are allowed to write and read data; sensing is done using the clock line. Fortunately, we only want to read data from the keyboard, so no sensing is needed.

If the PS/2 data signal is too noisy or contains errors, the FSM might be in a bogus state. As a remedy, the FSM is forced back into its init state every C_RESET_PERIOD cycles.

Whenever a scan code for the escape key is detected, an interrupt is generated. The interrupt stays asserted until it is being acknowledged using INTACKxSI.

7.7.5 Software

Reading the FiFo from within the software is easy. The code snippet below waits for PS/2 data and prints the received bytes. In the code, the core is assumed to be assigned the base address XPAR_KEYBOARD_BASEADDR.

```
char scancode;
while(1){
  do{ // loop until a scan code is available
    scancode = *((Xuint32*)XPAR_KEYBOARD_BASEADDR);
  }while(!scancode);
  // print the scan code
  printf("Received scan code: 0x%02x\n",scancode);
}
```

As the PS/2 core returns scan codes and not the ASCII code for the keys being pressed, you might want to implement a FSM that interprets the scan codes received and maps them to your keyboard layout.

7.7.6 Outlook

Sometimes a wrong scan code gets detected. At the time writing, it is not known whether this is a bug on the hardware side or on the software side. Some time should be spent to track down this bug.

7.8 OPB Register Watcher

Core Name: `opb_xfcregwatch`

7.8.1 Introduction

The OPB register watcher monitors the value of a μBlaze register and raises an interrupt if the value of the register is outside of a certain range. The range of allowed values can be configured at runtime from the software. As an example, this core can be used to detect stack violations by looking at the stack pointer.

7.8.2 Parameters

description	parameter name	allowable values	default value	VHDL type
Register watcher base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
Register watcher high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector
Hard lower limit of the address range to be watched	C_HARD_LOWER_LIMIT	any valid address $<$ C_HARD_UPPER_LIMIT	0x00000000	std_logic_vector
Hard upper limit of the address range to be watched	C_HARD_UPPER_LIMIT	any valid address $>$ C_HARD_LOWER_LIMIT	0xFFFFFFFF	std_logic_vector
Register number to be watched	C_REG_NUM	any value from 0x00 to 0x1F	0x01	std_logic_vector

***Table 7-12: Core Parameters**. Using these parameters, the address range being watched can be adjusted. The register to be monitored must be defined at synthesis time.*

The parameters that must be set for this core are those listed in table 7-12. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. There are two registers to be accessed by this core: the lower limit of the allowed range and the upper limit of the allowed range. Thus a total of 2 addresses need to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0x8 is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF000007.

To restrict the address range to be monitored to a subrange, it can be limited using a hard upper limit and a hard lower limit. The number of the register to be watched too has to be defined at synthesis

signal name	I/O	recomm. width	description
REGNUMXDI	I	5	Register number
REGVALXDI	I	32	Register value
VALIDINSTRXSI	I	1	High if the currently executed instruction is valid
REGWEXEI	I	1	High if the current register is being written.
INTXSO	O	1	Interrupt
ACKXSI	I	1	Interrupt acknowledge

Table 7-13: Core Signals. *The core only uses internal I/O signals that are connected either to the μBlaze or to an interrupt controller. If used with and interrupt controller, the interrupt signal must be assigned an interrupt handler other than the default handler.*

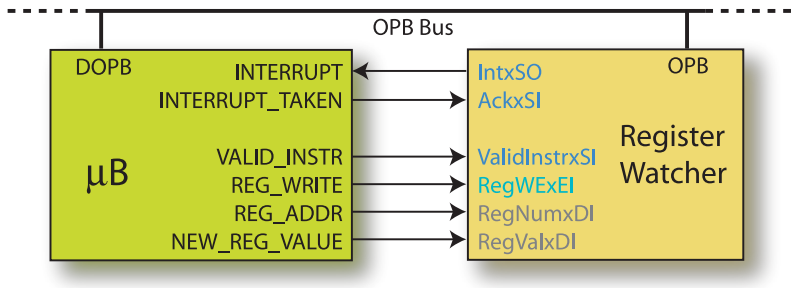


Figure 7-15: Wiring Example. *This figure is a suggestion on how to connect the register watcher to the μBlaze without the use of an interrupt controller.*

time.

7.8.3 I/O Signals

The signals used by this core are listed in table 7-13. The signal REGNUMXDI is assigned the number of the register that has recently been read from or written to by the μBlaze. It should be connected to the μBlaze's REG_ADDR output. REGVALXDI is the value currently being stored in register REGNUMXDI. This signal should be tapped at the μBlaze's NEW_REG_VALUE port. As VALIDINSTRXSI shows the validity of the instruction executed at the moment, these register values are meaningful only if this signal is high. VALIDINSTRXSI should be connected to the VALID_INSTR port of the μBlaze. If REGWEXEI is high, the register REGNUMXDI is being written. REGWEXEI should be connected to the REG_WRITE output of the μBlaze. For sake of clarity, these connections are depicted in figure 7-15.

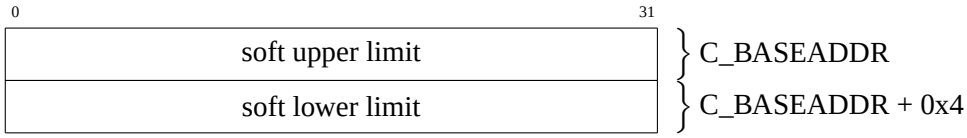


***Figure 7-16: Core Registers**. The upper limit and the lower limit of the memory range the monitored register is allowed to point at can be configured via OPB into these two registers.*

***Figure 7-17: Protected Memory Map**. As soon as the monitored register points to a location inside the disallowed region, the interrupt is asserted high. The two hard limits are configured at synthesis time, the soft limits can be moved at runtime.*

7.8.4 Core Operation

The core simply monitors the register selected with C_REG_NUM and raises an interrupt as soon as its value leaves the range between the soft lower limit and the soft upper limit. These limits are termed *soft* because they can be set using the software in contrast to C_HARD_LOWER_LIMIT and C_HARD_UPPER_LIMIT. The soft limits are stored at the address locations given in figure 7-16. The value of the register is only compared against the range of valid values when VALIDINSTRXSI and REGWEXEI are both 1.

To allow for more flexibility, the hard limits can be used to furher limit the range of disallowed values as depicted in figure 7-17, i.e. to monitor only part of the address range. If you don't want to use these hard limits, set C_HARD_LOWER_LIMIT to 0x00000000 and C_HARD_UPPER_LIMIT to 0xFFFFFFFF.

7.8.5 Driver

To assign an interrupt handler to the interrupt line, a driver supporting this feature is needed, e.g. `timerint`.

7.8.6 Software

The following code snippet provides macros to set the soft limits and to disable the register watcher core. XPAR_REGWATCH_BASEADDR is assumed to be set to the OPB base address of the register watcher core.

```
1 // macro to set the limits for the allowed address range
2 #define SET_REGWATCH_LIMITS(upper,lower) WRITE_UPPER_LIMIT(upper);\
3                                          WRITE_LOWER_LIMIT(lower)
4 // macro to disable the register watcher
5 #define DISABLE_REGWATCH() SET_REGWATCH_LIMITS(0xFFFFFFFF,0)
6 // low level macros used in the above macros
7 #define WRITE_UPPER_LIMIT(val) *((Xuint32*)(XPAR_REGWATCH_BASEADDR)) = val
8 #define WRITE_LOWER_LIMIT(val) *((Xuint32*)(XPAR_REGWATCH_BASEADDR + 0x4)) = val
```

7.8.7 Outlook

In a future version, this core may backup the values of all registers, which then could be read for debugging purposes.

7.9 OPB SRAM Controller

Core Name: `opb_xfcsram`

7.9.1 Introduction

This memory controller is a custom-tailored OPB core for the C-FPGA being part of the XFBOARD. This core completely replaces the v2.00.a solution, because it doubles access speed due to the fact that the IPIF (Intellectual Property Interface) provided by Xilinx has been replaced by custom logic. This speed increase is necessary because this controller will be also used to fetch instrucions where maximum efficiency is desired. There are no major changes due in your XPS project when upgrading to v3.00.a as the interface remained the same. The MIR register addresses that have been mandatory in v2.00.a are no longer used.

7.9.2 Parameters

You may tailor this core by adjusting parameters listed in table 7-14. The address range of the memory is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. As the capacity of the SRAM modules [2] attached to the C-FPGA sums up to 4 MB = 2^{22} = 0x3FFFF, a valid configuration would be a base address of 0xFFC00000 and a high address of 0xFFFFFFFF, or 0xF0000000 and 0xF03FFFFF.

7.9.3 I/O Signals

The I/O signals for the SRAM memory controller are listed in table 7-15. Please note the recommended ranges for these signals as these ranges exactly fit to the number of pins available in the user constraints file for the C-FPGA. The SRAM interface formed by the I/O signals is extended by an additional signal as the SRAM modules on the XFBOARD expect address bit 19 to be present both in its normal and its inverted state due to some hardware specialities[10].

7.9.4 Driver

No special driver is recommended, the driver `generic` may completely satisfy your needs.

7.9.5 Software

The SRAM core now supports word, half-word and byte access. An example C source that fills the SRAM with counter values and reads these values back from the memory is given below.

```
#include "xuartlite_l.h"
#include "xgpio_l.h"
#include "xbasic_types.h"
#include "xio.h"
#include "xparameters.h"

main() {
  Xuint32 i,j,w,e,c;
  Xuint32* ptr;


  xil_printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
  xil_printf("test of the opb sram controller\n");
  xil_printf("-------------------------------\n");
  xil_printf(">\n");

  while(1){
    while(XGpio_mReadReg(XPAR_SWITCH_BASEADDR,0)==0);

    xil_printf(">writing...\n");
    i=0;
    ptr=(Xuint32*)(XPAR_SRAM_BASEADDR); // set start address
    while(ptr <= XPAR_SRAM_HIGHADDR){ // the constants for the base
      // and the high address are defined
      // iff you use the xfcsram driver
      *ptr=(Xuint32)(i); // write value to ram
      ptr++;          // increment address
      i++;            // increment value
    }

    xil_printf(">reading...\n");
    i=0;         // value
    e=0;         // erroneous reads
    c=0;         // correct reads
    ptr=(Xuint32*)(XPAR_SRAM_BASEADDR);
    while(ptr <= XPAR_SRAM_HIGHADDR){
      w=*ptr;
      if (w!=i){ // error
     e++;
     xil_printf("error at %08X: \texpected:%08X \tactual:%08X\n",ptr,i,w);
      }
      else{      // no error
     c++;
      }
      ptr++;     // increment address
      i++;       // increment value
    }
    xil_printf("done with %d correct words and %d errors\n",c,e);
  }
}
```

description	parameter name	allowable values	default value	VHDL type
SRAM memory base address	C_SRAM_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
SRAM memory high address	C_SRAM_HIGHADDR	address range must be $\geq$ 0x3FFFFF and a power of 2	none	std_logic_vector

***Table 7-14: Core Parameters**. The only parameters that have to be adjusted for the SRAM controller are the limits of the memory range.*

signal name	I/O	recommended width	description
AxDO	O	20	memory address bus
A19xDBO	O	1	inverted MSB of the address bus
DxDIO	IO	32	bidirectional memory data bus
WxEBO	O	4	write enable, low active

***Table 7-15: Core Signals**. The inputs and outputs of the memory controller form a regular SRAM interface except for the special treatment of address bit 19 which also needs to be present in the inverted state.*

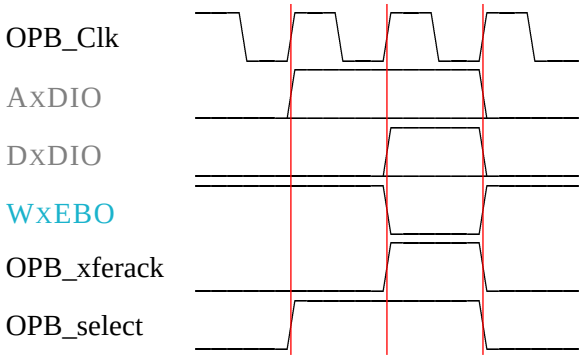


***Figure 7-18: Timing Waveform for Write Cycle**. These are the waveforms of the relevant SRAM controller signals for a write access. The duration of a write cycle is 2 clock periods.*

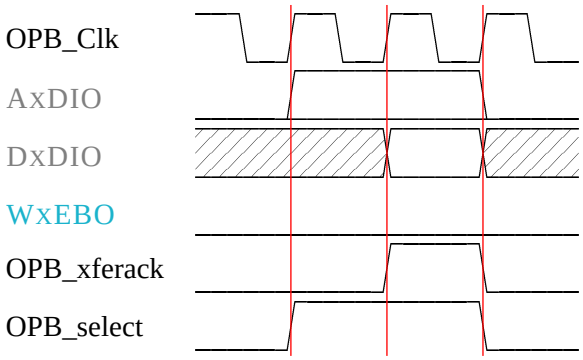


***Figure 7-19: Timing Waveform for Read Cycle**. These are the waveforms of the relevant SRAM controller signals for a read access. The duration of a read cycle is 2 clock periods.*

7.9.6 Timing for Memory I/O Signals

The total length of write and read operations is exactly two OPB bus cycles (see figures 7-18 and 7-19). From the moment the OPB bus assigns a valid address and the select signal, the controller needs one cycle to change into the write or read state plus one cycle to write or read the data from or to the memory and, at the same time, acknowledging the completion of the operation to the OPB, which finally deasserts the select signal.

As can been seen in the timing diagrams, the address is applied to the memory about two clock cycles (40 ns), which is sufficient due to the memory access time of 15 ns.

7.9.7 Outlook

Also, it might be possible to reduce access times to one clock cycle by directly feeding the OPB signals to the SRAM modules. However, first experiments with one clock access times failed, but still it might be possible with more in-depth explorations.

7.10 OPB Temperature Module

Core Name: `opb_xfctemp`

7.10.1 Introduction

This module is a ready-to use OPB core that allows reading from the temperature sensors [6] installed on the XFBOARD. It accesses these controllers via their SMBus interface.

7.10.2 Parameters

The only parameters available are the OBP address limits. Although the module only needs one address location, the address range is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0.

7.10.3 I/O Signals

The I/O signals for the SRAM memory controller are listed in table 7-16. Please note the recommended ranges for these signals as these ranges exactly fit to the number of pins available in the user constraints file for the C-FPGA!

7.10.4 Driver

No special driver is recommended, the driver `generic` may completely satisfy your needs.

7.10.5 Core Operation

First, the address of the temperature sensor attached to the C-FPGA is written to the SMBus to let the sensor know that it is the target of the following command. Then a command is written meaning that the temperature of the C-FPGA will be polled next. Again, the address of the temperature sensor must be written to the bus due to the change of dataflow direction. Now the temperature is read as an 8-bit signed integer. The same sequence is repeated for the R-FPGA. As soon as both temperatures are read, their bit sequences are concatenated and stored in a register that can be read via the OPB bus; only the bits 0 to 15 are used as shown in figure 7-20.

signal name	I/O	recommended width	description
SMBDxDIO	IO	1	SMBus bidirectional data line
SMBCLKxCO	O	1	SMBus clock

***Table 7-16: Core Signals**. The IO signal of the temperature module connects to the SMB bus attached to the temperature ADCs.*

7.10.6 Software

Accessing the temperature register is very simple as can be seen in the code snipped below:

```
1 Xuint32 temp;
2 Xint32 temp_c, temp_r;
3
4 temp =*((volatile Xuint32*)XPAR_TEMPERATURE_BASEADDR); // read temperature
5 temp_c = (Xint32)((temp>>24) & 0xFF); // bits 0 to 15
6 temp_r = (Xint32)((temp>>16) & 0xFF); // bits 16 to 23
```

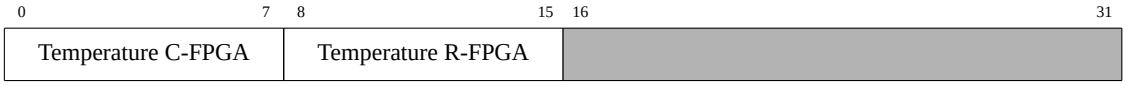


***Figure 7-20: Core Registers**. The temperature module uses one register which holds the temperatures of both FPGAs. Obviously, this register is read-only.*

7.11 OPB Timer

Core Name: `opb_xfctimer`

7.11.1 Introduction

This is a simple timer that raises an interrupt when a certain amount of time has elapsed. The amount of time can be programmed in the software by accessing a register in this core. Such timers are often used in systems combined with a process scheduler.

7.11.2 Parameters

description	parameter name	allowable values	default value	VHDL type
Timer base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
Timer high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector

***Table 7-17: Core Parameters**. The only parameters that need to be adjusted are the limits of the timer's memory range.*

The only parameters that must be set for this core are those listed in table 7-17. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. There is only one register used by this core, so a single address needs to be decoded. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0x4 is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF000003.

7.11.3 I/O Signals

The signals used by this core are listed in table 7-18. INTxSO is the level-sensitive, high active interrupt, ACKxSI is the signal used to acknowledge this interrupt. As depicted in figure 7-21, the interrupt should be connected to μBlaze's INTERRUPT port, and the interrupt acknowledge to its INTERRUPT_TAKEN port.

signal name	I/O	recommended width	description
INTxSO	O	1	Interrupt
ACKxSI	I	1	Interrupt acknowledge

***Table 7-18: Core Signals**. The register watcher's I/O signals.*

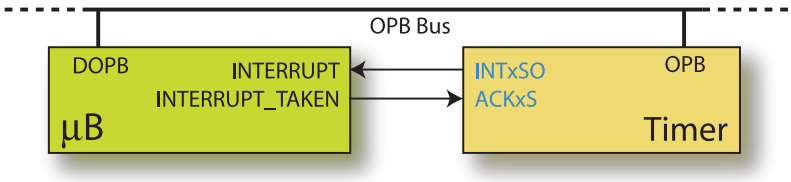


***Figure 7-21: Example Connection Scheme**. This figure shows how to connect the timer's interrupt signals to the µBlaze. Alternatively, an interrupt controller such as the one discussed in section 7.12 could be used instead of directly connecting the two modules.*

7.11.4 Core Operation

The internal counter is incremented up to a maximum value $c_{max} \cdot 2^{16}$, then the interrupt signal INTxSO is asserted high. The interrupt remains high until it is acknowledged using ACKxSI; now the counter restarts from zero. This process is depicted in figure 7-22. The value c_{max} can be set using the software. As c_{max} can be set to zero to force the timer to raise the interrupt, c_{max} is reset to its default value 1 as soon as the interrupt is acknowledged; this measure must be taken to prevent the core from being stuck in the interrupt state.

Assuming a system clock at 50 MHz, one unit c_{max} equals to a time interval of about 1.31 milliseconds. The maximum value for c_{max} is 0xFFFF, which is approximately 86 seconds.

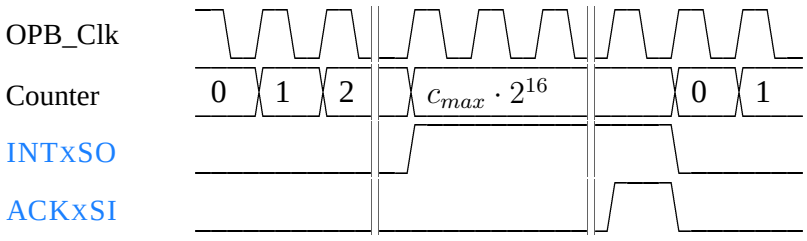


***Figure 7-22: Example Waveforms**. In this example, the timer reaches its maximum $c_{max} \cdot 2^{16}$ and asserts the interrupt. After a certain time, the interrupt is being handled and acknowledged.*

7.11.5 Driver

To assign an interrupt handler to the interrupt line, a driver supporting this feature is needed, e.g. `timerint`.

7.11.6 Software

The timer's maximum c_{max} can be set by writing to the base address of the timer, e.g.

```
*((Xuint32*)XPAR_TIMER_BASEADDR) = 0x3;
```

sets the time interval to $3 \cdot 1.31\text{ms} = 3.93\text{ms}$, assuming that XPAR_TIMER_BASEADDR is the core's base address.

7.12 OPB Interrupt Controller

Core Name: `opb_xfintc`

7.12.1 Introduction

This core is a simple, parametrized interrupt controller that, along with the appropriate bus interface, attaches to the OPB. It is able to handle up to 32 interrupts and merge them to one single interrupt signal forwarded to the μBlaze. Information about the interrupts that have occurred can be read from the core's status register.

7.12.2 Parameters

description	parameter name	allowable values	default value	VHDL type
Interrupt controller base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
Interrupt controller high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector
Number of interrupt inputs	C_NUM_INTERRUPTS	1 to 32	1	integer

***Table 7-19: Core Parameters**. Besides the limits of the memory range, the number of interrupts the controller is able to handle is parametrized.*

The parameters that must be set for this core are those listed in table 7-19. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. A total of 3 registers need to be accessible via OPB; thus a 3 addresses need to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0xC is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF00000B.

The interrupt controller can handle up to 32 interrupt inputs. The number of actually used inputs must be set using C_NUM_INTERRUPTS.

7.12.3 I/O Signals

The signals used by this core are listed in table 7-20. INTxSO should be connected to the INTERRUPT port of the μBlaze. INTRxSI and ACKxSO have to be connected to the cores generating the

signal name	I/O	recommended width	description
INTRxSI	I	C_NUM_INTERRUPTS	Interrupt inputs, level-sensitive, high active
INTxSO	O	1	Interrupt output, level-sensitive, high active
ACKxSO	O	C_NUM_INTERRUPTS	Acknowledge outputs

Table 7-20: Core Signals. *The interrupt controller's I/O signals.*

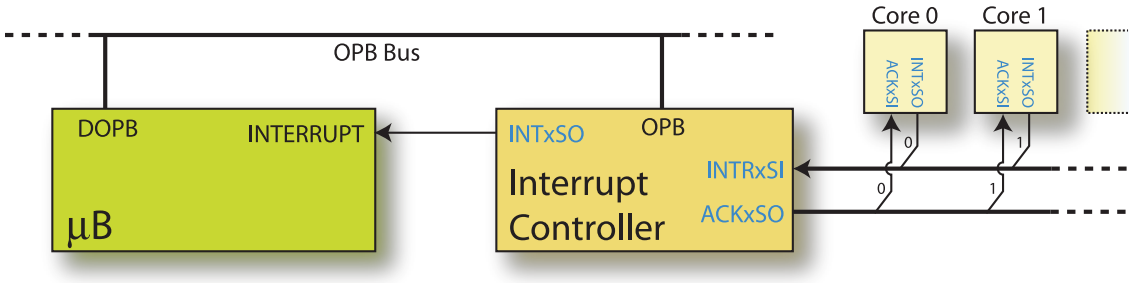


Figure 7-23: Example Connections. *This figure shows an example wiring of two cores able to generate interrupts with the interrupt controller. The interrupt controller merges all interrupt signals into a single interrupt, which is then forwarded to the µBlaze.*

interrupts. For sake of clarity, the wiring of the µBlaze and the interrupt controller including 2 cores generating interrupts is depicted in figure 7-23.

7.12.4 Core Operation

As soon as the interrupt controller detects an interrupt at its input, it sets the bit corresponding to this input in its internal interrupt register INTxDP. To be detected by the core, the interrupt input must be high for at least one clock cycle. Once a bit is set, it can be cleared only by setting the corresponding bit in the ACKxDP, which can be achieved by writing via OPB to this register. Whenever a bit in INTxDP is set and the corresponding enable bit in ENAxDP is set too, the core asserts its interrupt output INTxSO. So ENAxDP is used to mask the interrupts in INTxDP. The interrupt mask can be configured via OPB. The registers that can be read from and written to are depicted in figure 7-24.

To help understand the core operation, a detailed example shall be explained using figure 7-25. The example is divided into 5 phases, denoted as T0 to T4:

T0 Initially, no interrupt is active and no interrupt is masked. Then, an interrupt is detected on INTRxSI(0) which results in the corresponding bit being set in INTxDP. As a consequence, the

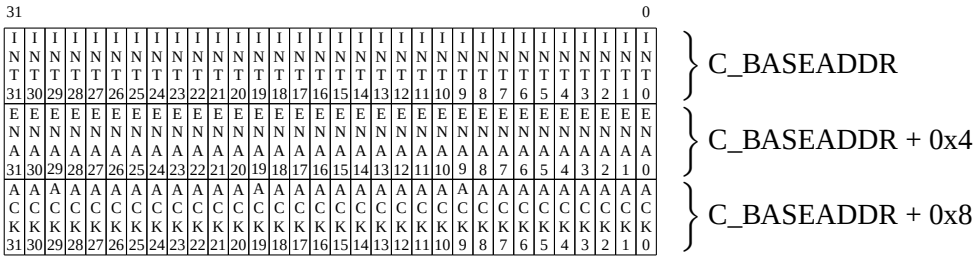


***Figure 7-24: Core Registers**. The core offers three registers that can be read or written via the OPB bus. The first contains the bits corresponding to the interrupts that have been triggered, the second contains the interrupt mask, and the third one is used to acknowledge the interrupts.*

core's interrupt output INTxSO gets asserted high. Let's assume that INTRxSI(0) comes from a core that expects an acknowledge, so this interrupt will stay high as long as it is not acknowledged.

T1 Later, the interrupt handler in the software becomes active and acesses the interrupt register via OPB to see which of the individual interrupts triggered the system interrupt. The core answers with 0x00000001 (signal Sln_Dbus) because INTRxSI(0) was triggered. As this interrupt still waits for an acknowledge, it remains active.

T2 Now the core connected to INTRxSI(1) generates an interrupt. This core does not expect the interrupt to be acknowledged and asserts the signal for only one clock cycle. The interrupt gets detected by the interrupt controller, and the corresponding bit is set in INTxDP. As a consequence, bits 0 and 1 are set, which is equal to 0x00000003.

T3 In this period, the interrupt handler acknowledges interrupt 0 by writing 0x00000001 to the acknowledge register ACKxDP. Bit 0 in INTxDP gets cleared, and the acknowledge signal is forwarded to the core that raised INTRxSI(0), which in turn deasserts the INTRxSI(0). Because the interrupt register still contains bits that are set, INTxSO remains high.

T4 In the last period of our example, the softare masks all interrupts by writing 0x00000000 to the enable register ENAxDP. This results in INTxSO being deasserted. The contents of INTxDP don't get altered as the interrupts are masked and not cleared.

7.12.5 Software

Using the interrupt controller from within the software is easy if the driver `xfintc` is linked to the code. For details on this driver refer to its documentation in section A. The most important elemtents

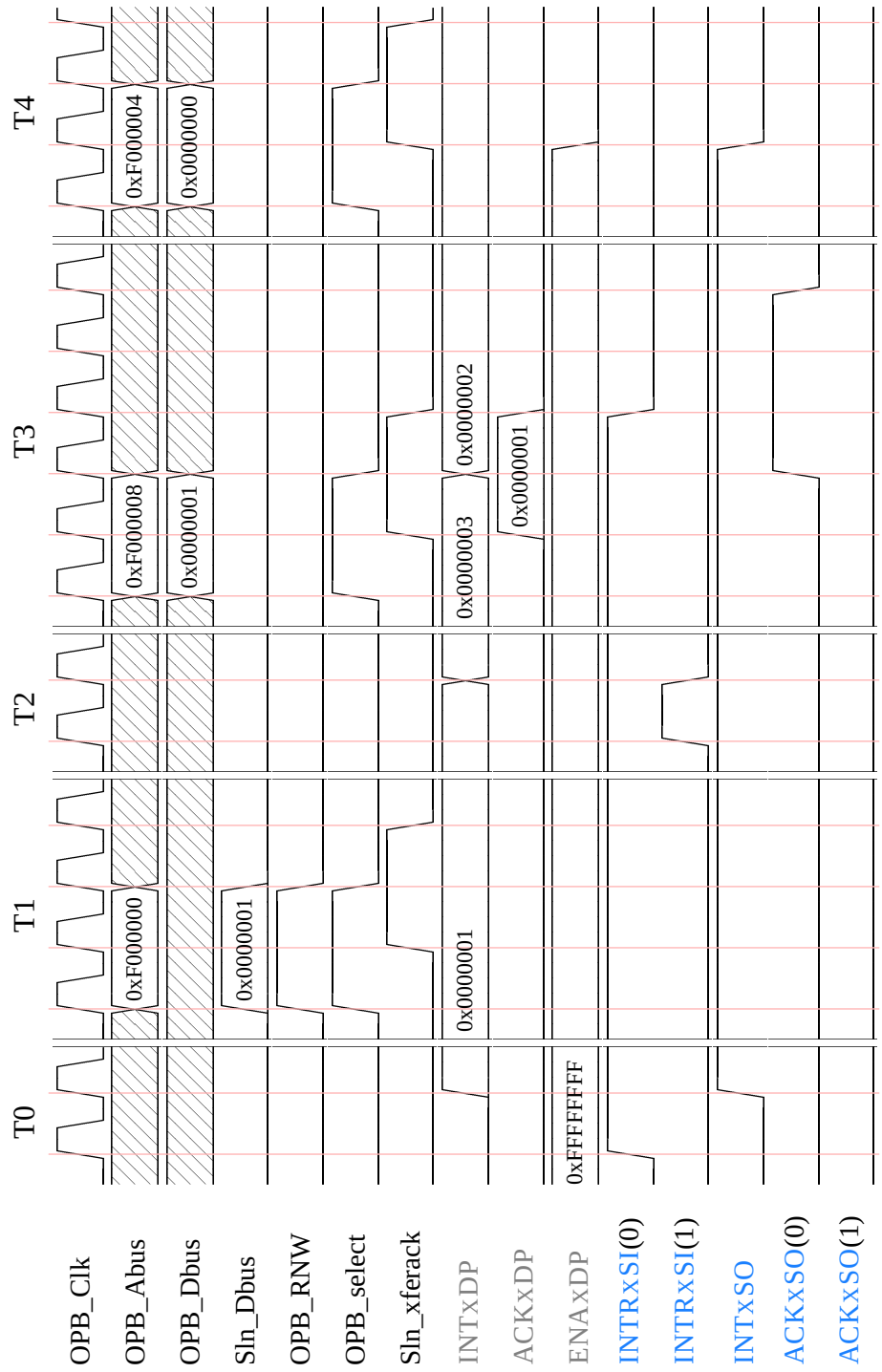


Figure 7-25: Example Waveforms. *These signal waveforms show an example interrupt scenario. A description on the various phases T0 to T4 is given in the text.*

of the driver are described here. The interrupt vector table and its element are defined as follows:

```
// declaration of the interrupt vector table
XFVectorTableEntry xfintc_interruptVectorTable[];

// entry in the interrupt vector table
typedef struct
{
  XInterruptHandler handler; // function pointer to the interrupt handler
  void *callBackRef;   // pointer to the base address of the interrupting core
} XFVectorTableEntry;
```

This interrupt vector table gets populated automatically with the functions defined as interrupt handler in the driver settings for the cores that are connected to the interrupt controller.

For your convenience, the driver also defines some useful macros to access the registers in the interrupt controller:

```
// register offsets to the interrupt controller's base address
#define XFINTC_INT_REG_OFFSET 0 // address offset of the interrupt register
#define XFINTC_ENA_REG_OFFSET 4 // address offset of the enable mask register
#define XFINTC_ACK_REG_OFFSET 8 // address offset of the acknowledge register

// macro to get bit pattern denoting the active interrupts
#define XFINTC_GET_INT_REG(BaseAddress) \
   XIntc_In32((BaseAddress) + XFINTC_INT_REG_OFFSET)

// macro to get bit pattern denoting the enabled interrupts
#define XFINTC_SET_ENA_REG(BaseAddress, EnableMask) \
   XIntc_Out32((BaseAddress) + XFINTC_ENA_REG_OFFSET, (EnableMask))

// macro to acknowledge interrupts using a bit pattern
#define XFINTC_ACK_INT(BaseAddress, AckMask) \
   XIntc_Out32((BaseAddress) + XFINTC_ACK_REG_OFFSET, (AckMask))
```

7.13 OPB Time Counter

Core Name: `opb_xftime`

7.13.1 Introduction

This core counts the number of periods since the last system reset.

7.13.2 Parameters

description	parameter name	allowable values	default value	VHDL type
Interrupt controller base address	C_BASEADDR	any valid address not overlapping with the memory space of other cores or memories	none	std_logic_vector
Interrupt controller high address	C_HIGHADDR	address range must be $\geq$ 0xF and a power of 2	none	std_logic_vector

***Table 7-21: Core Parameters**. The only parameters that can be adjusted for the time counter are the limits of the memory range.*

The parameters that must be set for this core are those listed in table 7-21. The address range of the core is restricted to be a power of 2. If the desired address range is represented by 2^n, then the n least significant bits of the base address must be 0. The only registers that need to be accessible via OPB are the upper and the lower word of the counter; thus 2 addresses need to be decoded by the core. Due to the fact that the μBlaze is a 32 bit processor, this means that a minimum memory range of 0x8 is needed. Therefore, a valid configuration would be a base address of 0xFF000000 and a high address 0xFF000007.

7.13.3 I/O Signals

This core does not use any signals additional to those needed for the OPB.

7.13.4 Core Operation

The number of clock periods since the last system reset is counted using a 43-bit register, therefore it is able to count $2^{43} = 8.7 \cdot 10^{12}$ clock periods before wrapping over. Assuming a system clock at 50 MHz, this yields a time span of about 48 hours and 52 minutes. The counter can be read by accessing the registers in figure 7-26 via OPB.

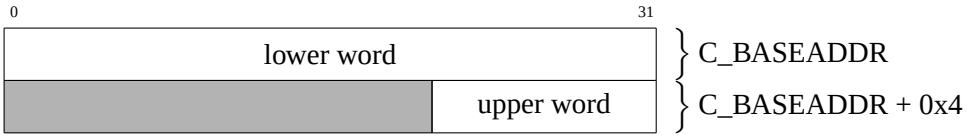


***Figure 7-26: Core Registers**. The core uses to registers containing the upper and the lower part of the number of clock cycles completed since power-up or reset. These registers are read-only.*

7.13.5 Software

Below is a code sample that prints the time elapsed since the last system reset. The time is printed in hours, minutes and seconds. This code sample has been written with the assumptions that the system clock is at 50 MHz and XPAR_TIME_BASEADDR is the OPB base address of the time counter.

```
1  void print_time(){
2    Xuint32 tHi,tLo;
3    Xuint32 sec,min,hr;
4     // get lower word
5    tLo = *((Xuint32*)(XPAR_TIME_BASEADDR))/50000000;
6     // get upper word
7     // (85899/1000) approximates (0x100000000/50000000) to prevent numerical problems
8    tHi = *((Xuint32*)(XPAR_TIME_BASEADDR+4))*(85899/1000);
9    sec = tLo+tHi;
10   min = sec/60;
11   hr = min/60;
12    // now print the time
13   printf("%3d:%02d:%02d",hr,min%60,sec%60);
14 }
```

7.14 OS Bridge, Part R-FPGA

7.14.1 Introduction

The OS Bridge is meant as a communication interface between C-FPGA and R-FPGA. Using a simple instruction set, data and configurations can be written to and read from hardware modules on the R-FPGA, e.g. FiFos could be monitored and dumped. The hardware component described in this section represents the R-FPGA part of the OS Bridge. Documentation on the C-FPGA part can be found in section 7.6. The OS bridge has been introduced in [19]. This part consists of an OS bridge bus master and a number of slaves. The instruction written to the OS bridge are interpreted by these slaves and forwarded to the corresponding system component in the R-FPGA. For every component to be controlled using the OS bridge, a new OS bridge slave has to be designed. An overview of the OS bridge is given in figure 7-27.

7.14.2 OS Bridge Bus Master

signal name	I/O	recommended width	description
OSBxDIO	IO	16	data / instruction bus
OSBTrstexTI	I	1	tristate control signal
OSBWxEI	I	1	write enable, active high
OSBRxEO	O	1	read enable, active high
OSBBusOpcodexDO	O	8	opcode
OSBBusDxDO	O	32	data (write)
OSBBusDxDI	I	16	data (read)
OSBBusSelxSO	O	1	bus select
OSBBusRNWxSO	O	1	read / not write
OSBBusRdyxSI	I	1	data ready

***Table 7-22: OS Bridge Master Signals**. The upper group of I/O signals is directly connected to the R-FPGA via the system's GPIO bus. The lower group connects to the OS bridge slaves, forming a down-sized version of the OPB.*

The OS bridge bus master reads the opcodes and data written from the C-FGPA and converts them to be put on the OS bridge bus which is a down-sized version of the OPB used in the μBlaze system. The same conversion is also performed in the opposite direction. The signals connecting to the C-FPGA and to the slaves are listed in table 7-22.

When a write operation is performed by the C-FPGA, the bus master detects which command type is being written by looking at the two most significant bits of the first word, and reads in the data according to this command type. The data are split into an 8-bit opcode and a 32-bit data word and then written to the according bus signals. The bus select line is pulled high to inform the slaves that

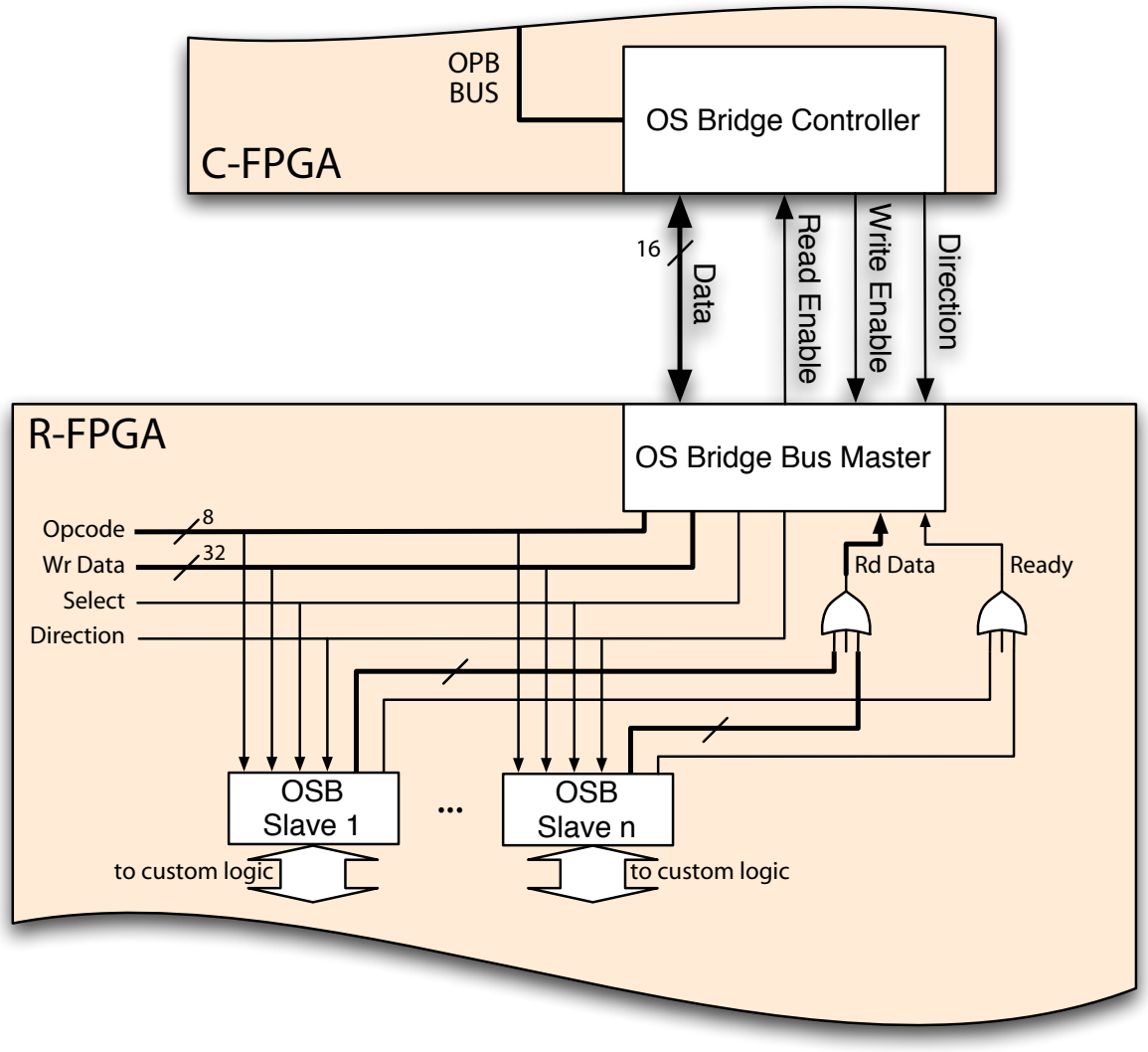


***Figure 7-27: OS Bridge**. The OS bridge as a complete system consists of an OPB OS bridge controller on the C-FPGA and an OS bridge bus master with a number of slaves attached on the R-FPGA. Commands are written by the controller and then forwarded to the slaves by the bus master. The slaves then interpret the opcodes an the data and communicate with the custom logic connected to them.*

signal name	I/O	recommended width	description
OSBBUSOPCODEXDI	I	8	opcode
OSBBUSDXDI	I	32	data (write)
OSBBUSDXDO	O	16	data (read)
OSBBUSSELXSI	I	1	bus select
OSBBUSRNWXSI	I	1	read / not write
OSBBUSRDYXSO	O	1	data ready

Table 7-23: OS Bridge Slave Signals. *These are the signals that connect to the OS bridge bus. The signals that connect to the hardware module to be communicated with are not listed as they are custom defined and varying from slave to slave.*

opcode and data are ready, pulling OSBBUSRNWXSO low which means that data are being written. The bus master does not expect the write process to be acknowledged. Instead, the bus select signal is active only one clock cycle. Opcode and data are left on the bus until new data arrive.

If a read operation is performed by the C-FPGA, the opcode gets fetched and written to the OS bridge bus. As it is a read operation, OSBBUSRNWXSO is high now. The select signal is asserted high. Now the bus master wait for a slave to reply to the operation by asserting OSBBUSRDYXSI. The ready signal has to be pulled high within 14 clock cycles starting from the rising edge of the select signal to prevent the master from timing out. If the ready signal is asserted in time, 16 bits of data are read from the read data bus and written to the C-FPGA via the GPIO bus, pulling OSBRXEO high to inform the OPB OS bridge controller that data are ready to be read.

7.14.3 OS Bridge Slaves

For every opcode being used in the system, an OS Bridge Slave must be present being able to perform the actions requested by this opcode. For read operations, there must be exactly one slave replying; write operations may be interpreted by more than one slaves. A slave is often able to interpret more than one opcode. OS bridge slaves form the interfaces between the OS bridge and the hardware module in the R-FPGA to be communicated with by the CPU.

The inputs of the slaves can be directly connected to the outputs of the OS bridge bus master. As a signal cannot be multiply driven, the outputs of the slave may not be connected directly to the inputs of the bus master, except for the case when only one slave is in the system. Instead, they are combined using a n-input OR gate whose output is connected to the input of the bus master. As a consequence, slaves not being active have to pull all their outputs to zero.

Two examples of OS bridge slaves shall be given:

example 1: a slave could translate the OS bridge bus protocol into a protocol to communicate with or read to and write from another element in the R-FPGA.

example 2: a slave could use the data written by the OS bridge to update internal registers whose outputs are used as configuration signals, e.g. to enable / disable hardware elements.

Whenever the bus select signal OSBBUSSELXSI is active, the slaves have to check whether they can handle the opcode present on the bus. When a slave is able to do so and the operation is a write request, it reads the data from the bus and communicates with the custom logic attached. If it is a read request, the slave gathers the data from the custom logic or from internal registers (depending on the opcode and the purpose of the slave) and writes them to the bus, pulling the ready signal high. As soon as the select signal is released by the master and thus the operation is terminated, the slave should pull all their outputs to zero.

8 Operating System Code

8.1 XF-Board Operating System Data Structure Documentation

8.1.1 BITS Struct Reference

8.1.1.1 Detailed Description

This struct describes a bitstream present in memory. It holds all information needed for the configuration.

Data Fields

- char Name [32]
  Name of the bitstream.
- Xuint32 Type
  Type of the bitstream: full / partial.
- Xuint32 SpanFrom
  First task slot occupied by the bitstream.
- Xuint32 SpanTo
  Last task slot occupied by the bitstream.
- Xuint32 MemLoc
  Memory Location of the bitstream.
- Xuint32 MemLength
  Length of the bitstream.

8.1.2 CommandEntry_t Struct Reference

8.1.2.1 Detailed Description

This structure describes a built-in command that can be executed the shell. Every command meant to be accessible from the shell has to be installed in the OS using an instance of this structure. The functions that can be called using a shell command commonly have a signature like `my_command(char **argv, Xuint32 argc)`.

Data Fields

- const char name [32]
  Name of the command. This string must be entered in the shell to execute the command.
- const FunctionPointer func
  Pointer to the function that has to be called when the command is entered.
- Xuint32 stackSize
  Minimal stack requirements for the above function.
- Xuint32 quanta
  Number of scheduler time quanta.
- Xuint32 suspendShell
  Define whether command suspends shell or not.

8.1.3 ContextDescriptor_t Struct Reference

8.1.3.1 Detailed Description

The task context descriptor is used for the context switches performed by the scheduler. As soon as a task is pre-empted, its register contents, the context, are stored in its task context descriptor to be loaded back into the registers the next time the task is activated. You should not change this struct without adjusting the file scheduler.s as the functions implemented there rely on the order and the presence of all members of the context descriptor

Data Fields

- Xuint32 sp
  R1, stack pointer.
- Xuint32 rosdaa
  R2, read-only small data area anchor.
- Xuint32 ret [2]
  R3, R4, return values.
- Xuint32 par [6]
  R5-R10, passing parameters / temporaries.
- Xuint32 tmp [2]

R11, R12, temporaries.

- Xuint32 rwsdaa
  R13, read-write small data area anchor.
- Xuint32 ri
  R14, return address for interrupt.
- Xuint32 rs
  R15, return address for sub-routine.
- Xuint32 rt
  R16, return address for trap.
- Xuint32 re
  R17, return address for exceptions.
- Xuint32 res
  R18, reserved for assembler.
- Xuint32 nv [13]
  R19-R31, non-volatiles.
- Xuint32 pc
  PC, program counter.
- Xuint32 rmsr
  Machine status register.

8.1.4 GraphicListItem_t Struct Reference

8.1.4.1 Detailed Description

An item in the`GraphicList` contains information on the graphics hook-up function that will be called by the `gfx_graphicManager()` and the state of this item. The state information is needed to allocate space on the screen for a graphics hook-up.

Data Fields

- GFXHookPtr gfxHook
  Pointer to a graphics hook-up function.
- Xuint8 blockState
  State of the item.
- Xuint8 ownerPID
  Owner PID of the item.

8.1.5 MemoryBlock_t Struct Reference

8.1.5.1 Detailed Description

Descriptors of memory blocks are used to build the memory allocation table `MemAllocTable`. A memory block descriptor contains information needed for the allocation algorithms, such as the type

of the memory block and its owning process. It also provides information to delimit blocks belonging together to form segments.

Data Fields

- Xuint8 ownerPID
  Owner of the memory block.
- Xuint8 memType
  Type of the memory block.
- XFbool lastBlock
  Flag: last block of a contiguous memory segment.

8.1.6 StackDescriptor_t Struct Reference

8.1.6.1 Detailed Description

This structure describing the dimensions of the stack memory of a process is used mainly for the allocation routine `mem_allocStack(Xuint32, StackDescriptor)`. When this function is called, the member `size` needs to be assigned a value, and then the two other fields `loAddr` and `hiAddr`, defining the borders of the memory segment used as stack memory by a process, are assigned the actual values.

Data Fields

- Xuint16 size
  Minimal size needed for the stack.
- Xuint32 loAddr
  Lowest address of actual stack memory.
- Xuint32 hiAddr
  Highest address of actual stack memory.

8.1.7 TaskDescriptor_t Struct Reference

8.1.7.1 Detailed Description

The task descriptor, which is used as a placeholder for a task / process in the `TaskList`, holds important information about a process such as a backup of its context and its arguments, to name only a few. A placeholder is either empty (`STATUS_UNUSED` or `STATUS_KILLED`) and can be assigned to a new task, or it is occupied (`STATUS_NEW`, `STATUS_RUNNING`, `STATUS_READY` or `STATUS_BLOCKED`). You should not change the order of the the first three members of this struct without adjusting the file scheduler.s as the functions implemented there strongly rely on the order and the presence of these members of the task descriptor

Data Fields

- FunctionPointer entryPoint
  Start point in the program code.
- ContextDescriptor context
  Structure for storing the context.
- StackDescriptor stack
  Descriptor of the task's stack.
- ProcessStatus status
  Status of the process or placeholder.
- Xuint32 quanta
  Number of Scheduler quanta.
- XFbool suspendShell
  Flag controlling shell suspension.
- char * name
  Name of the process.
- char * argv [CUI_MAXARGS]
  Array of pointers to the arguments.
- char argbuf [CUI_COMMANDLENGTH]
  Buffer containing a copy of the arguments.
- Xuint8 argc
  Number of arguments.

8.1.8 XContactInfo Struct Reference

8.1.8.1 Detailed Description

A structure of this type holds relevant information about the contact. This information is commonly needed to answer to a request.

Data Fields

- Xboolean isBroadcast
  If true, the transfer is a broadcast message.
- XPacketType packetType
  Type of the packet.
- XIPType ipType
  Protocol type.
- XARPType arpType
  ARP type; only applies for ARP packets.
- Xboolean isMACAddrValid
  The value in the field `MACAddr` *is valid.*
- Xboolean isIPAddrValid
  The IP address is valid.

- Xuint8 MACAddr [6]
  MAC address.
- Xuint8 IPHeader [20]
  Copy of the IP header.
- Xuint8 ICMPPacket [8]
  Copy of the ICMP packet.

8.1.9 XF_PFDL_t Struct Reference

8.1.9.1 Detailed Description

An item in the physical FiFo descriptor list contains information about the type and the dimensions of a FiFo physically present in the design.

Data Fields

- Xuint32 PFID
  ID of the physical FiFo.
- Xuint32 type
  Type of the FiFo.
- Xuint32 baseAddr
  Base address of the FiFo (in blocks).
- Xuint32 size
  Size of the FiFo (in blocks).
- Xuint32 rdPtr
  Read pointer (in bytes), relative to the base address converted to bytes.
- Xuint32 wrPtr
  Write pointer (in bytes), relative to the base address converted to bytes.

8.1.10 XF_VFDL_t Struct Reference

8.1.10.1 Detailed Description

An item in the virtual FiFo descriptor list contains information about the mapping between tasks and the read / write interfaces of the FiFos listed in the physical FiFo descriptor list.

Data Fields

- Xuint8 TID
  ID of the task connected to the FiFo.
- Xuint8 TRFID
  Task-relative FiFo ID.

- Xuint8 direction
  Access direction: read (0), write (1).
- Xuint8 PFID
  ID of the physically connected FiFo.

8.1.11 XFFAT Struct Reference

8.1.11.1 Detailed Description

This struct implements a table of bitstreams.

Data Fields

- Xuint32 NumOfBits
  Number of bitstreams in memory.
- BITS Bits [16]
  Array of bitstream descriptors.

8.1.12 XPacketData Struct Reference

8.1.12.1 Detailed Description

This structure holds the relevant information about the data inside a packet.

Data Fields

- Xuint16 intSourcePort
  Source port.
- Xuint32 * ptrData
  Pointer to a buffer containing the actual data.
- Xuint16 intDataLength
  Length of the data in the buffer (in bytes).
- Xuint16 volatile intDestinationPort
  Destination port.

8.1.13 XPacketData8 Struct Reference

8.1.13.1 Detailed Description

This structure holds the relevant information about the data inside a packet.

Data Fields

- Xuint16 intSourcePort
  Source port.
- Xuint8 * ptrData
  Pointer to a buffer containing the actual data.
- Xuint16 intDataLength
  Length of the data in the buffer (in bytes).
- Xuint16 volatile intDestinationPort
  Destination port.

8.1.14 XPacketInfo Struct Reference

8.1.14.1 Detailed Description

This structure holds the relevant information about the network contact belonging to a packet.

Data Fields

- Xuint8 bytMACAddress [6]
  Buffer for the source address.
- Xuint8 bytIPHeader [20]
  Buffer for the IP header.
- Xuint8 bytIPAddress [4]
  Buffer for the IP address of the source.

8.1.15 XPortListener Struct Reference

8.1.15.1 Detailed Description

This structure represents an element in the port listener list. Port listeners are used by user processes to get UDP data from a given port number.

Data Fields

- Xuint32 port
  Port being listened to.
- char * buffer
  Buffer for the data being received.
- Xuint32 length
  Length of the buffer.
- Xuint32 ownerPID

PID of the process owning the listener.

- volatile Xboolean done
  This flag is set as soon as a packet is received and copied to the buffer.

8.2 XF-Board Operating System File Documentation

8.2.1 boot.c File Reference

8.2.1.1 Detailed Description

In this file, the main routine to bring up the operating system is defined.

Author:
Samuel Nobs

Date:
2004-03-31
Revision
1.19

Functions

- void welcome ()
  Welcome Screen.
- int main ()
  OS Main Function.

Variables

- Xuint32 _erodata
  Points to the end of the read-only memory section. Needed for the Memory protection.

8.2.1.2 Function Documentation

int main ()

This function starts the operating system. First, the address of the end of the read-only memory section is written to the memory controller which uses this information to protect this memory section from unwanted accesses. Note: this information can be written only once.

Now, the following services are initialized:

- VGA display
- scheduler
- graphics manager
- SelectMAP configuration port
- network
- memory manager

Optionally, the SRAM memory can be filled up with all zeroes. This code is inserted only if the constant `MEM_INIT` is defined, i.e. the compiler is started with `-DMEM_INIT`.

Then, a welcome screen is displayed using `welcome()`.

Next, the following actions are performed:

- the graphics manager `gfx_graphicManager()`, the shell `shell()` and the ethernet daemon `inetd()` get started, i.e. added to the tasklist. Actually, they do not start being executed until the scheduler activates them.
- the graphics display gets populated with a message window `msg_messageWin()`, a display of the memory map `mem_display()`, an overview of the R-FPGA usage `selmap_display()`, information on the clock outputs `cm_clockDisplay()`, and a temperature monitor for both FPGAs `temperature()`.

Then, all interrupts get enabled and the scheduler is started. The function never returns as it loops forever after starting the scheduler.

void welcome ()

Prints a simple welcome screen.

8.2.2 clockman.c File Reference

8.2.2.1 Detailed Description

This file implements the function used with the core that generates the 4 additional clock signals for the R-FPGA.

Author:
Samuel Nobs

Date:
2004-03-31

Revision
1.2

Functions

- Xuint32 cm_clockDisplay (Xuint32 vpos)
  Display Clock Settings.
- Xuint32 cm_setClockFrontend (char **argv, Xuint32 argc)
  Set Clock Configuration (shell command).

8.2.2.2 Function Documentation

Xuint32 cm_clockDisplay (Xuint32* vpos*)

This graphic hook-up function displays the frequency and the on / off state of all 4 clock signals.

Parameters:

vpos the vertical position of the message window. If equal to GFX_GET_SIZE defined in graphix.h, this means that the caller wants to know the number of lines needed for this graphics element

Returns:

0 in normal mode, the number of lines needed if asked for

Xuint32 cm_setClockFrontend (char ** argv, Xuint32* argc*)

This command sets the frequencies and the on / off states of the 4 clocks output to the R-FPGA.

```
usage: setclk [-e <ena>] [-f <freq>] <clocknum>

       -e :        enable/disable
                   <ena>   : enable (1), disable (0)

       -f :        set frequency
       <freq>:     frequency number, 0-31

       <clocknum>: clock number, 0-3
```

Parameters:

argv the pointer to the argument list

argc the number of arguments

Returns:

0

8.2.3 clockman.h File Reference

8.2.3.1 Detailed Description

This is the header file for `clockman.c`

Author:
Samuel Nobs

Date:
2004-03-31
Revision
1.2

Defines

- #define CM_ENABLE_MASK 0x80000000
  The mask used to enable / disable the clock.
- #define CM_BASE_FREQ 125
  The base frequency used for calculation of the clock frequency.
- #define CM_SET_CLOCK(clock, val) *((Xuint32*)(XPAR_CLOCKMAN_BASEADDR+4*(clock))) = (val)
  Configure Clock Manager.
- #define CM_GET_CLOCK(clock) *((Xuint32*)(XPAR_CLOCKMAN_BASEADDR+4*(clock)))
  Get Clock Configuration.
- #define CM_ENABLE_CLOCK(clock) (CM_SET_CLOCK(clock,(CM_GET_CLOCK(clock) | CM_ENABLE_MASK)))
  Enable clock number `clock`.
- #define CM_DISABLE_CLOCK(clock) (CM_SET_CLOCK(clock,(CM_GET_CLOCK(clock) & ~CM_ENABLE_MASK)))
  Disable clock number `clock`.

Functions

- Xuint32 cm_clockDisplay (Xuint32 vpos)
  Display Clock Settings.
- Xuint32 cm_setClockFrontend (char **argv, Xuint32 argc)
  Set Clock Configuration (shell command).

8.2.3.2 Define Documentation

#define CM_GET_CLOCK(clock) *((Xuint32*)(XPAR_CLOCKMAN_BASEADDR+4*(clock)))

Reads the configuration value of clock number `clock`. The MSB of `val` is the enable / disable flag.

Parameters:

clock the clock number

#define CM_SET_CLOCK(clock, val) ∗((Xuint32∗)(XPAR_CLOCKMAN_-BASEADDR+4∗(clock))) = (val)

Configures the clock number `clock` with the value `val`. The MSB of `val` is the enable / disable flag.

Parameters:

clock the clock number

val the configuration value

8.2.4 cui.c File Reference

8.2.4.1 Detailed Description

This file implements a basic command user interface (shell) including tab completion, a command history, and function keys that can be assigned commonly used commands. Other special keys are handled as well. All user interaction is done using this shell, except for functions that catch keyboard input themselves.

Author:

Samuel Nobs

Date:

2004-02-01

Revision

1.16

Functions

- CommandEntry ExtOsCommandList[CUI_NUMBER_OF_EXTERN_COMMANDS] __-attribute__ ((section(".extos_commands")))=
  List of External OS Commands.
- void cui_addtohistory (char ∗bufPtr)
  Add Command to History.
- void cui_gethistory (Xuint32 index, char ∗bufPtr)
  Get History Entry.
- char ∗ cui_complete (char ∗commBuffer, char ∗bufPtr)
  Complete Command.
- void cui_getcommand (char ∗commBuffer, char ∗∗argv, Xuint32 ∗argc)
  Get Command from Input.

- void shell ()
  Command Shell.
- Xuint32 fkey (char **argv, Xuint32 argc)
  Assign Function Keys (shell command).
- Xuint32 cui_escapeKeyInterruptHandler ()
  Escape Key Interrupt Handler.
- char cui_readKeyboard ()
  Read from Keyboard.
- Xuint32 cui_resume (char **argv, Xuint32 argc)
  Resume Task (shell command).

Variables

- char cui_history [CUI_HISTORYLENGTH][CUI_COMMANDLENGTH]
  Circular history buffer containing recently entered commands.
- Xuint32 cui_histWritePos = 0
  Actual position in the history buffer, i.e. where the command entered next has to be stored.
- char * prompt = CUI_PROMPT
  The prompt used by the `shell()`.
- Xuint32 PIDsuspendingShell
  The PID of the process suspending the shell.
- const CommandEntry CommandList [CUI_NUMBER_OF_COMMANDS]
  List of Built-In Commands.
- CommandEntry * UserCommandList
  Pointer to the list containing the user commands.
- CommandEntry * ExtOsCommandList
  Pointer to the list containing the external OS commands.
- Xuint32 * NUserCommands
  Pointer to the number of user commands.
- char * MagicKeyword
  Pointer to the magic keyword used to determine wether code is loaded or not.
- char FKeyMapping [12][CUI_FKEY_COMM_LEN]
  Function key command mapping list.

8.2.4.2 Function Documentation

CommandEntry ExtOsCommandList [CUI_NUMBER_OF_EXTERN_COMMANDS] __attribute__ ((section(".extos_commands")))

This is a list containing all external commands recognized by the `shell()`. However, this list must be included in the external code by giving the option -DEXTERNAL_OS_CODE to the compiler. It is not included in the core OS code present in the block RAM. Refer to the linker script for information on where this list is put in the code, i.e. the code section .extos_commands .

void cui_addtohistory (char * bufPtr)

Adds a command to the circular history buffer `cui_history` at the position pointed at by `cui_histWritePos`, which then gets incremented.

Parameters:

bufPtr pointer to the string representing the command to add to the history

char* cui_complete (char * commBuffer, char * bufPtr)

Complete a command entered in the `shell()`. This function searches the `CommandList` and the `UserCommandList` and `ExtOsCommandList` (only if available) for commands starting with the string pointed at by the arguments. If no match is found, a pointer to the end of the string entered so far is returned. If various matches are found, the string is completed as long as it is common to the beginning of all matching commands, then a pointer to the end of the completed string is returned. If only one match is found, the string is completed to the end of the matching command, and then a pointer to the end of the completed string is returned.

Parameters:

commBuffer pointer to the beginning of the string to be completed

bufPtr pointer to the end of the string to be completed

Returns:

pointer to the end of the completed string

Xuint32 cui_escapeKeyInterruptHandler ()

This is the interrupt handler that gets called whenever the escape key gets hit. It first looks whether a task is currently suspending the shell. If no such task can be found, the interrupt handler exits. Otherwise, a dialog is shown letting the user decide whether to kill the task, suspend the task (and thus giving control back to the shell), or continuing the task.

Returns:

0

void cui_getcommand (char * commBuffer, char ** argv, Xuint32 * argc)

This function gets character input from the keyboard and completes this input using `cui_complete(char*, char*)` whenever the tab key is hit. It also listens to the up / down arrow keys to get a command from the history using `cui_gethistory(Xuint32, char*)`. It also treats moving forward and backward using the left / right arrow keys and deleting using backspace. However, all key inputs work in insert mode so far, e.g. when backspace is hit in the middle of a string, the characters to the right of the cursor are not shifted to the left.

Additionally, the page up and page down keys tell the graphic manager to go to the next / previous page by calling `gfx_nextPage()` or `gfx_prevPage()`.

As soon as the user hits the return key, the string entered is added to the history using `cui_addtohistory(char*)`, then it is split into one command and a list of arguments. Arguments with one or more blanks have to be enclosed by single quotes.

Parameters:

commBuffer buffer where the characters are written to

argv pointer to the list of arguments

argc pointer to the number of arguments

void cui_gethistory (Xuint32 **index,** *char* * **bufPtr***)*

Gets a command from the history buffer `cui_history` at the position specified. As a second argument, a buffer is needed. A copy of the command found in the history is returned in this buffer. Make sure to allocate `CUI_COMMANDLENGTH` bytes for this string.

Parameters:

index position in the history buffer

bufPtr pointer to a buffer that will be filled with the command

char cui_readKeyboard ()

Whenever a user task suspending the shell is reading the keyboard, this function should be used instead of `kbd_getc()` because it filters the page up / page down keys and forwards them to the graphics manager. This means that the graphics display can still be paged through although the shell (which normally handles the paging) is suspended.

Returns:

the character read from the keyboard

Xuint32 cui_resume (char ** **argv,** *Xuint32* **argc***)*

Whenever a task has been suspended, it can be again put into service using this command. The command takes one conditionally optional argument, a process ID; this ID is not necessary as long as there is only one command currently being suspended. As soon as the result of calling this command without argument is ambiguous, the PID to be resumed is requested.

Parameters:

argv pointer to the argument list

argc argument count

Returns:

0

Xuint32 fkey (char ** argv, Xuint32 argc)

This command assigns a comment including arguments to a function key on the keyboard. The first argument in the list is expected to be a number between 1 and 12, denoting the number of the function key the string given as a second argument will be assigned to. If no argument is given, the current function key mapping is printed to the screen.

```
usage: fkey <fkeynum> <commandstr>

       put a shortcut to <commandstr> on F<keynum>
```

If the command to be mapped exceeds `CUI_FKEY_COMM_LEN` it is truncated. If a command includes white space, it must be enclosed by single quotes, i.e. `'command using space'`. These quotes are not accounted for when calculating the string's length.

Parameters:

argv pointer to the argument list

argc argument count

Returns:

0

void shell ()

This is a loop representing the command user interface (shell) of the os. It prints the `prompt` string to the screen, then it waits to get a command and a list of arguments from `cui_getcommand(char*, char**, Xuint32*)`. If the command is empty, the next `prompt` is printed and a new command is waited for. If the command is not empty, it is searched for in the `CommandList`. If the command is found in the list, it is added to the task list, including the argument list, the argument count, the name of the command, the number of scheduling time quanta and the minimal stack size required. This step is done using the scheduler function `sch_addToTaskList(FunctionPointer, char**, Xuint32, char*, Xuint32, Xuint32, Xboolean)`. A message about the absence of a built-in command matching the command entered is printed if the command could not be found in the `CommandList`.

8.2.4.3 Variable Documentation

const CommandEntry CommandList[CUI_NUMBER_OF_COMMANDS]

Initial value:

```
{
 {"bitslist",  (FunctionPointer)&bitslist,        2048,10,XTRUE},
 {"cls",       (FunctionPointer)&vga_cls,         500,10,XFALSE},
```

```
  {"config",    (FunctionPointer)&config,          500,SCH_INFINITE_QUANTA,XFALSE},
  {"context",   (FunctionPointer)&context,         500,20,XTRUE},
  {"fkey",      (FunctionPointer)&fkey,            500,10,XTRUE},



  {"inetd",     (FunctionPointer)&inetd,           1024,SCH_DEFAULT_QUANTA,XFALSE},

  {"ipconfig",  (FunctionPointer)&net_ipconfig,    500,10,XTRUE},
  {"kill",      (FunctionPointer)&kill,            500,10,XTRUE},
  {"ps",        (FunctionPointer)&ps,              800,20,XTRUE},
  {"resume",    (FunctionPointer)&cui_resume,      800,20,XFALSE},
  {"setquanta", (FunctionPointer)&setquanta,       500,10,XTRUE},
}
```

This is a list containing all built-in commands recognized by the `shell()`

char FKeyMapping[12][CUI_FKEY_COMM_LEN]

Initial value:

```
{

  "config 0\0",
  "config 1\0",
  "bitslist\0",
  "ps\0",
  "cls\0",
  "\0",
  "\0",
  "\0",
  "\0",
  "\0",
  "\0",
  "fkey\0",
}
```

8.2.5 cui.h File Reference

8.2.5.1 Detailed Description

This is the header file for cui.c .

Author:
samuel nobs

Date:
2004-02-01

Revision
1.14

Data Structures

- struct CommandEntry_t
  Command Entry.

Defines

- #define CUI_COMMANDLENGTH 64
  Maximum length of a command.
- #define CUI_ARGLENGTH 17
  Maximum length of an argument string.
- #define CUI_MAXARGS 6
  Maximum number of arguments.
- #define CUI_HISTORYLENGTH 10
  Length of the command history.
- #define CUI_PROMPT ">XF-%d>"
  String used for the prompt.
- #define CUI_PROMPT_ASK "XF-?>"
  String used for the prompt for ambiguous results on tab completion.
- #define CUI_FKEY_COMM_LEN 21
  Maximum length of the commands being mapped to the F-Keys (including string terminator).
- #define CUI_NUMBER_OF_COMMANDS 11
  Number of built-in commands.
- #define CUI_NUMBER_OF_EXTERN_COMMANDS 3
  Number of built-in commands in extern memory.
- #define CUI_PROGRAM_LOADED_KEYWORD "badger "
  Magic keyword to determine whether program has been loaded or not.
- #define CUI_PROGRAM_LOADED_KEYWORD_LEN 8
  Lengt of the magic keyword, including terminator.
- #define XF_COMMAND_LIST
  User Command List.

Typedefs

- typedef CommandEntry_t CommandEntry
  Command Entry.

Functions

- void cui_getcommand (char ∗commBuffer, char ∗∗argv, Xuint32 ∗argc)

Get Command from Input.

- Xuint32 fkey (char **argv, Xuint32 argc)
  Assign Function Keys (shell command).
- void shell ()
  Command Shell.
- Xuint32 cui_resume (char **argv, Xuint32 argc)
  Resume Task (shell command).
- char cui_readKeyboard ()
  Read from Keyboard.

Variables

- Xuint32 PIDsuspendingShell
  The PID of the process suspending the shell.

8.2.5.2 Define Documentation

#define XF_COMMAND_LIST

Value:

```
char    theKeyword[CUI_PROGRAM_LOADED_KEYWORD_LEN] \
__attribute__((section(".magic_keyword")))= CUI_PROGRAM_LOADED_KEYWORD;\
Xuint32 UserCommandCount                           \
__attribute__((section(".command_count"))) = XF_NUMBER_OF_COMMANDS;\
CommandEntry MyCommandList[XF_NUMBER_OF_COMMANDS]  \
__attribute__((section(".user_commands")))
```

This macro can be used in programs that are intended to reside in external memory. It adds user-defined commands and the external OS commands to the `shell()`. To enable these user commands, this macro builds a table containing all the information necessary to call the user-defined functions. This table is then inserted at the correct location by the linker, the `.user_commands` section. This where it is expected to be by the OS. The macro also defines the magic keyword that is written to the memory. This keyword is used by the `shell()` to determine whether external code is available or not.

8.2.5.3 Typedef Documentation

typedef struct CommandEntry_t CommandEntry

This structure describes a built-in command that can be executed the shell. Every command meant to be accessible from the shell has to be installed in the OS using an instance of this structure. The functions that can be called using a shell command commonly have a signature like `my_command(char **argv, Xuint32 argc)`.

8.2.6 graphix.c File Reference

8.2.6.1 Detailed Description

This file contains graphics routines such as the graphic manager and graphical elements. All functions are meant to be used for the graphics column of the display.

Author:
Samuel Nobs

Date:
2004-01-31
Revision
1.12

Functions

- void gfx_init ()
  Initialize Graphic Manager.
- void gfx_clearline ()
  Clear Line.
- void gfx_hline (Xuint8 vpos)
  Draw Horizontal Line.
- Xuint32 gfx_graphicManager ()
  Graphic Manager.
- void gfx_nextPage ()
  Go to Next Page.
- void gfx_prevPage ()
  Go to Previous Page.
- XFError gfx_addHook (GFXHookPtr hook)
  Add Graphics Display Hook-Up.
- XFError gfx_removeHooksForPID (Xuint32 pid)
  Remove hooks for PID.
- void gfx_fifofill (const char *label, Xuint32 val, Xuint32 lim, Xuint32 lowthres, Xuint32 highthres, Xuint32 size)
  FiFo fill-level monitor.
- void gfx_verticalBargraph (Xuint32 val, Xuint32 max, Xuint32 hpos, Xuint32 vpos, Xuint32 size)
  Vertical Bargraph.
- void gfx_historyBargraph (Xuint32 val, Xuint32 *hist, Xuint32 histlen, Xuint32 max, Xuint32 hpos, Xuint32 vpos, Xuint32 height)
  Vertical History Bargraph.
- void gfx_time ()
  Draw Up-Time.

Variables

- Xuint8 gfx_displayedPage
  The page currently displayed by the graphic manager.
- Xuint8 gfx_lastDisplayedPage
  The last page displayed by the graphic manager.
- GraphicListItem GraphicList [GFX_NUMBER_OF_PAGES][GFX_NUMBER_OF_LINES]
  The list containing the graphics hook-up functions.
- const char gfx_fifoBar [9] = {0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7,0xf8}
  The character set for horizontal bars.
- const char gfx_vBar [9] = {177,19,20,21,22,23,24,25,219}
  The character set for vertical bars.

8.2.6.2 Function Documentation

XFError gfx_addHook (GFXHookPtr* hook*)

Install a graphics display hook-up in the `GraphicList` which is called by `gfx_graphicManager()` to draw accordingly to the screen. To allocate enough room for the hook-up installed, it is asked for it's desired size by calling `hook(GFX_GET_SIZE)`. Then, a free slot in the `GraphicList` is searched using a simple first-fit algorithm.

Parameters:

hook graphics display hook-up function to add

Returns:

an XFError code:

- ERR_NONE if hook-up was successfully attached
- ERR_NO_FREE_GFXSLOT if no free slot was found

void gfx_clearline ()

Clears the current line. The text insertion point must reside at the beginning of the line when this function is called.

void gfx_fifofill (const char* ∗ label, *Xuint32* val, *Xuint32* lim, *Xuint32* lowthres, *Xuint32* highthres, *Xuint32* size)

This is a function displaying the fill-level of a FIFO queue in a manner compareable to a progress bar, including a text representing the fill-level as a percentage. In contrast to the progress bar, this graphics element is open on the right side.

Parameters:

label a label for this graphics element

val the value representing the fill-level of the FIFO queue

lim the maxium value being displayed

lowthres if val is below this threshold, the percentage text is highlighted

highthres if val is above this threshold, the percentage text is highlighted

size the size of the bar (in characters)

Xuint32 gfx_graphicManager ()

This is the graphic manager used by the operating system. Before it is started, it has to be initialized using `gfx_init()`. First, it disables all interrupts as interrupting the graphic manager might mess up the look of the display. Next, it draws the page numbers, highlighting the number of the page currently displayed. Then, it loops through the `GraphicList` and calls the hook-ups for the actual page found there, passing the vertical position as an argument. Then it recedes from the CPU and waits for being given CPU time again to start redrawing.

Returns:

0

void gfx_historyBargraph (Xuint32 **val,** *Xuint32* ∗ **hist,** *Xuint32* **histlen,** *Xuint32* **max,** *Xuint32* **hpos,** *Xuint32* **vpos,** *Xuint32* **height)**

Generates a vertical bargraph including information about the past. The buffer containing the history must be given as an argument, i.e. if the last 20 values should be displayed, a buffer of length 20 should be allocated. The function then shifts left the contents of this buffer, inserting the newest value to the right of the buffer. So the most recent value is displayed on the right side of the display; the right side represents the array element with the highest index. The user does not need to care about the content of the history buffer as it gets managed by the function. To get reasonable results, *val* should be in the range of `0` to *max*.

Parameters:

val the actual value

hist the history buffer of size `histlen`

histlen the size of `hist`, i.e. the width of the graph (in characters)

max the upper limit for `val`

hpos the horizontal position of the graph (in characters)

vpos the vertical position of the bottom of the graph (in lines)

height the height of the graph (in lines)

void gfx_hline (Xuint8* vpos*)

This function draws a horizontal line across the graphics column to separate the display vertically.

Parameters:

vpos vertical position in lines, starting with 0

void gfx_init ()

Before using the graphic manager `gfx_graphicManager()`, this function must be called to initialize the variables `gfx_displayedPage` and `GraphicList` .

void gfx_nextPage ()

Tells the graphic manager to display the next page on redraw. This is done by incrementing `gfx_displayedPage`.

void gfx_prevPage ()

Tells the graphic manager to display the previous page on redraw. This is done by decrementing `gfx_displayedPage`.

XFError gfx_removeHooksForPID (Xuint32* pid*)

Removes all graphics hook-up function associated with a certain PID. This function is used for cleaning up when a task terminates or is killed.

Parameters:

pid PID whose hooks should be removed

Returns:

0

void gfx_time ()

This function draws the up-time to the screen in the format `HHH:MM:SS`

void gfx_verticalBargraph (Xuint32* val*, Xuint32* max*, Xuint32* hpos*, Xuint32* vpos*, Xuint32* size*)

This is a function used to display a value as a vertical bar. To get a reasonable result, *val* should be in the range of `0` to *max*.

Parameters:

val the value to be displayed

max the upper limit for this value

hpos the horizontal position of the graph (in characters)

vpos the vertical position of the bottom of the graph (in lines)

size the height of the graph (in lines)

8.2.7 graphix.h File Reference

8.2.7.1 Detailed Description

This is the header file for graphix.c.

Author:
Samuel Nobs

Date:
2004-01-31
Revision
1.8

Data Structures

- struct GraphicListItem_t
  Item in the `GraphicList`.

State Constants

States of a `GraphicListItem`

- #define GFX_BSTATE_FREE 0
  Item is free and can be allocated.
- #define GFX_BSTATE_USED 1
  Item is used.
- #define GFX_BSTATE_LAST 2
  Item is the last item of a group of used items.

Defines

- #define GFX_NUMBER_OF_PAGES 3
  Number of pages being managed.
- #define GFX_NUMBER_OF_LINES (VGA_NUMBER_OF_LINES - 3)
  Number of lines per page.

- #define GFX_BASELINE (VGA_NUMBER_OF_LINES - 2)
  Start of the baseline comments (pages, up-time...).
- #define GFX_TIME_COLUMN 53
  Column at which the time gets displayed.
- #define GFX_TIME_LEN 9
  Length of the time string displayed.
- #define GFX_GET_SIZE 255
  Command used to request size from a graphic hook-up.

Typedefs

- typedef Xuint8 GFXHook
  Type of a graphic hook-up function.
- typedef GFXHook(∗ GFXHookPtr)(Xuint8)
  Pointer on a graphic hook-up function taking one argument.
- typedef GraphicListItem_t GraphicListItem
  Item in the `GraphicList`.

Functions

- void gfx_init ()
  Initialize Graphic Manager.
- void gfx_nextPage ()
  Go to Next Page.
- void gfx_prevPage ()
  Go to Previous Page.
- void gfx_time ()
  Draw Up-Time.
- void gfx_clearline ()
  Clear Line.
- XFError gfx_addHook (GFXHookPtr hook)
  Add Graphics Display Hook-Up.
- XFError gfx_removeHooksForPID (Xuint32 pid)
  Remove hooks for PID.
- Xuint32 gfx_graphicManager ()
  Graphic Manager.
- void gfx_fifofill (const char ∗label, Xuint32 val, Xuint32 lim, Xuint32 lowthres, Xuint32 highthres, Xuint32 size)
  FiFo fill-level monitor.
- void gfx_verticalBargraph (Xuint32 val, Xuint32 max, Xuint32 hpos, Xuint32 vpos, Xuint32 size)
  Vertical Bargraph.
- void gfx_historyBargraph (Xuint32 val, Xuint32 ∗hist, Xuint32 histlen, Xuint32 max, Xuint32 hpos, Xuint32 vpos, Xuint32 height)
  Vertical History Bargraph.

Variables

- GraphicListItem GraphicList [GFX_NUMBER_OF_PAGES][GFX_NUMBER_OF_LINES]
  The list containing the graphics hook-up functions.
- Xuint8 gfx_displayedPage
  The page currently displayed by the graphic manager.

8.2.7.2 Typedef Documentation

typedef struct GraphicListItem_t GraphicListItem

An item in the`GraphicList` contains information on the graphics hook-up function that will be called by the `gfx_graphicManager()` and the state of this item. The state information is needed to allocate space on the screen for a graphics hook-up.

8.2.8 kbd_layout_en.c File Reference

8.2.8.1 Detailed Description

This file contains the key mapping for an english keyboard layout.

Author:
Samuel Nobs

Date:
2004-02-03
Revision
1.3

Variables

- char kbd_keymap [2][128]
  Character map.

8.2.8.2 Variable Documentation

char kbd_keymap[2][128]

Initial value:

```
{{
```

```
    0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,'\t', ''',  0 ,
    0 ,  0 ,  0 ,  0 ,  0 , 'q', '1',  0 ,  0 ,  0 , 'z', 's', 'a', 'w', '2',  0 ,
    0 , 'c', 'x', 'd', 'e', '4', '3',  0 ,  0 ,  , 'v', 'f', 't', 'r', '5',  0 ,
    0 , 'n', 'b', 'h', 'g', 'y', '6',  0 ,  0 ,  0 , 'm', 'j', 'u', '7', '8',  0 ,
    0 , ',', 'k', 'i', 'o', '0', '9',  0 ,  0 , '.', '/', 'l', ';', 'p', '-',  0 ,
    0 ,  0 ,'\'',  0 , '[', '=',  0 ,  0 ,  0 ,  0 ,'\n', ']',  0 ,'\\',  0 ,  0 ,
    0 ,  0 ,  0 ,  0 ,  0 ,  0 ,'\b',  0 ,  0 , '1',  0 , '4', '7',  0 ,  0 ,  0 ,
  '0', '.', '2', '5', '6', '8',  0 ,  0 ,  0 ,  0 , '3',  0 ,  0 , '9',  0 ,  0
},
{

    0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,  0 ,'\t', '~',  0 ,
    0 ,  0 ,  0 ,  0 ,  0 , 'Q', '!',  0 ,  0 ,  0 , 'Z', 'S', 'A', 'W', '@',  0 ,
    0 , 'C', 'X', 'D', 'E', '$', '#',  0 ,  0 ,  , 'V', 'F', 'T', 'R', '%',  0 ,
    0 , 'N', 'B', 'H', 'G', 'Y', '^',  0 ,  0 ,  0 , 'M', 'J', 'U', '&', '*',  0 ,
    0 , '<', 'K', 'I', 'O', ')', '(',  0 ,  0 , '>', '?', 'L', ':', 'P', '_',  0 ,
    0 ,  0 ,'\"',  0 , '{', '+',  0 ,  0 ,  0 ,  0 ,'\n', '}',  0 , '|',  0 ,  0 ,
    0 ,  0 ,  0 ,  0 ,  0 ,  0 ,'\b',  0 ,  0 , '1',  0 , '4', '7',  0 ,  0 ,  0 ,
  '0', '.', '2', '5', '6', '8',  0 ,  0 ,  0 ,  0 , '3',  0 ,  0 , '9',  0 ,  0
}}
```

This matrix maps scan codes to characters. `kbd_keymap`[0] contains the characters for keys being pressed without the shift key while `kbd_keymap`[1] stands for the keys being pressed when the shift key is held down.

8.2.9 kbd_layout_en.h File Reference

8.2.9.1 Detailed Description

Author:
Samuel Nobs

Date:
2004-02-03
Revision
1.6

Modifier and Navigation Keys Scan Codes

- #define KBD_LSHIFT 0x12
  Left shift key.
- #define KBD_RSHIFT 0x59
  Right shift key.
- #define KBD_UARROW 0x75
  Up arrow key.
- #define KBD_DARROW 0x72

Down arrow key.

- #define KBD_LARROW 0x6B
  Left arrow key.
- #define KBD_RARROW 0x74
  Right arrow key.
- #define KBD_TAB 0x0D
  Tabulator key.
- #define KBD_PAGEUP 0x7D
  Page up key.
- #define KBD_PAGEDN 0x7A
  Page down key.

Function Keys Scan Codes

- #define KBD_F1 0x05
  F1 Key.
- #define KBD_F2 0x06
  F2 Key.
- #define KBD_F3 0x04
  F3 Key.
- #define KBD_F4 0x0C
  F4 Key.
- #define KBD_F5 0x03
  F5 Key.
- #define KBD_F6 0x0B
  F6 Key.
- #define KBD_F7 0x83
  F7 Key.
- #define KBD_F8 0x0A
  F8 Key.
- #define KBD_F9 0x01
  F9 Key.
- #define KBD_F10 0x09
  F10 Key.
- #define KBD_F11 0x78
  F11 Key.
- #define KBD_F12 0x07
  F12 Key.

Special Scan Codes

- #define KBD_BREAK 0xF0
  Break scan code, sent when a key is released.
- #define KBD_EXTEND_CODE 0xE0

Extension scan code, prepended to some keys' scan codes.

- #define KBD_PRTSC 0x7C
  Print screen key scan code.
- #define KBD_ESCAPE 0x76
  Escape key scan code.
- #define KBD_PLUS 0x79
  Plus key (num pad) scan code.
- #define KBD_MINUS 0x7B
  Minus key (num pad) scan code.

Variables

- char kbd_keymap [2][128]
  Character map.

8.2.9.2 Variable Documentation

char kbd_keymap[2][128]

This matrix maps scan codes to characters. `kbd_keymap`[0] contains the characters for keys being pressed without the shift key while `kbd_keymap`[1] stands for the keys being pressed when the shift key is held down.

8.2.10 keyboard.c File Reference

8.2.10.1 Detailed Description

This file provides functions that represent the software part of the keyboard driver.

Author:
samuel nobs

Date:
2004-01-31
Revision
1.10

Functions

- char kbd_getc ()
  Get Character.
- Xuint32 kbd_scan ()
  Fetch Scan Code.

8.2.10.2 Function Documentation

char kbd_getc ()

Gets a character from the keyboard. Regularly printable characters available on a standard keyboard are returned using their corresponding ASCII value. Values not used for regular keyboard characters are returned to inform the caller about special keys being hit on the keyboard, i.e. arrow keys or function keys. Basically, this function is a state machine fetching the scan codes !=0 from the keyboard and treating special cases (shift key, break codes etc.).

Returns:
a number representing a keyboard input:

- numbers $<$ 200 are standard ASCII values
- numbers $>=$ 200 refer to special keys (see keyboard.h)

Xuint32 kbd_scan ()

This low level function fetches a scan code from the keyboard using a non-blocking read. If no scan code is present, 0 is returned.

Returns:
a scan code or 0

8.2.11 keyboard.h File Reference

8.2.11.1 Detailed Description

This file is the header for keyboard.c

Author:
samuel nobs

Date:
2004-02-01
Revision
1.9

Arrow and Navigation Keys

These constants are returned by `kbd_getc()` when an arrow / navigation key was hit on the keyboard.

- #define KBD_UPKEY 200

Up arrow key.

- #define KBD_DOWNKEY 201
  Down arrow key.
- #define KBD_LEFTKEY 202
  Left arrow key.
- #define KBD_RIGHTKEY 203
  Right arrow key.
- #define KBD_TABKEY 204
  Tabulator key.
- #define KBD_PGDNKEY 205
  Page down key.
- #define KBD_PGUPKEY 206
  Page up key.

Function Keys

These constants are returned by `kbd_getc()` when a function key was hit on the keyboard

- #define KBD_F1KEY 210
  F1 key.
- #define KBD_F2KEY 211
  F2 key.
- #define KBD_F3KEY 212
  F3 key.
- #define KBD_F4KEY 213
  F4 key.
- #define KBD_F5KEY 214
  F5 key.
- #define KBD_F6KEY 215
  F6 key.
- #define KBD_F7KEY 216
  F7 key.
- #define KBD_F8KEY 217
  F8 key.
- #define KBD_F9KEY 218
  F9 key.
- #define KBD_F10KEY 219
  F10 key.
- #define KBD_F11KEY 220
  F11 key.
- #define KBD_F12KEY 221
  F12 key.

Defines

- #define KBD_PRTSCKEY 222
  Print screen key.
- #define KBD_ESCAPEKEY 223
  Escape key.
- #define KBD_PLUSKEY 224
  Plus key (num pad).
- #define KBD_MINUSKEY 225
  Minus key (num pad).

Functions

- char kbd_getc ()
  Get Character.
- Xuint32 kbd_scan ()
  Fetch Scan Code.

8.2.12 lock.h File Reference

8.2.12.1 Detailed Description

This file provides low-level macros used for working with the hardware test-and-set structure

Author:
Samuel Nobs

Date:
2004-04-05

Revision:
1.1

Defines

- #define LOCK()
  Lock a task.
- #define LOCK_OWNER() ∗((volatile Xuint32∗)(XPAR_HWLOCK_BASEADDR))
  Get PID of the locking task.
- #define UNLOCK()
  Unlock a task.

Variables

- Xuint32 runningTask
  PID of the task currently running.

8.2.12.2 Define Documentation

#define LOCK()

Value:

```
sch_block();\
while(*((volatile Xuint8 *)(XPAR_HWLOCK_BASEADDR + runningTask)))
```

#define UNLOCK()

Value:

```
sch_unblock();\
*((volatile Xuint32*)(XPAR_HWLOCK_BASEADDR))=0
```

8.2.13 memory.c File Reference

8.2.13.1 Detailed Description

This file implements a simple memory manager for allocating memory for the stack and for larger amounts of data.

Author:
Samuel Nobs

Date:
2004-02-09
Revision
1.8

Functions

- void mem_init ()
  Initialize memory manager.
- void mem_allocStack (Xuint32 pid, StackDescriptor *stack)
  Allocate stack memory.
- Xuint32 mem_dealloc (Xuint32 pid)

Deallocate memory.

- void * malloc (Xuint32 size)
  Allocate Memory.
- void free (void *ptr)
  Free memory.
- Xuint32 mem_display (Xuint8 vpos)
  Display memory allocation map.
- void errorLMBInterrupt ()
  LMB Error Interrupt Handler.
- void errorStackInterrupt ()
  Stack Error Interrupt Handler.

Variables

- MemoryBlock MemAllocTable [MEM_NUM_BLOCKS]
  The table keeping track of the memory blocks.
- Xuint32 _erodata
  Points to the end of the read-only memory section. Needed for the Memory protection.

8.2.13.2 Function Documentation

void errorLMBInterrupt ()

This function gets called whenever the corresponding interrupt has been triggered by an LMB memory access error, i.e. someone tried to write to the read-only sections. This interrupt calls this handler, which kills the task and sends a debugging message to the host computer's debug port. The debugging message consists of the program counter.

Todo

A future version of this interrupt handler may send more detailed debugging information.

void errorStackInterrupt ()

The core watching the stack pointer generates an interrupt whenever it points to memory outside the allowed range. This interrupt calls this handler, which kills the task and sends a debugging message to the host computer's debug port. The debugging message consists of the program counter.

Todo

A future version of this interrupt handler may send more detailed debugging information.

void free (void * ptr)

Memory that has been allocated using `malloc(Xuint32)` can be returned to the pool of free memory blocks by calling this function. Starting from the location pointed to by the argument, all blocks

up to the one marked as being the last block of the segment are declared as free (`MEMTYPE_FREE`) if they are owned by the calling process.

Parameters:

ptr pointer to the memory section being freed

void∗ malloc (Xuint32 size*)*

This function tries to find a memory segment being an integer multiple of `MEM_BLOCKSIZE`, large enough to fulfill the requirements stated by the argument *size*. For every memory block allocated, the `MemAllocTable` is updated accordingly: the block is marked as being malloced (`MEMTYPE_MALLOC`), then it is assigned the ID of the currently running task, i.e. the caller of this function, and the block at the highest address of the segment is marked as being the last block of the segment, which simplifies freeing using `free(void∗)`.

As this function needs to be fast and efficient, the first-fit algorithm is used which yields the best results with respect to the tradeoff between low fragmentation and low complexity.

Parameters:

size size of the memory segment in bytes

Returns:

a pointer to the newly allocated memory segment on success, 0 on failure

void mem_allocStack (Xuint32 pid*, StackDescriptor ∗* stack*)*

Whenever a new task is going to be launched, memory for its stack variables is allocated using this function. The desired amount of memory is rounded up to the next integer multiple of `MEM_BLOCKSIZE`. If a contiguous memory segment is found, the `loAddr` and `hiAddr` fields of *stack* are adjusted to point to the start and to the end of the segment, respectively. If no segment was found, these fields are set to 0. For every memory block allocated for the stack, the allocation table is updated: the block is marked as being stack memory (`MEMTYPE_STACK`), and it is assigned the PID of the owning process.

As this function needs to be fast and efficient, the first-fit algorithm is used which yields the best results with respect to the tradeoff between low fragmentation and low complexity.

Parameters:

pid ID of the process whose stack will reside in the memory segment asked for

stack `StackDescriptor` containing the desired memory size

Xuint32 mem_dealloc (Xuint32 pid*)*

Deallocate all memory blocks owned by a certain process. This function is useful for cleanup when a task terminates or is killed. All blocks belonging to the process are marked as being free and therefore are available for other processes.

Parameters:

pid ID of the process whose memory will be deallocated

Returns:

0

Xuint32 mem_display (Xuint8 vpos*)*

This is a graphics hook-up function displaying the memory allocation map including a legend. This function can be added to the pool of graphics functions handled by the `gfx_graphicManager()`.

Parameters:

vpos Vertical position of the message window. If equal to `GFX_GET_SIZE` defined in `graphix.h`, this means that someone wants to know the number of lines needed for this graphics element

Returns:

0 in normal mode, the needed size if asked for

void mem_init ()

Before using the memory manager, its allocation table `MemAllocTable` must be initialized using this function. All available blocks are declared as free, and the owner's PID is set to 0.

8.2.14 memory.h File Reference

8.2.14.1 Detailed Description

This is the header file for memory.c

Author:

Samuel Nobs

Date:

2004-02-09

Revision

1.6

Data Structures

- struct MemoryBlock_t
  Descriptor of a Memory Block.
- struct StackDescriptor_t
  Stack Descriptor.

Types of Memory Blocks

- #define MEMTYPE_FREE 0
  Block is free.
- #define MEMTYPE_STACK 1
  Block is used for stack.
- #define MEMTYPE_MALLOC 2
  Block has been allocated using `malloc(Xuint32)`.

Defines

- #define MEM_BASEADDR (XPAR_SRAM_BASEADDR)
  Base address of the memory being managed.
- #define MEM_SIZE 0x00020000
  Size of the memory being managed.
- #define MEM_EXT_CODE (MEM_BASEADDR + MEM_SIZE)
  Base address of the memory that is used for external program code.
- #define MEM_BLOCKSIZE 0x00000100
  Size of the blocks the memory is organized in.
- #define MEM_NUM_BLOCKS (MEM_SIZE/MEM_BLOCKSIZE)
  Number of blocks the memory is split up into.
- #define MEM_DISPLAY_BLOCKSPERLINE 43
  Number of blocks displayed per line in the memory allocation map.

Typedefs

- typedef MemoryBlock_t MemoryBlock
  Descriptor of a Memory Block.
- typedef StackDescriptor_t StackDescriptor
  Stack Descriptor.

Enumerations

- enum MemoryBlockGFXState { blockFree = 176, blockMalloc = 178, blockStack = 219 }
  ASCII codes for the characters used for displaying the allocation map.

Functions

- Xuint32 mem_dealloc (Xuint32 pid)
  Deallocate memory.
- Xuint32 mem_display (Xuint8 vpos)
  Display memory allocation map.
- void free (void *ptr)

Free memory.

- void mem_allocStack (Xuint32 pid, StackDescriptor *stack)
  Allocate stack memory.
- void mem_init ()
  Initialize memory manager.
- void * malloc (Xuint32 size)
  Allocate Memory.

Variables

- MemoryBlock MemAllocTable [MEM_NUM_BLOCKS]
  The table keeping track of the memory blocks.

8.2.14.2 Typedef Documentation

typedef struct MemoryBlock_t MemoryBlock

Descriptors of memory blocks are used to build the memory allocation table `MemAllocTable`. A memory block descriptor contains information needed for the allocation algorithms, such as the type of the memory block and its owning process. It also provides information to delimit blocks belonging together to form segments.

typedef struct StackDescriptor_t StackDescriptor

This structure describing the dimensions of the stack memory of a process is used mainly for the allocation routine `mem_allocStack(Xuint32, StackDescriptor)`. When this function is called, the member `size` needs to be assigned a value, and then the two other fields `loAddr` and `hiAddr`, defining the borders of the memory segment used as stack memory by a process, are assigned the actual values.

8.2.14.3 Enumeration Type Documentation

enum MemoryBlockGFXState

Enumeration values:

blockFree Block is free.

blockMalloc Block is malloced.

blockStack Block is used as stack.

8.2.15 messagewin.c File Reference

8.2.15.1 Detailed Description

This file provides the functions for displaying messages in the graphics column using the memory manager.

Author:
Samuel Nobs

Date:
2004-01-30
Revision
1.1

Functions

- Xuint8 msg_messageWin (Xuint8 vpos)
  Draw Message Window.
- Xuint32 msg_printMsg (char *msgStr)
  Print Message.

Variables

- char msg_MessageBuffer [MSG_NUM_LINES][VGA_CHARS_PER_LINE-1]
  Text buffer containing the messages.
- Xuint32 msg_numOfLines = 0
  Number of lines that have been written to the message window so far.

8.2.15.2 Function Documentation

Xuint8 msg_messageWin (Xuint8 vpos*)*

This is the hook-up for drawing the message window. It gets called by the graphics manager. It can be used to display status messages in the graphics column that will not interfere with the `shell()` in the text column.

Parameters:
vpos Vertical position of the message window. If equal to GFX_GET_SIZE defined in graphix.h, this means that the caller wants to know the number of lines needed for this graphics element

Returns:
0 in paint mode, the needed size if being asked for

Xuint32 msg_printMsg (char * msgStr)

Print a message to the text buffer `msg_MessageBuffer` to be displayed in the message window. The current up-time is prepended to the message.

Todo

The current version of this function does not support formatted string. This feature may be enabled in a future version.

Parameters:

msgStr the string to be printed

8.2.16 messagewin.h File Reference

8.2.16.1 Detailed Description

This is the Header file for messagewin.c

Author:

Samuel Nobs

Date:

2004-01-30

Revision

1.1

Defines

- #define MSG_NUM_LINES 15
  Number of lines displayed in the message window.

Functions

- Xuint8 msg_messageWin (Xuint8 vpos)
  Draw Message Window.
- Xuint32 msg_printMsg (char *msgStr)
  Print Message.

8.2.17 mmu.c File Reference

8.2.17.1 Detailed Description

This file implements a configuration and debug interface to the memory management unit on the R-FPGA.

Author:
Samuel Nobs

Date:
2004-04-01
Revision
1.4

Functions

- void mmu_readVFDL (XF_VFDL *vfdlPtr, Xuint32 length, Xboolean sequential)
  Read VFDL.
- void mmu_writeVFDL (XF_VFDL *vfdlPtr, Xuint32 length, Xboolean sequential)
  Write VFDL.
- void mmu_readPFDL (XF_PFDL *pfdlPtr, Xuint32 length, Xboolean sequential)
  Read PFDL.
- void mmu_writePFDL (XF_PFDL *pfdlPtr, Xuint32 length, Xboolean sequential)
  Write PFDL.
- Xuint32 mmu_writeToFifo (XF_PFDL *pfdlPtr, Xuint16 value, Xboolean first)
  Write to FiFo.
- Xuint32 mmu_readFromFifo (XF_PFDL *pfdlPtr, Xboolean first)
  Read from FiFo.
- Xuint32 mmu_fifoFillLevel (XF_PFDL *pfdlPtr)
  Get FiFo Fill-Level.
- void mmu_dumpFifo (XF_PFDL *pfdlPtr, Xuint32 start, Xuint32 end)
  Dump FiFo Contents.

8.2.17.2 Function Documentation

void mmu_dumpFifo (XF_PFDL * pfdlPtr, Xuint32 start, Xuint32 end)

This function dumps the values between index *start* and *end* without affecting the FiFo's read / write pointers.

Parameters:

pfdlPtr the pointer to a `XF_PFDL` element describing the FiFo to get the fill-level from

start the index where to start the dump

end the index where to stop the dump

Xuint32 mmu_fifoFillLevel (XF_PFDL * pfdlPtr)

The fill-level of a FiFo defined by the description *pfdlPtr* is pointing at can be read using this function.

Parameters:

pfdlPtr the pointer to a `XF_PFDL` element describing the FiFo to get the fill-level from

Returns:

the fill-level of the FiFo

*Xuint32 mmu_readFromFifo (XF_PFDL * pfdlPtr, Xboolean first)*

This function can be used to read a value from the FiFo defined by the description *pfdlPtr* is pointing at. If the *first* flag is set to true, the address in the MMU to read from is set. As the OS bridge core implements support for array read mode, *first* can be set to false for write accesses following immediately. This allows faster reading the FiFo. Be sure to guarantee that no other access to the MMU changed the address in the meantime due to the system's being scheduled!

Todo

Currently, this function changes the read pointer of the FiFo in the MMU. Its functionality could be extended by supporting a read mode leaving the read pointer untouched. This task can be accomplished by explicitly reading the memory at the location the FiFo resides at, but this approach is not too comfortable.

Parameters:

pfdlPtr the pointer to a `XF_PFDL` element describing the FiFo to read from

first the flag to control writing of the address

Returns:

the value read from the fifo

*void mmu_readPFDL (XF_PFDL * pfdlPtr, Xuint32 length, Xboolean sequential)*

This function reads information from the physical FiFo description list in the MMU. There are two methods to read data from the PFDL, both of which assume a list of type `XF_PFDL` of length *length* being created in advance:

- the `PFID` field of the first element is assigned a value and the `sequential` argument is set to `MMU_SEQUENTIAL`. The function then fills in the information starting at the PFID defined in the first element of the list. This is the fastest way to get consecutive entries from the PFDL as the OS bridge core supports a special array read mode for successive addresses.

- the `PFID` fields of all elements of the list are assigned a value; the values do not need to be in a specific order. When the function is called with the *sequential* flag set to `MMU_PUNCTUAL` or `!MMU_SEQUENTIAL`, the information for the PFIDs is fetched from the VFDL. This approach is less effective as the address for each PFID must be sent to the MMU because the PFIDs are not consecutive.

Parameters:

vfdlPtr the pointer to a list of `XF_PFDL` entries

length the number of entries in this list

sequential the flag to choose between sequential mode and punctual mode

void mmu_readVFDL (XF_VFDL* ∗ vfdlPtr, *Xuint32* length, *Xboolean* sequential)

This function reads information from the virtual FiFo description list in the MMU. There are two methods to read data from the VFDL, both of which assume a list of type `XF_VFDL` of length *length* being created in advance:

- the `TID` and `TRFID` fields of the first element are assigned a value and the `sequential` argument is set to `MMU_SEQUENTIAL`. The function then fills in the information starting at the TID / TRFID pair defined in the first element of the list. This is the fastest way to get consecutive entries from the VFDL as the OS bridge core supports a special array read mode for successive addresses.

- the `TID` and `TRFID` fields of all elements of the list are assigned a value; the values do not need to be in a specific order. When the function is called with the *sequential* flag set to `MMU_PUNCTUAL` or `!MMU_SEQUENTIAL`, the information for the TID / TRFID pairs are fetched from the VFDL. This approach is less effective as the address for each pair must be sent to the MMU because the pairs are not consecutive.

Parameters:

vfdlPtr the pointer to a list of `XF_VFDL` entries

length the number of entries in this list

sequential the flag to choose between sequential mode and punctual mode

void mmu_writePFDL (XF_PFDL* ∗ pfdlPtr, *Xuint32* length, *Xboolean* sequential)

This function writes information to the physical FiFo description list in the MMU. There are two methods to write data to the PFDL, both of which assume a list of type `XF_PFDL` of length *length* being created and filled in the `type`, `baseAddr`, `size`, `rdPtr` and `wrPtr` in advance:

- the `PFID` field of the first element is assigned a value and the `sequential` argument is set to `MMU_SEQUENTIAL`. The function then feeds the information to the MMU, starting at the PFID defined in the first element of the list. This is the fastest way to copy consecutive entries to the PFDL as the OS bridge core supports a special array write mode for successive addresses.

- the `PFID` fields of all elements of the list are assigned a value; the values do not need to be in a specific order. When the function is called with the *sequential* flag set to `MMU_PUNCTUAL` or `!MMU_SEQUENTIAL`, the information for the PFIDs is written to the VFDL. This approach is less effective as the address for each PFID must be sent to the MMU because the PFIDs are not consecutive.

Parameters:

vfdlPtr the pointer to a list of XF_PFDL entries

length the number of entries in this list

sequential the flag to choose between sequential mode and punctual mode

Xuint32 mmu_writeToFifo (XF_PFDL ∗ **pfdlPtr,** *Xuint16* **value,** *Xboolean* **first)**

This function can be used to write a value in the FiFo defined by the description *pfdlPtr* is pointing at. If the *first* flag is set to true, the address in the MMU to write to is set. As the OS bridge core implements support for array write mode, *first* can be set to false for write accesses following immediately. This allows faster filling the FiFo. Be sure to guarantee that no other access to the MMU changed the address in the meantime due to the system's being scheduled!

Parameters:

pfdlPtr the pointer to a XF_PFDL element describing the FiFo to write to

value the value to be written to the FiFo

first the flag to control writing of the address

Returns:

0

void mmu_writeVFDL (XF_VFDL ∗ **vfdlPtr,** *Xuint32* **length,** *Xboolean* **sequential)**

This function reads information from the virtual FiFo description list in the MMU. There are two methods to write data to the VFDL, both of which assume a list of type XF_VFDL of length *length* being created and filled in the direction and PFID fields in advance:

- the TID and TRFID fields of the first element are assigned a value and the sequential argument is set to MMU_SEQUENTIAL. The function then feeds the information to the MMU, starting at the TID / TRFID pair defined in the first element of the list. This is the fastest way to write consecutive entries to the VFDL as the OS bridge core supports a special array write mode for successive addresses.

- the TID and TRFID fields of all elements of the list are assigned a value; the values do not need to be in a specific order. When the function is called with the *sequential* flag set to MMU_PUNCTUAL or !MMU_SEQUENTIAL, the information for the TID / TRFID pairs are written to the VFDL. This approach is less effective as the address for each pair must be sent to the MMU because the pairs are not consecutive.

Parameters:

vfdlPtr the pointer to a list of XF_VFDL entries

length the number of entries in this list

sequential the flag to choose between sequential mode and punctual mode

8.2.18 mmu.h File Reference

8.2.18.1 Detailed Description

This is the header file for `mmu.c`.

Author:
Samuel Nobs

Date:
2004-04-01
Revision
1.3

Data Structures

- struct XF_PFDL_t
  Physical FiFo Descriptor List Item.
- struct XF_VFDL_t
  Virtual FiFo Descriptor List Item.

MMU Macros

- #define OSB_MMUSETADDR(addr, res) OSB_WRITE(OSBOPC_-MMUSETADDR,(addr+((res)<<MMU_OFF_RES)))
  Set address and resource ID.
- #define OSB_MMUVFDLADDR(tid, trfid) (trfid+((tid)<<MMU_OFF_TID))
  Calculate address from task ID and task relative fifo ID.
- #define OSB_MMUPFDLADDR(pfid, fld) (fld+((pfid)<<MMU_OFF_PFID))
  Calculate address from physical FiFo ID and the requested field in the List.
- #define OSB_MMUXFIFO_EMTPY(data) ((data) & MMU_MSK_XFI_EMPTY)
  Check for Xilinx CoreGen FiFo's empty flag.
- #define OSB_MMUXFIFO_FULL(data) ((data) & MMU_MSK_XFI_FULL)
  Check for Xilinx CoreGen FiFo's full flag.
- #define OSB_MMUXFIFO_LEVEL(data) ((data) & MMU_MSK_XFI_FILL)
  Get Xilinx CoreGen FiFo's fill-level.
- #define OSB_MMUSETADDR_PFDL(pfid, fld)
  Set address to write to PFDL.
- #define OSB_MMUSETADDR_VFDL(tid, trfid)
  Set address to write to PFDL.
- #define MMU_READ() OSB_READ(OSBOPC_MMUREAD)
  Read from MMU.
- #define MMU_WRITE(val) OSB_WRITE(OSBOPC_MMUWRITE, val)
  Write to MMU.

Opcodes for MMU

- #define OSBOPC_MMUSETADDR 0x44
  Write address / resource ID to MMU.
- #define OSBOPC_MMUREAD 0x45
  Read from MMU.
- #define OSBOPC_MMUWRITE 0x45
  Write to MMU.

MMU Resource IDs

- #define MMU_RES_VFDL 0
  Virtual FiFo description list.
- #define MMU_RES_PFDL 1
  Physical FiFo description list.
- #define MMU_RES_SRAM1 2
  SRAM, bank 1.
- #define MMU_RES_SRAM2 3
  SRAM, bank 2.
- #define MMU_RES_BRAM 4
  BRAM.
- #define MMU_RES_XFIFO 5
  Xilinx CoreGen FiFo.

FiFo Types

- #define MMU_FTYPE_SRAM1 0
  FiFo implemented in SRAM bank 1.
- #define MMU_FTYPE_SRAM2 1
  FiFo implemented in SRAM bank 2.
- #define MMU_FTYPE_BRAM 2
  FiFo implemented in BlockRAM.
- #define MMU_FTYPE_XILINX 3
  Xilinx CoreGen FiFo.

MMU Physical FiFo description list fields

- #define MMU_FLD_TPSZ 0
  Type.
- #define MMU_FLD_BASE 1
  Base address.
- #define MMU_FLD_RDPT 2
  Read pointer.
- #define MMU_FLD_WRPT 3
  Write pointer.

MMU Xilinx CoreGen FiFos

- #define MMU_XFI_ETHTX 0
  ETH transmit FiFo.
- #define MMU_XFI_ETHARP 1
  ETH ARP FiFo.
- #define MMU_XFI_ETHUDP1 2
  ETH UDP FiFo 1.
- #define MMU_XFI_ETHUDP2 3
  ETH UDP FiFo 2.
- #define MMU_XFI_UARTTX 4
  RS-232 transmit FiFo.
- #define MMU_XFI_UARTRX 5
  RS-232 receive FiFo.
- #define MMU_XFI_AUDIOTX 6
  Audio transmit FiFo.
- #define MMU_XFI_AUDIORX1 7
  Audio receive FiFo 1.
- #define MMU_XFI_AUDIORX2 8
  Audio receive FiFo 2.

Data Masks

- #define MMU_MSK_XFI_FILL (1<<4)
  Flag to read Xilinx CoreGen FiFo's fill-level information.
- #define MMU_MSK_XFI_EMPTY (1<<15)
  Mask for the Xilinx CoreGen FiFo's empty flag.
- #define MMU_MSK_XFI_FULL (1<<14)
  Mask for the Xilinx CoreGen FiFo's full flag.
- #define MMU_MSK_XFI_COUNT 0x3FFF
  Mask for the Xilinx CoreGen FiFo's fill-level.
- #define MMU_MSK_TID 0x7
  Mask for the task ID.
- #define MMU_MSK_TRFID 0x7
  Mask for the task-relative FiFo ID.
- #define MMU_MSK_SIZE 0x3F
  Mask for the physical FiFo size.
- #define MMU_MSK_TYPE 0x7
  Mask for the physical FiFo type.
- #define MMU_MSK_VFDLDIR 0x10
  Mask for the direction bit in the VFDL.
- #define MMU_MSK_VFDLPFID 0xF
  Mask for the physical FiFo ID in the VFDL.

Offsets

- #define MMU_OFF_TID 3
  Offset of task ID.
- #define MMU_OFF_PFID 2
  Offset of physical FiFo ID.
- #define MMU_OFF_TYPE 6
  Offset of physical FiFo type.
- #define MMU_OFF_VFDLDIR 4
  Offset of the direction bit in the VFDL.
- #define MMU_OFF_RES 20
  Offset of resource ID.

Defines

- #define MMU_MIN_FIFO_SIZE 0x100
  Minimum size of a FiFo (in bytes).
- #define MMU_FIFO_BLOCK_SIZE MMU_MIN_FIFO_SIZE
  Block size for FiFos.
- #define MMU_MAX_FIFO_SIZE 0x4000
  Maximum size of a FiFo.
- #define MMU_SEQUENTIAL XTRUE
  Constant to initiate a sequential read / write access to the MMU.
- #define MMU_PUNCUTAL XFALSE
  Constant to initiate a punctual read / write access to the MMU.

Typedefs

- typedef XF_VFDL_t XF_VFDL
  Virtual FiFo Descriptor List Item.
- typedef XF_PFDL_t XF_PFDL
  Physical FiFo Descriptor List Item.

Functions

- void mmu_readVFDL (XF_VFDL *vfdlPtr, Xuint32 length, Xboolean sequential)
  Read VFDL.
- void mmu_writeVFDL (XF_VFDL *vfdlPtr, Xuint32 length, Xboolean sequential)
  Write VFDL.
- void mmu_readPFDL (XF_PFDL *pfdlPtr, Xuint32 length, Xboolean sequential)
  Read PFDL.
- void mmu_writePFDL (XF_PFDL *pfdlPtr, Xuint32 length, Xboolean sequential)
  Write PFDL.
- Xuint32 mmu_writeToFifo (XF_PFDL *pfdlPtr, Xuint16 value, Xboolean first)

Write to FiFo.

- Xuint32 mmu_readFromFifo (XF_PFDL *pfdlPtr, Xboolean first)
  Read from FiFo.
- Xuint32 mmu_fifoFillLevel (XF_PFDL *pfdlPtr)
  Get FiFo Fill-Level.
- void mmu_dumpFifo (XF_PFDL *pfdlPtr, Xuint32 start, Xuint32 end)
  Dump FiFo Contents.

8.2.18.2 Define Documentation

#define OSB_MMUSETADDR_PFDL(pfid, fld)

Value:

```
OSB_MMUSETADDR(\
                                        OSB_MMUPFDLADDR(pfid,fld),MMU_RES_PFDL)
```

#define OSB_MMUSETADDR_VFDL(tid, trfid)

Value:

```
OSB_MMUSETADDR(\
                                          OSB_MMUVFDLADDR(tid,trfid),MMU_RES_VFDL)
```

8.2.18.3 Typedef Documentation

typedef struct XF_PFDL_t XF_PFDL

An item in the physical FiFo descriptor list contains information about the type and the dimensions of a FiFo physically present in the design.

typedef struct XF_VFDL_t XF_VFDL

An item in the virtual FiFo descriptor list contains information about the mapping between tasks and the read / write interfaces of the FiFos listed in the physical FiFo descriptor list.

8.2.19 network.c File Reference

8.2.19.1 Detailed Description

This file includes all functions needed for network handling, like ping, UDP/ICMP/ARP reply, and other basic IP functions. Most functions can be put into a mode of higher verbosity, i.e. for debugging, by defining the symbol `NET_DEBUG`

Author:
Marco Kuster, Samuel Nobs

Date:
2004-04-01
Revision
1.10

Functions

- void net_init ()
  Initialize Network.
- void net_setphymode (Xboolean mode)
  Set PHY chip's operation mode.
- Xboolean IsPacketAvailable ()
  Packet Availability.
- void PacketAnalyzer ()
  Analyze Packet.
- void net_PingReply (XContactInfo ∗contactInfo)
  Ping Reply.
- void net_RecvUDP (XContactInfo ∗contactInfo)
  Receive UDP Packet.
- Xboolean XFNetCmdWriteRAM (Xuint16 cmdseqnr, Xuint8 ∗srcadr, Xuint8 ∗destadr, Xuint16 len, XContactInfo ∗contactInfo)
  XFNet Command: Write RAM.
- Xboolean XFNetCmdReadRAM (Xuint16 cmdseqnr, Xuint8 ∗srcadr, Xuint16 len, XContactInfo ∗contactInfo)
  XFNet Command: Read RAM.
- Xboolean SendUDPTo (Xuint8 bytDestinationIP[4], Xuint8 bytDestinationMAC[6], Xuint16 intDestinationPort, Xuint8 ∗ptrDataBuffer, Xuint16 DataLength)
  Send UDP Packet.
- void SetMDIO (Xboolean bit)
  Set MDIO.
- void DoMDIOClk ()
  Generate MDIO Clock.
- Xuint16 ReadMDIO (Xuint8 reg)
  Read MDIO.
- void WriteMDIO (Xuint8 argReg, Xuint16 argValue)
  Write MDIO.
- void net_setMAC (Xuint8 ∗mac)
  Set MAC Address.
- void net_setIP (Xuint8 ∗ip)
  Set IP Address.
- void net_ARPReply (XContactInfo ∗contactInfo)
  ARP Reply.

- Xuint16 calc_checksum32 (Xuint32 *buf, Xuint16 len)
  Calculate Checksum.
- Xuint16 calc_UDPchecksum (Xuint32 *buf, Xuint16 *PseudoHeader, Xuint16 DataLength)
  Calculate UDP Checksum.
- Xboolean net_compareIP (Xuint32 *ipPtr)
  Compare IP.
- XPortListener * net_installPortListener (char *buffer, Xuint32 size, Xuint32 port)
  Install Port Listener.
- Xuint32 net_removePortListener (Xuint32 pid)
  Remove Port Listener.
- Xuint32 net_ipconfig (char **argv, Xuint32 argc)
  Configure Ethernet Interface (shell command).
- void net_parseMAC (char *strMAC, Xuint8 *MAC)
  Parse MAC Address.
- void net_parseIP (char *strIP, Xuint8 *IP)
  Parse IP Address.

Variables

- XPacketInfo ReceivedPacketInfo
  Information about the packet recently received.
- Xuint32 outword
  Shared variable holding configuration data written to the configuration port of the PHY chip.
- Xuint8 net_IPBoardAddress [4]
  The C-FPGA's IP address.
- Xuint8 net_MACBoardAddress [6]
  The C-FPGA's MAC address.
- XPortListener PortListenerList [NET_MAX_PORTLISTENERS]
  Table containing the port listeners.
- Xuint32 net_numPacketsSent
  The number of packets sent.

8.2.19.2 Function Documentation

Xuint16 calc_checksum32 (Xuint32 * buf, Xuint16 len)

This function calculates the checksum of the data in the buffer `buf`.

Parameters:

buf the buffer containing the data to calculate the checksum of

len the length of the data buffer

Returns:

the checksum

Xuint16 calc_UDPchecksum (Xuint32 * buf, Xuint16 * PseudoHeader, Xuint16 DataLength)

This function calculates the checksum of a UDP packet including the pseudo header.

Parameters:

buf the buffer containing the packet

PseudoHeader the packet's pseudo header

DataLength length of the packet's payload

Returns:

the checksum

void DoMDIOClk ()

This function writes a rising and a falling clock edge to the MDIO interface.

Xboolean IsPacketAvailable ()

This function simply checks the PHY chip's control register to see whether a packet is available or not.

Returns:

XTRUE if a packet is available, XFALSE otherwise

void net_ARPReply (XContactInfo * contactInfo)

ARP requests are handled using this function. It sends a reply to the requester described in `contactInfo`.

Parameters:

contactInfo the information about the sender of the ARP request

Xboolean net_compareIP (Xuint32 * ipPtr)

This function compares the IP pointed at by ipPtr with the board's IP.

Parameters:

ipPtr the pointer to the IP to be compared

Returns:

XTRUE if IPs match, XFALSE if they don't

void net_init ()

This function must be called prior to the use of the network services. It initializes the global variables containing the IP and MAC addresses of the board and writes the MAC to the PHY chip. It set the PHY chip's operation mode and initializes the port listener table.

***XPortListener*∗ net_installPortListener (char ∗ buffer, *Xuint32* size, *Xuint32* port)**

If a task needs to receive UDP packets, it may install a port listener listening to port `port`. This function checks whether there is a free slot in the port listener list and allocates a slot if a free one can be found. To guarantee consistency of this list, the interrupts get disabled before accessing it to avoid other function to change its contents. The port listener description is returned to the calling function, which then needs to poll the `done` flag of the descriptor to see whether a packet has been received. If another packet is being expected, the `done` flag can simply be cleared again.

If more than one listener for the same port are installed and ready, the one at the lowest index in the list is used when a packet is received.

Parameters:

buffer the buffer where the data being received should be written to

size the size of the buffer

port the port to be listened to

Returns:

the description of the port listener if a free slot in the list could be found, 0 otherwise

Xuint32 net_ipconfig (char ∗∗ argv, *Xuint32* argc)

This shell command can be used to configure the IP and MAC addresses of the C-FPGA and to display the IP and MAC addresses of both FPGAs.

```
usage: ipconfig <set> <C> <MAC|IP> <Addr>
```

Todo

The configuration of R-FPGA's ethernet interface must be included.

Parameters:

argv pointer to the argument list

argc number of arguments

Returns:

0

void net_parseIP (char ∗ strIP, *Xuint8* ∗ IP)

This function parses an IP address from a string. The string is expected to contain the IP address in decimal format, the bytes being separated by dots.

Parameters:

strIP the string to be parsed

IP the buffer to be filled in the IP address

void net_parseMAC (char* strMAC, Xuint8* MAC)

This function parses a MAC address from a string. The string is expected to contain the MAC address in hexadecimal format, the bytes being separated by colons.

Parameters:

strMAC the string to be parsed

MAC the buffer to be filled in the MAC address

void net_PingReply (XContactInfo* contactInfo)

This function simply replies an ICMP ping request with 32 bytes.

Parameters:

contactInfo the information about the contact requesting the ping reply

void net_RecvUDP (XContactInfo* contactInfo)

This function analyzes an incoming UDP packet. If the packet is an XFNet command, i.e. it is sent to the port number `0xF0CE`, the command is getting executed; it is either a read RAM command, a write RAM command or a write program code command. If the packet does not represent a XFNet command, the list of port listeners is checked to see whether a function waits for data to be received for the current port. If a listener is found, the data of the packet is copied to the buffer of the listener, and a flag is set that this buffer has been filled. If the packet could not be interpreted at all, i.e. no port listener was installed for the packet's destination port or the done flag of the listener of this port is already set, a message is printed to the message window.

Parameters:

contactInfo the information about the contact sending the packet

Xuint32 net_removePortListener (Xuint32 pid)

When a process does not need it's port listeners anymore, it can remove them from the list. Calling this function removes all listeners associated with the given PID.

Todo

A function to remove the port listeners more selectively may be useful.

Parameters:

pid the PID of the process whose port listeners are to be removed

Returns:

0

void net_setIP (Xuint8 * ip)

Using this function, the IP address pointed at by the argument can be copied to the global variable containing the IP address of the C-FPGA, `net_IPBoardAddress`.

Parameters:

ip the pointer to the array containing the IP address

void net_setMAC (Xuint8 * mac)

This function sets the global variable containing the MAC address of the C-FPGA, `net_MACBoardAddress`, and writes it to the PHY chip.

Parameters:

mac the pointer to the array containing the MAC address

void net_setphymode (Xboolean mode)

This function sets the PHY chip's operation mode. It resets the device and selects MDDIS to MDIO control mode. Then it activates the device and waits for at least 0.3 ms. If mode is set to PHY_MODE_AUTO, auto-negation is enabled, advertising that 10 Mbps and full/half duplex is supported. Otherwise, the PHY is configured to 10Mbps half-duplex mode without auto-negotiation.

Parameters:

mode the flag activating auto-negotiation

void PacketAnalyzer ()

The packet analyzer checks the packet and protocol type and calls the according handler functions. Currently, only ARP requests, Pings and UDP packets are supported.

Todo

Additional protocols and packet types could be supported.

Xuint16 ReadMDIO (Xuint8 reg)

The MDIO register `reg` can be read using this function.

Parameters:

reg number of the register to be read

Returns:

the value read from the register

Xboolean SendUDPTo (Xuint8* bytDestinationIP*[4], Xuint8* bytDestinationMAC*[6], Xuint16* intDestinationPort*, Xuint8* ∗ ptrDataBuffer*, Xuint16* DataLength*)

To send a UDP Packet to a host the IP and MAC addresses are known of, this function can be used.

Todo

As the system does not have an ARP table, the full information about the communication partner needs to be known, i.e. the IP and MAC addresses. Additional effort could be invested to include such an ARP table to make sending of packets easier.

Parameters:

bytDestinationIP[4] the IP of the destination

bytDestinationMAC[6] the MAC of the destination

intDestinationPort the destionation port

ptrDataBuffer the buffer containing the data to be transmitted

DataLength the length of the data to be transmitted

Returns:

XTRUE

void SetMDIO (Xboolean* bit*)

This is a simple function to set or clear the MDIO flag.

Parameters:

bit the state of the MDIO to be written

void WriteMDIO (Xuint8* argReg*, Xuint16* argValue*)

Write the value `argValue` to the MDIO register `argReg`.

Parameters:

argReg number of the register to be written to

argValue value to be written

Xboolean XFNetCmdReadRAM (Xuint16* cmdseqnr*, Xuint8* ∗ srcadr*, Xuint16* len*, XContactInfo* ∗ contactInfo*)

This function handles incoming XFNet commands for read access to the RAM. It sends the data stored at the address defined in the command to the sender of the command, repeating the sequence number of the command.

Parameters:

cmdseqnr the sequence number of the command

srcadr the address where to send the data from

len the length of the data to be sent

contactInfo information about the sender of the command

Returns:
XTRUE

Xboolean XFNetCmdWriteRAM (Xuint16* cmdseqnr, *Xuint8* * srcadr, *Xuint8* * destadr, *Xuint16* len, *XContactInfo* * contactInfo*)

This function handles incoming XFNet commands for write access to the RAM. It writes the payload of the packet to the address defined in the command and sends back a confirmation containing the sequence number to the sender.

Parameters:
cmdseqnr the sequence number of the command

srcadr the pointer to the packet data

destadr the address where to store the data

len the length of the data to be stored

contactInfo information about the sender of the command

Returns:
XTRUE

8.2.20 network.h File Reference

8.2.20.1 Detailed Description

This is the header file for `network.c`

Author:
Marco Kuster, Samuel Nobs

Date:
2004-04-01

Revision
1.11

Data Structures

- struct XContactInfo
  Contact Information.
- struct XPacketData
  Packet Data, 32 Bit Version.
- struct XPacketData8
  Packet Data, 8 Bit Version.
- struct XPacketInfo
  Packet Info.
- struct XPortListener
  Port Listener.

Port Definitions

- #define XFBoard_CommandPort 0x22B8
  Port for commands.
- #define XFBoard_DataPort 0x22B9
  Port for data.
- #define XFBoard_UDPPort 0x22
  UDP port.
- #define PC_SOURCEPORT 0xF0CE
  Port for XFNet commands.
- #define XFBoard_DebugPort 0xFDE8
  Debug port.

Network Commands

- #define NETCMD_WRITERAM 0x1000
  Write data to RAM.
- #define NETCMD_READRAM 0x1001
  Read data from RAM.
- #define NETCMD_WRITEPROG 0xC0DE
  Write program code in motorola srec format to RAM.

Defines

- #define NET_SPAWN_SEPARATE_TASKS
  Let the ethernet daemon spawn a separate analyzer task for each packet received.
- #define CNET_IPADDRESS {192,168,1,111}
  IP address of the C-FPGA.
- #define CNET_MACADDRESS {0xF0,0xCE,0x00,0x00,0x00,100}
  MAC address of the C-FPGA.
- #define RNET_IPADDRESS {192,168,1,214}

IP address of the R-FPGA.

- #define RNET_MACADDRESS {0x00,0x01,0x03,0xd8,0x08,0xf7}
  MAC address of the R-FPGA.
- #define HOST_IPADDRESS {192,168,1,10}
  IP address of the host.
- #define HOST_MACADDRESS {0x00,0x0A,0x5E,0x04,0x4D,0x03}
  MAC address of the host.
- #define TX_BASEADDR XPAR_ETHERNET_LITE_BASEADDR
  Base address of the transmit buffer in the ethernet core.
- #define RX_BASEADDR (XPAR_ETHERNET_LITE_BASEADDR+0x2000)
  Base address of the receive buffer in the ethernet core.
- #define RX_CTL_ADDR (XPAR_ETHERNET_LITE_BASEADDR+0x3FFC)
  Address of the ethernet core's control register, receiver side.
- #define TX_LEN_ADDR (TX_BASEADDR+0x1FF4)
  Address pointing to the lower byte of the length of the packet to be transmitted.
- #define TX_LEN_ADDR_HIGHER (TX_BASEADDR+0x1FF4)
  Address pointing to the higher byte of the length of the packet to be transmitted.
- #define TX_CTL_ADDR (TX_BASEADDR+0x1FFC)
  Address of the ethernet core's control register, transmit side.
- #define PHY_MODE_AUTO XTRUE
  Enable auto-negotiation.
- #define PHY_MODE_FIX XFALSE
  Disable auto-negotiation.
- #define NET_MAX_PACKETLENGTH 1500
  Maximum length of a packet.
- #define NET_MAX_PORTLISTENERS 5
  Maximum number of port listeners.

Enumerations

- enum XPacketType { packetTypeNONE = 0x0, packetTypeARP = 0x0806, packetTypeIP = 0x0800 }
  Packet types.
- enum XIPType {

  ipTypeNOIP = 0, ipTypeICMP = 1, ipTypeIGMP = 2, ipTypeGGP = 3,

  ipTypeTCP = 6, ipTypeEGP = 8, ipTypeUDP = 17 }
  Protocol Type.
- enum XARPType { arpTypeREQ = 0x0001 }
  Type of ARP packet.

Functions

- Xboolean IsPacketAvailable ()
  Packet Availability.

- Xboolean SendUDPTo (Xuint8 bytDestinationIP[4], Xuint8 bytDestinationMAC[6], Xuint16 intDestinationPort, Xuint8 *ptrDataBuffer, Xuint16 DataLength)
  Send UDP Packet.
- Xboolean **UDPReply** (Xuint16 DestinationPort, Xuint32 *ptrDataBuffer)
- Xboolean net_compareIP (Xuint32 *ipPtr)
  Compare IP.
- Xuint16 ReadMDIO (Xuint8 reg)
  Read MDIO.
- Xuint16 calc_UDPchecksum (Xuint32 *buf, Xuint16 *PseudoHeader, Xuint16 DataLength)
  Calculate UDP Checksum.
- Xuint16 calc_checksum32 (Xuint32 *buf, Xuint16 len)
  Calculate Checksum.
- Xuint16 **calc_chksum** (Xuint8 *buf, int len)
- void DoMDIOClk ()
  Generate MDIO Clock.
- void PacketAnalyzer ()
  Analyze Packet.
- void SetMDIO (Xboolean bit)
  Set MDIO.
- void WriteMDIO (Xuint8 argReg, Xuint16 argValue)
  Write MDIO.
- void net_ARPReply (XContactInfo *contactInfo)
  ARP Reply.
- void net_PingReply (XContactInfo *contactInfo)
  Ping Reply.
- void net_RecvUDP (XContactInfo *contactInfo)
  Receive UDP Packet.
- void net_init ()
  Initialize Network.
- void net_setIP (Xuint8 *ip)
  Set IP Address.
- void net_setMAC (Xuint8 *mac)
  Set MAC Address.
- void net_setphymode (Xboolean mode)
  Set PHY chip's operation mode.
- Xboolean XFNetCmdReadRAM (Xuint16 cmdseqnr, Xuint8 *srcadr, Xuint16 len, XContact-Info *contactInfo)
  XFNet Command: Read RAM.
- Xboolean XFNetCmdWriteRAM (Xuint16 cmdseqnr, Xuint8 *srcadr, Xuint8 *destadr, Xuint16 len, XContactInfo *contactInfo)
  XFNet Command: Write RAM.
- XPortListener * net_installPortListener (char *buffer, Xuint32 size, Xuint32 port)
  Install Port Listener.
- Xuint32 net_removePortListener (Xuint32 pid)

Remove Port Listener.

- Xuint32 net_ipconfig (char **argv, Xuint32 argc)
  Configure Ethernet Interface (shell command).
- void net_parseMAC (char *strMAC, Xuint8 *MAC)
  Parse MAC Address.
- void net_parseIP (char *strIP, Xuint8 *IP)
  Parse IP Address.

Variables

- Xuint8 net_IPBoardAddress [4]
  The C-FPGA's IP address.
- Xuint8 net_MACBoardAddress [6]
  The C-FPGA's MAC address.
- Xuint32 net_numPacketsSent
  The number of packets sent.

8.2.20.2 Enumeration Type Documentation

enum XARPType

Enumeration values:

arpTypeREQ ARP request.

enum XIPType

Enumeration values:

ipTypeNOIP No IP packet.

ipTypeICMP Internet control message protocol.

ipTypeIGMP Internet group management protocol.

ipTypeGGP Generic gateway protocol.

ipTypeTCP Transmission control protocol.

ipTypeEGP Exterior gateway protocol.

ipTypeUDP User datagram protocol.

enum XPacketType

Enumeration values:

packetTypeNONE no specific type

packetTypeARP ARP packet.

packetTypeIP IP packet.

8.2.21 osbridge.h File Reference

8.2.21.1 Detailed Description

This file defines the macros for the communication with the R-FPGA. Some of the opcodes needed for the communication are described in this file, but most of them can be found in the header files of the service they are belonging to.

Author:
Samuel Nobs

Date:
2004-02-06

Version: **Revision**
1.4

General Macros

- #define OSB_WRITE(opcode, val) (∗((Xuint32∗)(XPAR_OS_BRIDGE_-BASEADDR+opcode))=(Xuint32)val)
  Write to OS Bridge.
- #define OSB_READ(opcode) (Xuint32)(∗((Xuint32∗)(XPAR_OS_BRIDGE_-BASEADDR+opcode)))
  Read from OS Bridge.

Defines

- #define OSBOPC_TASKCIFWRITE 0x40
  Write to the Task Control Interface.
- #define OSBOPC_TASKCIFREAD 0x40
  Read from the Task Control Interface.
- #define OSBOPC_TAKSCIFFREEZE 0x00
  Write freeze signal.

8.2.22 scheduler.c File Reference

8.2.22.1 Detailed Description

This file contains algorithms for simple round-robin scheduling and the dispatching mechanisms for saving and restoring the context of a task.

Author:
Samuel Nobs

Date:
2004-02-10
Revision
1.11

Functions

- void sch_idle ()
  Idle Task.
- void sch_dispatch (TaskDescriptor *task, Xuint32 stackHiAddr, Xuint32 stackLoAddr)
  Dispatch Task.
- void sch_saveContext (TaskDescriptor *task)
  Save Task Context.
- void sch_start (char **argv, Xuint32 argc, TaskDescriptor *task, Xuint32 stackHiAddr, Xuint32 stackLoAddr)
  Start Task.
- void sch_schedule ()
  Round Robin Scheduler.
- void sch_init ()
  Initialize Scheduler.
- XFError sch_addToTaskList (FunctionPointer task, char **argv, Xuint32 argc, char *name, Xuint32 quanta, Xuint32 stackSize, Xboolean suspendShell)
  Add Task to Task List.
- void sch_finished ()
  Task Finished.
- void sch_setStackWatch ()
  Set Stack Watch.
- void sch_block ()
  Mark Task as Blocked.
- void sch_unblock ()
  Mark Task as Running.
- void sch_kill (Xuint32 pid)
  Kill Process.
- void sch_context (Xuint32 pid)
  Dump Process Context.

Variables

- TaskDescriptor TaskList [SCH_TASKLIST_LENGTH]
  A list describing the tasks being scheduled.
- Xuint32 runningTask
  PID of the task currently running.

8.2.22.2 Function Documentation

***XFError** sch_addToTaskList (**FunctionPointer** **task**, **char** ** **argv**, **Xuint32** **argc**, **char** * **name**, **Xuint32** **quanta**, **Xuint32** **stackSize**, **Xboolean** **suspendShell**)*

This function adds a task being referred to by a `FunctionPointer` to the `TaskList`. The list is searched for a free task placeholder represented by a `TaskDescriptor` with its field `status` set to `STATUS_UNUSED` or `STATUS_KILLED`. Then, the placeholder's fields are filled in the information given in the arguments.

Todo

This function is also used by processes to spawn child processes. The OS must be extended to do some bookkeeping about the dependencies between processes and their childs.

Parameters:

task the pointer to the function beginning in the program code

argv the array containing the argument strings

argc the number of arguments

name the name of the task

quanta the number of time quanta assigned to the task

stackSize the minimal stack requirements

suspendShell the flag used to determine if the command suspends the shell

Returns:

An `XFError`:

- `ERR_NONE` if task was successfully added to the `TaskList`
- `ERR_NO_FREE_TASKSLOT` on failure, i.e. no free placeholder was found.

void sch_block ()

This function is called to set the `status` of the current task's `TaskDescriptor` to `STATUS_BLOCKED`

void sch_context (Xuint32* **pid**)

To dump the register context of a process, this function can be called. It simply prints out the values that have been saved in the descriptor of the process.

Parameters:

pid the ID of the process

void sch_dispatch (TaskDescriptor * task, Xuint32 stackHiAddr, Xuint32 stackLoAddr)

This function is actually written in assembly language and can be found in scheduler.s. It restores the context of a task by saving the contents of the field `context` of the tasks `TaskDescriptor` back to their original registers. Additionally, it writes the limits of the memory space used for the process' stack to the core watching the stack pointer `r1`. As the program counter is restored too, the execution of the program continues exactly where it was stopped before.

Parameters:

task `TaskDescriptor` of the task to be dispatched

void sch_finished ()

This helper function is called by every task who finishes execution: the timer is set to 0 to recede from CPU.

void sch_init ()

Before the scheduler is started, the `TaskList` has to be set to its initial state with all task placeholders empty (`STATUS_UNUSED`) except for the placeholder at `TaskList`[0] which is assigned the system idle task.

void sch_kill (Xuint32 pid)

To properly kill a process with PID *pid*, this function should be called. It marks the process with the status `STATUS_KILLED` in the process list, deallocates all its memory, removes its graphic hooks from the graphic manager, removes all its port listeners and finally returns the lock if this process owned it.

Todo

This kill function does not yet check the process dependencies. It mus be extended to kill child processes of a process too. However, it is the OS that does not keep track of such dependencies either.

Parameters:

pid the ID of the process to be killed

void sch_saveContext (TaskDescriptor * task)

This function is written in assembly language and can be found in the file schedule.s. The register contents of the task that has been pre-empted by the scheduler before are saved to the field `context` of the task's `TaskDescriptor` . It is a tedious challenge to gather these contents due to the following:

- as every call to the scheduler is preceded by an interrupt, some of the registers of the task being interrupted are put on the stack of the `interrupt_handler()`
- as the `sch_schedule()` function being executed next uses some additional registers, their content is stored in the stack of `sch_schedule()`
- some of the registers are not touched by the `interrupt_handler()` and `sch_schedule()`, therefore they have not been saved so far. Their content is still unchanged.

This means that the values of the registers have to be grabbed from three different locations. As a result, scheduler.s must be compared with the assembly language listing of the `sch_schedule()` function whenever a change in the scheduler's source code has been performed, because `sch_schedule()` now might use other registers, which means that their values will be stored somewhere else than before. The same measure must be applied when the compiler's optimization level is altered from `-O3`.

Parameters:

task `TaskDescriptor` of the task whose context is to be saved

void sch_schedule ()

This function is the actual scheduler. It is called through the `interrupt_handler()` whenever the timer has reached 0. To figure out how it works, its course of action is described here:

- first, the task currently running on the CPU is pre-empted
- if the task was in the state `STATUS_RUNNING` and has not yet reached the end of execution, its context is stored using `sch_saveContext(TaskDescriptor*)` and its state is set to `STATUS_READY`
- if the task has terminated, which means that it has reached the end of execution, its placeholder is set to `STATUS_UNUSED` and the memory that has been allocated is freed using `mem_dealloc(Xuint32)`. So the task is physically removed from the `TaskList` now. If the task did not release a lock that it has taken, this lock is now released by the scheduler.
- then the `TaskList` is searched for the next task that is ready for execution on the CPU
- if the task found has been newly added to the `TaskList` and is assigned the CPU for the first time, memory for its stack is allocated using `mem_allocStack(Xuint32, StackDescriptor*)`. Then it is started by setting the program counter to the task's start point in the program code.
- if the task is not new, its context is restored, and then the execution is continued at the very point it has been interrupted.

void sch_setStackWatch ()

This function is called whenever a process is started for the first time or resumed on the beginning of its next time slice. It is called to set the upper and the lower limit of the memory space used for the stack of the process. The OPB core watching the stack raises an interrupt as soon as the stack pointer `r1` points to a location outside of this memory space. The function is written in assembly language; the arguments are stored in registers `r8` (upper limit) and `r9` (lower limit)

void sch_start (char ∗∗ argv, Xuint32 argc, TaskDescriptor ∗ task, Xuint32 stackHiAddr, Xuint32 stackLoAddr)

When a task is started for the first time, this function is used to pass the arguments, adjust the stack pointer and finally jumping to the start position in the program code, setting the return address to sch_finished() at the same time. Adjusting the stack pointer also includes writing the upper and the lower limit of the memory space valid for the stack to the hardware that observes the stack pointer `r1`.

Parameters:

argv Pointer to the argument list

argc Number of arguments

task Pointer to the `TaskDescriptor` of the task about to be started

stackHiAddr High address of the stack memory

stackLoAddr Low address of the stack memory (for future use)

void sch_unblock ()

This function is called to set the `status` of the current task's `TaskDescriptor` to `STATUS_RUNNING`

8.2.23 scheduler.h File Reference

8.2.23.1 Detailed Description

This is the header file for scheduler.c and scheduler.s.

Author:

Samuel Nobs

Date:

2004-02-10

Revision

1.12

Data Structures

- struct ContextDescriptor_t
  Task Context Descriptor.
- struct TaskDescriptor_t
  Task Descriptor.

Status of a Task or it's Placeholder

- #define STATUS_UNUSED 0
  Task placeholder is empty.
- #define STATUS_NEW 1
  Task has been added to the `TaskList`*, but has not been started yet.*
- #define STATUS_RUNNING 2
  Task is currently running on the CPU.
- #define STATUS_READY 3
  Task is ready for being executed.
- #define STATUS_BLOCKED 4
  Task is blocked, e.g. due to a call to `LOCK()`.
- #define STATUS_SUSPENDED 5
  Task is suspended.
- #define STATUS_KILLED 6
  Task has been killed.

Defines

- #define SCH_TASKLIST_LENGTH 15
  Maximum number of tasks handled by the scheduler.
- #define SCH_DEFAULT_QUANTA 1
  Default number of time quanta assigned to a newly launched task.
- #define SCH_INFINITE_QUANTA 0xFFFFFFFF
  Maximum number of time quanta that can be assigned to a task.
- #define SCH_RECEDE() ∗((volatile Xuint32∗)XPAR_TIMER_BASEADDR)=0
  Recede from CPU by setting timer maximum to 0.
- #define SCH_RECEDING_LOCK()

Typedefs

- typedef ContextDescriptor_t ContextDescriptor
  Task Context Descriptor.
- typedef TaskDescriptor_t TaskDescriptor
  Task Descriptor.

Functions

- XFError sch_addToTaskList (FunctionPointer task, char **argv, Xuint32 argc, char *name, Xuint32 quanta, Xuint32 stackSize, Xboolean suspendShell)
  Add Task to Task List.
- void **exit** ()
- void **sch_saveContext** (TaskDescriptor *task)
- void **sch_start** (char **argv, Xuint32 argc, TaskDescriptor *task, Xuint32 stackHiAddr, Xuint32 stackLoAddr)
- void sch_block ()
  Mark Task as Blocked.
- void **sch_finish** ()
- void sch_finished ()
  Task Finished.
- void sch_init ()
  Initialize Scheduler.
- void sch_schedule ()
  Round Robin Scheduler.
- void sch_setStackWatch ()
  Set Stack Watch.
- void sch_unblock ()
  Mark Task as Running.
- void sch_kill (Xuint32 pid)
  Kill Process.
- void sch_context (Xuint32 pid)
  Dump Process Context.

Variables

- Xuint32 runningTask
  PID of the task currently running.
- TaskDescriptor TaskList [SCH_TASKLIST_LENGTH]
  A list describing the tasks being scheduled.
- Xuint32 PIDsuspendingShell
  The PID of the process suspending the shell.

8.2.23.2 Define Documentation

#define SCH_RECEDING_LOCK()

Value:

```
sch_block();\
asm volatile("LA_%=:\n"\
"lwi r12, r0, %0\n"\
```

```
"beqi r12, LB_%=\n"\
"swi  r0, r0, %1\n"\
"bri  LA_%=\n"\
"LB_%=:\n"\
:: "i" (XPAR_HWLOCK_BASEADDR), "i" (XPAR_TIMER_BASEADDR)\
:"r12")
```

Lock, and recede from CPU if already locked by another task. This macro is partially written in assembly language as the compiler generates garbage when it is coded in C

8.2.23.3 Typedef Documentation

typedef struct ContextDescriptor_t ContextDescriptor

The task context descriptor is used for the context switches performed by the scheduler. As soon as a task is pre-empted, its register contents, the context, are stored in its task context descriptor to be loaded back into the registers the next time the task is activated. You should not change this struct without adjusting the file scheduler.s as the functions implemented there rely on the order and the presence of all members of the context descriptor

typedef struct TaskDescriptor_t TaskDescriptor

The task descriptor, which is used as a placeholder for a task / process in the `TaskList`, holds important information about a process such as a backup of its context and its arguments, to name only a few. A placeholder is either empty (`STATUS_UNUSED` or `STATUS_KILLED`) and can be assigned to a new task, or it is occupied (`STATUS_NEW`, `STATUS_RUNNING`, `STATUS_READY` or `STATUS_BLOCKED`). You should not change the order of the the first three members of this struct without adjusting the file scheduler.s as the functions implemented there strongly rely on the order and the presence of these members of the task descriptor

8.2.24 selectmap.c File Reference

8.2.24.1 Detailed Description

In this file the functions used for the configuration of the R-FPGA using the SelectMAP port are implemented.

Author:
Herbert Walder, Samuel Nobs

Date:
2004-04-05
Revision
1.8

Functions

- Xboolean selmap_init ()
  Initialize SelectMAP.
- void selmap_erase ()
  Erase.
- Xboolean selmap_Configure (Xuint32 startadr, Xuint32 len, Xuint32 type)
  Configure.
- void SetCS (Xboolean value)
  Set Chip Select.
- Xboolean GetInit ()
  Get Init Bit.
- void SetProgram (Xboolean value)
  Set Program Bit.
- void SetRDRW (Xboolean value)
  Set Direction.
- Xboolean GetDone ()
  Get Done Bit.
- Xuint32 selmap_display (Xuint8 vpos)
  Display R-FPGA Occupancy.
- Xuint32 selmap_installInGui (char *bitName, Xuint32 startslot, Xuint32 endslot)
  Install Bitstream in Display.

Variables

- Xuint32 volatile selmap_output
  Shared variable holding the data written to the SelectMAP port.
- Xuint32 selmap_slotTable [SELMAP_NUM_SLOTS]
  Table containing the mapping between task slot number and task number.
- char selmap_slotNames [SELMAP_NUM_SLOTS][SELMAP_GFX_HEIGHT]
  List containing the names of the bitstreams configured to the various task slots.
- Xuint32 selmap_uniqueID = 1
  Counter variable used to assign unique IDs to the tasks configured to R-FPGA.
- Xuint32 selmap_hStringPos [7]
  Array containing the positions of the strings in the R-FPGA display.
- Xuint32 selmap_hLinePos [7]
  Array containing the positions of the vertical lines in the R-FPGA display.

8.2.24.2 Function Documentation

Xboolean GetDone () `[inline]`

This helper function monitors the done bit of the SelectMAP port. It is declared as an inline function as it needs to be executed as fast as possible.

Returns:
XTRUE if the done bit is set, XFALSE otherwise

Xboolean GetInit () [inline]

This helper function reads the init bit from the SelectMAP port and returns its value. It is declared as an inline function as it needs to be executed as fast as possible.

Returns:
XTRUE if the init bit is high, XFALSE otherwise

Xboolean selmap_Configure (Xuint32 **startadr,** *Xuint32* **len,** *Xuint32* **type)**

This function writes the configuration of length *len* found at address *startaddr* to the R-FPGA using the SelectMAP port. It supports partial and full bitstreams which are distinguished using the argument *type*.

Todo
The approach to configure the R-FPGA by setting the SelectMAP data lines by software is rather slow. A DMA controller for this task is being developped. As soon as this controller is ready, this function must be adapted accordingly.

Parameters:
startadr the start address of the biststream

len the length of the bitsream

type the type of the bitstream: full (1) or partial (0) bitstream

Returns:
XTRUE

Xuint32 selmap_display (Xuint8 **vpos)**

This is a graphics hook-up function displaying the occupancy of the R-FPGA. The task slots being currently used are shaded and labelled with the name of the bitstream currently being configured into. This function can be added to the pool of graphics functions handled by the `gfx_graphicManager()`.

Parameters:
vpos the vertical position of the display. If equal to GFX_GET_SIZE defined in `graphix.h`, this means that someone wants to know the number of lines needed for this graphics element

Returns:
0 in normal mode, the needed number of lines if asked for

void selmap_erase ()

This function completely removes the configuration of the R-FPGA and empties the configuration table `selmap_slotTable`.

Xboolean selmap_init ()

Before starting to write any configuration data to the SelectMAP port, it must be initialized using this function. This means setting the correct port direction and assigning meaningful default values to the lines connecting to the R-FPGA. The `selmap_slotTable` is set to all empty.

Returns:

Xuint32 selmap_installInGui (char* * **bitName**, *Xuint32* **startslot**, *Xuint32* **endslot**)

Whenever a bitstream is configured to the R-FPGA, its name and the area it is occupying can be added to the display using this function. The lists and tables holding the data relevant for the display, `selmap_slotNames`, `selmap_slotTable`, `selmap_hLinePos` and `selmap_hStringPos` are updated according to the information given in the arguments.

Parameters:

bitName the name of the bitstream

startslot the first slot occupied by the bitstream

endslot the last slot occupied by the bitstream

Returns:

0

***void SetCS (Xboolean* **value**) `[inline]`

This helper function sets or clears the chip select signal according to *value*. It is declared as an inline function as it needs to be executed as fast as possible.

Deprecated

This function will be meaningless as soon as the DMA configuration controller will be included in the design.

Parameters:

value 0: set chip select, 1: clear chip select

void SetProgram (Xboolean* value*) `[inline]`

This helper function sets or clears the program bit according to the parameter *value*. It is declared as an inline function as it needs to be executed as fast as possible.

Parameters:

value 0: set program bit, 1: clear program bit

void SetRDRW (Xboolean* value*) `[inline]`

This helper function sets the direction of the access to the SelectMAP port. When *value* is equal to 1, this means that the port should be configured for a read access. If *value* is 0, this means that a write access is about to occur. The function is declared as an inline function as it needs to be executed as fast as possible.

Parameters:

value 0: write, 1: read

8.2.25 selectmap.h File Reference

8.2.25.1 Detailed Description

This is the header file for `selectmap.c`.

Author:

Date:

2004-04-05

Revision

1.6

Data Structures

- struct BITS
  Bitstream Descriptor.
- struct XFFAT
  Bitstream Allocation Table.

Defines

- #define SELMAP_XFFAT_BASEADDR XPAR_SDRAM_BASEADDR
  Base address of the bitstream allocation table.

- #define SELMAP_DATA_MASK 0xFF
  Mask for the data lines of the GPIO connected to the SelectMAP.
- #define SELMAP_CLOCK_MASK 0x100
  Mask for the clock line of the GPIO connected to the SelectMAP.
- #define SELMAP_WRITE_MASK 0x2400
  Mask used to set the GPIO's tristate buffers for writing data.
- #define SELMAP_READ_MASK 0x24FF
  Mask used to set the GPIO's tristate buffers for reading data.
- #define SELMAP_GETSTATUS 0
  Constant for getting the status register's content using `selmap_getStatus(Xuint32)`.
- #define SELMAP_GETCONFIG 1
  Constant for getting the configuration options register's content using `selmap_getStatus(Xuint32)`.
- #define SELMAP_GFX_FRAME_WIDTH 29
  Width of the frame used for the R-FPGA display (in characters).
- #define SELMAP_GFX_SLOT_WIDTH 4
  Width of the slots in the R-FPGA display (in characters).
- #define SELMAP_GFX_HEIGHT 13
  Height of the area inside the frame used for the R-FPGA display (in lines).
- #define SELMAP_GFX_HOFFSET 18
  Horizontal offset of the R-FPGA display (in characters).
- #define SELMAP_NUM_SLOTS 7
  Number of task slots in the R-FPGA.
- #define SELMAP_CLOCK()
  Clock Macro.

Functions

- Xboolean selmap_init ()
  Initialize SelectMAP.
- Xboolean selmap_Configure (Xuint32 startadr, Xuint32 len, Xuint32 type)
  Configure.
- Xuint32 **selmap_getStatus** ()
- void selmap_erase ()
  Erase.
- void **WriteByte** (Xuint8 argByte)
- void SetCS (Xboolean value)
  Set Chip Select.
- Xboolean GetInit ()
  Get Init Bit.
- void SetProgram (Xboolean value)
  Set Program Bit.
- void SetRDRW (Xboolean value)
  Set Direction.
- Xboolean GetDone ()
  Get Done Bit.

- Xuint32 selmap_display (Xuint8 vpos)
  Display R-FPGA Occupancy.
- Xuint32 **testConfig** (char **argv)
- Xuint32 selmap_installInGui (char *bitName, Xuint32 startslot, Xuint32 endslot)
  Install Bitstream in Display.

8.2.25.2 Define Documentation

#define SELMAP_CLOCK()

Value:

```
selmap_output |= SELMAP_CLOCK_MASK;\
XGpio_mWriteReg(XPAR_SELECTMAP_BASEADDR, XGPIO_DATA_OFFSET, selmap_output);\
selmap_output &= ~SELMAP_CLOCK_MASK;\
XGpio_mWriteReg(XPAR_SELECTMAP_BASEADDR, XGPIO_DATA_OFFSET, selmap_output);\
```

This macro creates a rising and a falling clock edge on the SelectMAP port. As it is used rather frequently, this function is defined as a macro to be directly included in the code to avoid the context switches being inherent to regular function calls.

8.2.26 srec.c File Reference

8.2.26.1 Detailed Description

This file implements the functions used to handle code being uploaded in the Motorola SREC format.

Author:
Samuel Nobs

Date:
2004-04-05
Revision
1.7

Functions

- int decode_srec_line8 (Xuint8 *buffer, Xuint32 len)
  Decode SREC Line from UDP.
- int decode_srec_line (char *buffer)
  Decode SREC Line.
- void srec_initForDisplay (Xuint32 srecLen)
  Initialize for Display.

- Xuint32 srec_installStatusHook ()
  Install Status Display.
- Xuint32 srec_loadingStatus (Xuint32 vpos)
  Display Status.

Variables

- volatile Xuint32 srec_numberOfLines = 0
  Number of SREC lines to be sent in the current upload.
- volatile Xuint32 srec_lineCount = 0
  Number of SREC lines uploaded so far.

8.2.26.2 Function Documentation

int decode_srec_line (char * buffer)

This function decodes an SREC line and writes its contents to memory. As the length of the line is included in the SREC format, no buffer length needs to be given.

Parameters:

buffer the buffer containing the code in SREC format

Returns:

the number of 32 bit words decoded and written to memory

int decode_srec_line8 (Xuint8 * buffer, Xuint32 len)

As the data received by the ethernet core are in a special format, this function is to be used to decode an SREC line received in a UDP packet. The ethernet core stores the bytes received in the packet in a 32 bit word. These bytes are first copied to a buffer of 8 bit words and then submitted to `decode_srec_line(char*)` to do the SREC decoding.

Parameters:

buffer the buffer containing the SREC line

len the length of the buffer

Returns:

the number of 32 bit words decoded

void srec_initForDisplay (Xuint32 srecLen)

Prior to using the upload status display, the shared variables keeping track of the upload progress must be initialized with this function.

Parameters:

srecLen number of SREC lines being uploaded

Xuint32 srec_installStatusHook ()

This is a wrapper function that installs the code upload status display and removes it as soon as the upload is complete.

Bug

Sometimes the display is not shown. Until now I was not able to track down the exact problem.

Returns:

0

Xuint32 srec_loadingStatus (Xuint32 **vpos***)*

This graphics hook-up function is used to display the progress of the code upload currently being performed. It can be added to the pool of graphics functions handled by the `gfx_graphicManager()`.

Parameters:

vpos the vertical position of the display. If equal to `GFX_GET_SIZE` defined in `graphix.h`, this means that someone wants to know the number of lines needed for this graphics element

Returns:

0 in normal mode, the needed number of lines if asked for

8.2.27 srec.h File Reference

8.2.27.1 Detailed Description

This is the header file for `srec.h`.

Author:

Samuel Nobs

Date:

2004-04-05

Revision

1.4

Functions

- int decode_srec_line (char *buffer)
  Decode SREC Line.
- int decode_srec_line8 (Xuint8 *buffer, Xuint32 len)
  Decode SREC Line from UDP.
- Xuint32 srec_installStatusHook ()
  Install Status Display.
- Xuint32 srec_loadingStatus (Xuint32 vpos)
  Display Status.
- void srec_initForDisplay (Xuint32 srecLen)
  Initialize for Display.

8.2.28 user.c File Reference

8.2.28.1 Detailed Description

This file mainly implements shell commands and display hook-up functions. It represents the basics for user interaction in combination with the command user interface defined in `cui.h`. The strings to start the various shell commands are given in the descriptions, however, these strings are defined in the command lists in `cui.c`.

Author:
Samuel Nobs

Date:
2004-04-05
Revision
1.14

Functions

- Xuint32 ps ()
  Process Status (shell command).
- Xuint32 setquanta (char *argv[2], Xuint32 argc)
  Set Number of Quanta (shell command).
- Xuint32 kill (char *argv[1], Xuint32 argc)
  Kill Process (shell command).
- Xuint32 context (char *argv[1], Xuint32 argc)
  Print Process Context (shell command).
- Xuint32 setcolors (char *argv[3], Xuint32 argc)
  Setup Colors (shell command).
- Xuint8 temperature (Xuint8 vpos)
  Temperature Display.

- Xuint32 inetd (char ∗argv[1], Xuint32 argc)
  Ethernet Daemon (shell command).
- Xuint8 ETHStatusDisplay (Xuint8 vpos)
  Ethernet Status Display.
- Xuint32 config (char ∗argv[2], Xuint32 argc)
  Configure R-FPGA (shell command).
- Xuint32 bitslist ()
  List of Bitstreams (shell command).
- Xuint32 printScreen ()
  Print Screen.

8.2.28.2 Function Documentation

Xuint32 bitslist ()

This command prints a list of the bitstreams currently present in the memory. The information displayed for each bitstream is the following:

- bitstream id
- name of the file that contained the bitstream on the host computer
- type of the bitstream: full / partial
- span of task slots occupied by this bitstream
- memory location of the bitstream
- length of the bitstream (in bytes)

The command is started by simply entering `bitslist` in the shell.

Returns:
0

Xuint32 config (char ∗ argv[2], Xuint32 argc)

This command can be used to configure the R-FPGA with a bitstream that has been previously loaded to the external memory or to erase the configuration currently present. Additionally, the command can be used to read the status register and the configuration options register of the SelectMAP port of the R-FPGA. However, these register functions are only included in the code when the `CONFIG_DEBUG` symbol is defined at compile time.

```
Usage: config <BID|OPTION>
       BID:    bitstream id (number)
```

```
OPTION: -r,--erase:      erase R-FPGA
        -s,--getStatus:  get Status Register
        -c,--getConfOpt: get Configuration Options
```

Parameters:

argv pointer to the arguments list

argc number of arguments

Returns:

0

Xuint32 context (char* ∗ **argv[1], Xuint32* **argc***)*

This command is a frontend wrapper to `sch_context(Xuint32)` defined in `scheduler.c` and is used to dump the register context of the process associated with the PID entered as an argument.

```
usage: context <pid>
```

Parameters:

argv pointer to the arguments list

argc number of arguments

Returns:

0

Xuint8 ETHStatusDisplay (Xuint8* **vpos)*

This graphics hook-up function displays basic information about the status of the C-FPGA's ethernet interface and the daemon. The hook-up can be added to the pool of graphics functions handled by the `gfx_graphicManager()`. It displays the following information:

- link state: up / down
- full / half duplex
- 10Mbit/s / 100Mbit/s
- number of received and transmitted packets

Parameters:

vpos the vertical position of the display. If equal to `GFX_GET_SIZE` defined in `graphix.h`, this means that someone wants to know the number of lines needed for this graphics element

Returns:

0 in normal mode, the needed number of lines if asked for

Xuint32 inetd (char * argv[1], Xuint32 argc)

Although defined as a shell command, this function does not normally need to be started from the shell as it gets launched automatically during system bringup. However, it sometimes might be unavoidable to kill the process and start it again from the shell.

This function waits for a packet and calls the packet analyzer as soon as a packet is available. If `NET_SPAWN_SEPARATE_TASKS` is defined, the packet analyzer is started as a separate process, otherwise it is started as a regular subroutine call.

When the daemon is started, it installs an instance of the `ETHStatusDisplay` in the graphics manager.

If the function is called with $argc > 0$, it returns the number of packets received so far.

Parameters:

argv pointer to the arguments list

argc number of arguments

Returns:

0

Xuint32 kill (char * argv[1], Xuint32 argc)

This command is basically a frontend wrapper to `sch_kill(Xuint32)` defined in `scheduler.c` and is used to kill a process with a given PID.

```
usage: kill <pid>
```

Parameters:

argv pointer to the arguments list

argc number of arguments

Returns:

0

Xuint32 printScreen ()

This function is used to dump the content of the text memory displayed on the screen via the RS-232 interface. The function is attached to the print screen button on the keyboard.

Returns:

0

Xuint32 ps ()

This command prints the status of all processes currently in the list `TaskList`. It can be started without additional arguments in the shell using the string `ps`. The following information is printend:

- the name of the process
- its process ID
- the number of scheduler time quanta assigned to this process
- the low and the high address of the stack memory
- the status of the process
- if the process suspending the shell or not

For a description of the process statuses have a look at the documentation of `scheduler.h`.

Returns:
0

Xuint32 setcolors (char* ∗ **argv**[3], Xuint32* **argc**)

This command is a frontend wrapper to `vga_setupColors(Xuint32,Xuint32,Xuint32)` and is used to setup the color configuration of the vga display. Using this command, the colors for the primary and secondary text and for the background can be set. The code for this function is not included in the core code of the OS. It must be compiled into the user program by giving the option `-DEXTERNAL_OS_CODE` to the compiler.

```
usage: setcolors <textcol1> <textcol2> <backgroundcol>

       black:0  blue   :1  green: 2  cyan :3
       red  :4  magenta:5  yellow:6  white:7
```

Parameters:
argv pointer to the arguments list

argc number of arguments

Returns:
0

Xuint32 setquanta (char * argv[2], Xuint32 argc)

This shell command can be used to adjust the number of scheduler time quanta assigned to a process with a given PID.

```
usage: setquanta <pid> <quanta>
```

One time quantum corresponds to a time lapse of about 1.3 milliseconds.

Parameters:

argv pointer to the arguments list

argc number of arguments

Returns:

0

Xuint8 temperature (Xuint8 vpos)

This graphics hook-up function displays the junction temperature of both C-FPGA and R-FPGA. The hook-up can be added to the pool of graphics functions handled by the `gfx_graphicManager()`.

Bug

Sometimes both displays show a junction temperature of zero degrees or another bogus value. I currently assume that the problem might be found in the communication between the C-FPGA and the temperature ADCs. Normally, this effect vanishes after an additional reset.

Parameters:

vpos the vertical position of the display. If equal to `GFX_GET_SIZE` defined in `graphix.h`, this means that someone wants to know the number of lines needed for this graphics element

Returns:

0 in normal mode, the needed number of lines if asked for

8.2.29 user.h File Reference

8.2.29.1 Detailed Description

This is the header file for user.c.

Author:

Samuel Nobs

Date:

2004-04-05

Revision:

1.1

Functions

- Xuint32 bitslist ()
  List of Bitstreams (shell command).
- Xuint32 **config** ()
- Xuint32 **context** ()
- Xuint32 **crashme** ()
- Xuint32 inetd (char ∗argv[1], Xuint32 argc)
  Ethernet Daemon (shell command).
- Xuint32 **kill** ()
- Xuint32 **malloctest** ()
- Xuint32 **ping** (char ∗argv[2], Xuint32 argc)
- Xuint32 ps ()
  Process Status (shell command).
- Xuint32 **ram** ()
- Xuint32 printScreen ()
  Print Screen.
- Xuint32 **setcolors** ()
- Xuint32 **setquanta** ()
- Xuint32 **uartd** ()
- Xuint8 ETHStatusDisplay (Xuint8 vpos)
  Ethernet Status Display.
- Xuint8 temperature (Xuint8 vpos)
  Temperature Display.

8.2.30 util.c File Reference

8.2.30.1 Detailed Description

This file contains commonly used utility functions.

Author:
Samuel Nobs

Date:
2004-04-05

Revision:
1.1

Functions

- unsigned char nybble_to_val (char x)
  Convert Nybble to Value.
- int grab_hex_byte (char ∗buffer)

Get Hexadecimal Byte.

- unsigned int grab_hex_word (char *buffer)
  Get Hexadecimal Word.
- unsigned int grab_hex_dword (char *buffer)
  Get Hexadecimal Word.
- Xuint32 parseInteger (char *str)
  Get Integer from String.
- char asc (Xuint8 n)
  Return Character.

8.2.30.2 Function Documentation

char asc (Xuint8* n*)

Returns printable ASCII representation of a number. If the charater is not printable, replaces it with a dot.

Parameters:
n the number to convert

Returns:
the character

int grab_hex_byte (char * buffer)

Grabs a hexadecimal byte from a string and returns its integer value.

Parameters:
buffer the string to grab the byte from

Returns:
the integer value found in the string

unsigned int grab_hex_dword (char * buffer)

Grabs a hexadecimal doubleword from a string and returns its integer value.

Parameters:
buffer the string to grab the doubleword from

Returns:
the integer value found in the string

unsigned int grab_hex_word (char * buffer)

Grabs a hexadecimal word from a string and returns its integer value.

Parameters:
buffer the string to grab the word from

Returns:
the integer value found in the string

unsigned char nybble_to_val (char x)

Convert a hex digit to a value.

Parameters:
x the hex digit to convert

Returns:
the conversion result

Xuint32 parseInteger (char * str)

Converts a number given as a string into an integer value.

Parameters:
str string to convert into integer number

Returns:
the integer found in the string

8.2.31 util.h File Reference

8.2.31.1 Detailed Description

This is the header file for `util.c` .

Author:
Samuel Nobs

Date:
2004-04-05

Revision:
1.1

Functions

- unsigned char nybble_to_val (char x)
  Convert Nybble to Value.
- int grab_hex_byte (char *buffer)
  Get Hexadecimal Byte.
- unsigned int grab_hex_word (char *buffer)
  Get Hexadecimal Word.
- unsigned int grab_hex_dword (char *buffer)
  Get Hexadecimal Word.
- void **fill_hex_dword** (char *str)
- void **sleep** (Xuint32 n)
- char asc (Xuint8 n)
  Return Character.
- Xuint32 parseInteger (char *str)
  Get Integer from String.

8.2.32 vga.c File Reference

8.2.32.1 Detailed Description

This file provides functions that represent the software part of the VGA (XGA) driver.

Author:
Samuel Nobs

Date:
2004-02-02
Revision
1.14

Functions

- void vga_outbyte (char c, Xuint8 col)
  Put byte on screen.
- void vga_updatecursor ()
  Update cursor position.
- void vga_clearline (Xboolean keepPosition, Xuint8 col)
  Clear line.
- void vga_newline ()
  New line.
- void vga_init ()
  Initialize VGA driver.
- Xuint32 vga_cls ()

Clear screen.

- void vga_cgfx ()
  Clear graphics column.
- void vga_setupColors (Xuint32 textcol1, Xuint32 textcol2, Xuint32 bgcol)
  Set up color configuration.
- void vga_tab ()
  Tabulator.
- void vga_backspace ()
  Backspace.
- void vga_printstr (char *text, Xuint8 col)
  Print unformatted string.
- void vga_cursorfwd ()
  Move cursor forward.
- void vga_cursorbwd ()
  Move cursor backward.
- void vga_setdeletestop ()
  Set delete stop marker.
- void vga_todeletestop ()
  Jump to the delete stop marker.
- void vga_putc (unsigned char c, Xboolean updateCursor, Xuint8 column)
  Put character on screen.
- void vga_setColor (int col)
  Set text color.
- void vga_padding (const int l_flag, Xuint8 col)
  Pad output.
- void vga_outs (char *lp, Xuint8 col)
  Move string to output buffer.
- void vga_outnum (Xuint32 numl, Xuint32 base, Xuint8 col)
  Move number to output buffer.
- int vga_getnum (char **linep)
  Get number from format string.
- void vga_setlocation (Xuint32 xPos, Xuint32 yPos)
  Set insertion point.
- void vga_printf (Xuint8 col, const char *ctrl1,...)
  Print formatted string.

Variables

- volatile Xuint32 * vga_textMem [2]
  Two pointers to the VGA driver's text memory: one for the text column, one for the graphics column.
- Xuint32 vga_currLine
  Line the text column's insertion point currently resides at.
- Xuint32 vga_txtMemLineOffs = 0
  Offset by which the text column gets scrolled.
- Xboolean vga_incLineOffs = XFALSE

Determines whether to increment the scroll offset or not.

- Xuint32 vga_colorMask
  Determines whether to use the primary or the secondary text color.
- Xuint32 * vga_deleteStop
  Position up to which text can be deleted using the backspace key.
- int vga_doPadding
  Determines whether to pad numbers printed to screen or not.
- int vga_leftFlag
  A flag used in context of padding.
- int vga_len
  Used in context of padding: containts the length of a number as string.
- int vga_num1
  Temporary variable used by various functions.
- int vga_num2
  Temporary variable used by various functions.
- char vga_padChar
  The character used for padding.

8.2.32.2 Function Documentation

void vga_backspace ()

When a backspace character is found in a string to be printed, this function is called. The character to the left of the insertion point is deleted, then the insertion point moves back one character. If the cursor is at the beginning of the line, is moved to the end of the previous line.

void vga_cgfx ()

Clears the graphics column's text memory without affecting the text column.

void vga_clearline (Xboolean* **keepPosition,** *Xuint8* **col)

Clears a line by filling it with 0 starting at the actual text insertion position until the end of the line. An additional argument tells whether to keep the actual text insertion position or to move it to the beginning of the next line.

Parameters:

keepPosition preserve actual text insertion point

col the screen column:

- VGA_TXT for the text column
- VGA_GFX for the graphics column

Xuint32 vga_cls ()

This function clears the text memory of the text and the graphics column and resets all scrolling information. The text insertion point and the cursor are moved to the upper left corner of the text column.

Returns:
0

void vga_cursorbwd ()

Moves the cursor backward.

void vga_cursorfwd ()

Moves the cursor an forward.

int vga_getnum (char ∗∗ linep)

This routine gets a number from the format string. It is a helper function for `vga_printf`(Xuint8, const char∗, ...).

Parameters:
linep handle to the format string

Returns:
number parsed from the format string

void vga_init ()

Initializes the VGA driver by clearing the text memory using `vga_cls()` and resetting the `vga_colorMask` to its default value 0, meaning that the primary text color will be used. Also, the color configuration is set to its default state, i.e white as the primary text color, red as secondary text color, black as the background color. This function should be called before using any other VGA driver function.

void vga_newline ()

Performs a line break in the text column, including the handling of the scroll feature.

void vga_outbyte (char c, Xuint8 col)

Function used by `vga_printf`(Xuint8, const char∗, ...) to print a byte to the screen by simply forwarding the arguments to `vga_putc(unsigned char, Xboolean, Xuint8)`.

Parameters:

c the character / byte to print

col the screen column:

- VGA_TXT for the text column
- VGA_GFX for the graphics column

void vga_outnum (Xuint32 **numl***, Xuint32* **base***, Xuint8* **col***)*

This routine moves a number to the output buffer as directed by the padding and positioning flags vga_len and vga_leftFlag. It is a helper function for vga_printf(Xuint8, const char∗, ...).

Parameters:

numl number to output

base base of the number system: 2 to 16

col the screen column:

- VGA_TXT for the text column
- VGA_GFX for the graphics column

void vga_outs (char ∗ **lp***, Xuint8* **col***)*

This routine moves a string to the output buffer as directed by the padding and positioning flags, vga_len and vga_leftFlag. It is a helper function for vga_printf(Xuint8, const char∗, ...).

Parameters:

lp pointer to the string to pad

col the screen column:

- VGA_TXT for the text column
- VGA_GFX for the graphics column

void vga_padding (const int **l_flag***, Xuint8* **col***)*

This routine puts pad characters into the output buffer. It is a helper function for vga_-printf(Xuint8, const char∗, ...).

Parameters:

l_flag if > 0, padding is performed

col the screen column:

- VGA_TXT for the text column
- VGA_GFX for the graphics column

void vga_printf (Xuint8* col, *const char* * ctrl1, ...)

This routine operates just like a printf/sprintf routine without the overhead most run-time libraries involve.. It outputs a set of data under the control of a formatting string. Not all of the standard C format control are supported. The ones provided are primarily those needed for embedded systems work. Primarily the floaing point routines are omitted. Other formats could be added easily by following the examples shown for the supported formats.

Supported format modifiers:

- **%c** prints a single character
- **%s** prints a string
- **%d** prints a decimal number
- **%x** prints a hexadecimal number
- **%h** prints following characters with secondary text color
- **%n** prints following characters with primary text color

If a number is found between the percent sign and the x or d modifier, this number is interpreted as the length the printed number will be padded to using white spaces. If a zero is found between the percent sign and this number, 0's will be used for padding instead of the white spaces.

Parameters:

col the screen column:

- `VGA_TXT` for the text column
- `VGA_GFX` for the graphics column

ctrl1 pointer to the format string

... variable number of arguments

void vga_printstr (char* * text, *Xuint8* col)

Prints an unformatted string to the screen. Currently, line wrapping is not correctly handled, so this function is recommended to be used in the graphics column only. If you want to print a formatted string, you have to use `vga_printf`(Xuint8, const char*, ...) instead.

Parameters:

text pointer to the string to be printed

col the screen column:

- `VGA_TXT` for the text column
- `VGA_GFX` for the graphics column

void vga_putc (unsigned char* **c, *Xboolean* **updateCursor, *Xuint8* **column)

Prints a character on the screen, treating special characters like backspace, newline or tab. In the text column, line wrapping is treated correctly.

Parameters:

c character to print

updateCursor determines whether to advance the cursor or not

column the screen column:

- `VGA_TXT` for the text column
- `VGA_GFX` for the graphics column

void vga_setColor (int* **col)

Set either the primary or the secondary text color to be used.

Parameters:

col color to use for printing:

- 0: use primary text color
- 1: use secondary text color

void vga_setdeletestop ()

Marks the actual position as the current delete stop, i.e. forthcoming characters can be deleted using backspace up to this point. For example, this is used to prevent the `shell()` prompt from being deleted when a command is backspaced.

void vga_setlocation (Xuint32* **xPos, *Xuint32* yPos)

Set the coordinates of the text insertion point in the graphics column.

Parameters:

xPos x coordinate / line number, starting at 0

yPos y coordinate / character, starting at 0

void vga_setupColors (Xuint32* **textcol1, *Xuint32* textcol2, *Xuint32* bgcol)

Sets the colors for the primary and the secondary text and for the background.

Parameters:

textcol1 primary text color

textcol2 secondary text color

bgcol background color

void vga_tab ()

This function is called by `vga_printf`(Xuint8, const char∗, ...) to move the text insertion point to the next tab stop. There are no tab stops defined in the graphics column, so this function only applies for the text column.

void vga_todeletestop ()

Moves the cursor to the delete stop marker.

void vga_updatecursor ()

Updates the cursor position in the text column by writing the actual text insertion position to the cursor register located at `VGA_CUR_REG`. No cursor shall be displayed in the graphics column, that's why this function applies for the text column only.

8.2.33 vga.h File Reference

8.2.33.1 Detailed Description

This file is the header for keyboard.c .

Author:
Samuel Nobs

Date:
2004-02-02

Revision
1.9

Text Display Constants

- #define VGA_CHARS_PER_LINE 128
  Total characters per line.
- #define VGA_CHARS_IN_TEXT 64
  Characters per text column line.
- #define VGA_CHARS_IN_GRAPH 63
  Characters per graphics column line.
- #define VGA_NUMBER_OF_LINES 48
  Number of lines on display.

Color Constants

- #define VGA_BLACK 0
  Black color.
- #define VGA_RED 4
  Red color.
- #define VGA_GREEN 2
  Green color.
- #define VGA_BLUE 1
  Blue color.
- #define VGA_YELLOW (VGA_RED+VGA_GREEN)
  Yellow color.
- #define VGA_CYAN (VGA_GREEN+VGA_BLUE)
  Cyan color.
- #define VGA_MAGENTA (VGA_RED+VGA_BLUE)
  Magenta color.
- #define VGA_WHITE (VGA_RED+VGA_GREEN+VGA_BLUE)
  White color.

Text Memory Constants

- #define VGA_TEXT_MEM_BASEADDR XPAR_DLMB_VGA_BASEADDR
  Base address of the text memory.
- #define VGA_TEXT_MEM_HIGHADDR
  Highest address in the text memory, at the end of the graphics column.
- #define VGA_GRAPH_BASEADDR
  Base address of the text memory for the graphics column.

Register Constants

- #define VGA_COL_REG VGA_TEXT_MEM_HIGHADDR
  Register containing the color information.
- #define VGA_LIN_REG (VGA_TEXT_MEM_HIGHADDR+4)
  Register containing the line offset for scrolling.
- #define VGA_CUR_REG (VGA_TEXT_MEM_HIGHADDR+8)
  Register containing the cursor position.

Position Keeping Constants

- #define VGA_KEEP_POS XTRUE
  Preserve insertion point when clearing a line.
- #define VGA_CHANGE_POS XFALSE
  Update insertion point after clearing a line.

Display Columns

- #define VGA_TXT 0
  Text column.
- #define VGA_GFX 1
  Graphics column.

Printing Macros

- #define printf(format,) vga_printf(VGA_TXT,format, ## __VA_ARGS__)
  Formatted printing to the text column.
- #define gprintf(format,) vga_printf(VGA_GFX,format, ## __VA_ARGS__)
  Formatted printing to the graphics column.
- #define debug(format,)
  Formatted debug message, including file name and line number.
- #define gputc(c) vga_putc(c,XFALSE,VGA_GFX)
  Print a single character to the graphics column.
- #define **gset**(x, y) vga_setlocation(x,y)

Defines

- #define VGA_TABSIZE 16
  Tab stop spacing.

Functions

- Xuint32 vga_cls ()
  Clear screen.
- void vga_cgfx ()
  Clear graphics column.
- void vga_clearline (Xboolean keepPosition, Xuint8 col)
  Clear line.
- void vga_cursorbwd ()
  Move cursor backward.
- void vga_cursorfwd ()
  Move cursor forward.
- void vga_init ()
  Initialize VGA driver.
- void vga_printf (Xuint8 col, const char *ctrl1,...)
  Print formatted string.
- void vga_printstr (char *text, Xuint8 col)
  Print unformatted string.
- void vga_putc (unsigned char c, Xboolean updateCursor, Xuint8 column)
  Put character on screen.

- void vga_setColor (int col)
  Set text color.
- void vga_setdeletestop ()
  Set delete stop marker.
- void vga_setlocation (Xuint32 xPos, Xuint32 Ypos)
  Set insertion point.
- void vga_setupColors (Xuint32 textcol1, Xuint32 textcol2, Xuint32 bgcol)
  Set up color configuration.
- void vga_todeletestop ()
  Jump to the delete stop marker.

8.2.33.2 Define Documentation

#define debug(format)

Value:

```
vga_printf(VGA_TXT,"%s, %s(), line %d:",__FILE__,__func__,__LINE__);\
                                vga_printf(VGA_TXT,format, ## __VA_ARGS__)
```

#define VGA_GRAPH_BASEADDR

Value:

```
(XPAR_DLMB_VGA_BASEADDR + \
                                VGA_NUMBER_OF_LINES*VGA_CHARS_IN_TEXT*4)
```

#define VGA_TEXT_MEM_HIGHADDR

Value:

```
(XPAR_DLMB_VGA_BASEADDR + \
                                VGA_NUMBER_OF_LINES * \
                                (VGA_CHARS_IN_TEXT+VGA_CHARS_IN_GRAPH)*4)
```

8.2.34 xfinclude.h File Reference

8.2.34.1 Detailed Description

This file contains various global definitions and commonly used types.

Author:

Date:
2004-04-05
Revision:
1.1

Error Constants

- #define ERR_NONE 0
  No error.
- #define ERR_NO_FREE_TASKSLOT 0x00004000
  Scheduler error: no free task slot available.
- #define ERR_NO_FREE_GFXSLOT 0x00005000
  Graphic manager error: no free graphics slot.
- #define ERR_UNDEF 0xFFFFFFFF
  Undefined error.

Defines

- #define USE_ARG(n) n=n
  This macro can be used to avoid the unused arguments *compiler warnings.*

Typedefs

- typedef Xuint32 XFError
  Type for errors.
- typedef Xuint8 XFbool
  8 bit boolean
- typedef Xuint32(∗ FunctionPointer)(char ∗∗argv, Xuint32 argc)
  Type for functions used for shell commands.
- typedef Xuint32 ProcessStatus
  Type for process status.

Functions

- void microblaze_enable_interrupts (void)
  Enable interrupts.
- void microblaze_disable_interrupts (void)
  Disable interrupts.

8.3 XF-Board Operating System Page Documentation

8.3.1 Todo List

Global errorLMBInterrupt() A future version of this interrupt handler may send more detailed debugging information.

Global errorStackInterrupt() A future version of this interrupt handler may send more detailed debugging information.

Global msg_printMsg(msgStr) The current version of this function does not support formatted string. This feature may be enabled in a future version.

Global mmu_readFromFifo(pfdlPtr, first) Currently, this function changes the read pointer of the FiFo in the MMU. Its functionality could be extended by supporting a read mode leaving the read pointer untouched. This task can be accomplished by explicitly reading the memory at the location the FiFo resides at, but this approach is not too comfortable.

Global net_ipconfig(argv, argc) The configuration of R-FPGA's ethernet interface must be included.

Global net_removePortListener(pid) A function to remove the port listeners more selectively may be useful.

Global PacketAnalyzer() Additional protocols and packet types could be supported.

Global SendUDPTo(bytDestinationIP[4], bytDestinationMAC[6], intDestinationPort, ptrDataBuffer, DataLength) As the system does not have an ARP table, the full information about the communication partner needs to be known, i.e. the IP and MAC addresses. Additional effort could be invested to include such an ARP table to make sending of packets easier.

Global sch_addToTaskList(task, argv, argc, name, quanta, stackSize, suspendShell) This function is also used by processes to spawn child processes. The OS must be extended to do some bookkeeping about the dependencies between processes and their childs.

Global sch_kill(pid) This kill function does not yet check the process dependencies. It mus be extended to kill child processes of a process too. However, it is the OS that does not keep track of such dependencies either.

Global selmap_Configure(startadr, len, type) The approach to configure the R-FPGA by setting the SelectMAP data lines by software is rather slow. A DMA controller for this task is being developped. As soon as this controller is ready, this function must be adapted accordingly.

8.3.2 Deprecated List

Global SetCS(value) This function will be meaningless as soon as the DMA configuration controller will be included in the design.

8.3.3 Bug List

Global srec_installStatusHook() Sometimes the display is not shown. Until now I was not able to track down the exact problem.

Global temperature(vpos) Sometimes both displays show a junction temperature of zero degrees or another bogus value. I currently assume that the problem might be found in the communication between the C-FPGA and the temperature ADCs. Normally, this effect vanishes after an additional reset.

```
Clock 0:  20.833 MHz, on     Clock 2:  20.833 MHz, on
Clock 1:  20.833 MHz, on     Clock 3:  20.833 MHz, on
```

Figure 8-1: *Clock Display*

Figure 8-2: *FiFo Fill-Level Graphics Element*

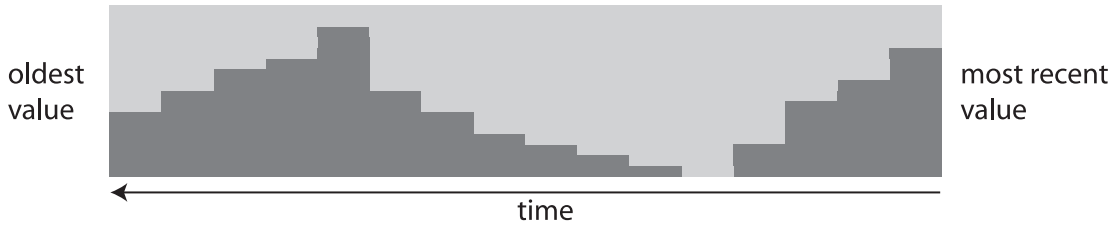


Figure 8-3: *Vertical History Bargraph Graphics Element*

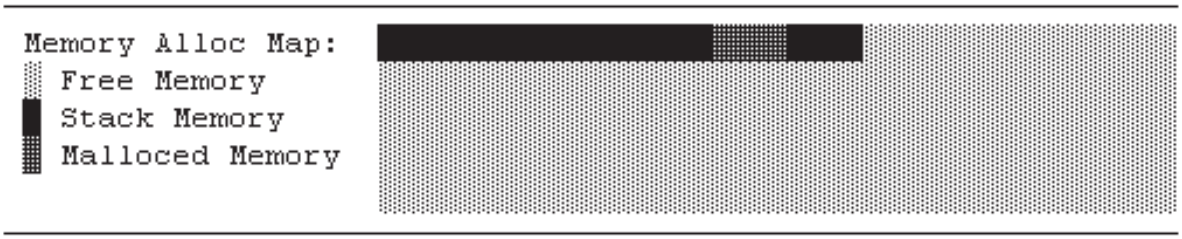


Figure 8-4: *Example Display of the Memory Allocation Map*

R-FPGA usage:

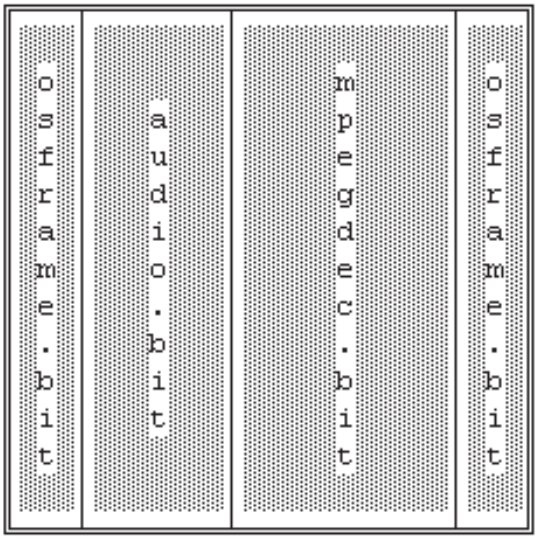


Figure 8-5: *R-FPGA Occupancy Display*

Ethernet : Link:UP, Full Duplex, 10Mbps, RX:1565, TX:1563

Figure 8-6: *Ethernet Status Display*

Temperature: C-FPGA: 42°C, R-FPGA: 32°C

Figure 8-7: *Temperature Display*

9 Skill Forwarding

This chapter is meant as a collection of HOW-TOs to allow for a quick development start. For most of the processes there exit lengthy explanations, either in documents provided by XILINX or in other sections in this report. The most important points of these documents are resumed here.

9.1 How to Build the System

This section shall guide you through the basic steps to get the XFBOARD OS up and running.

To build the XFBOARD OS hardware and software under Windows[1], copy the directory `CD/Projects/RHWOS_v1/` from the CD. Be sure to place this directory at a location with no spaces in the path name. Copying the directory to your Desktop is a bad idea, as the path to the Desktop contains spaces: `C:/Documents and Settings/you/Desktop`. This condition should be met because some of the XILINX programs don't work when they see spaces in the path.

The contents of this directory, the project directory, are listed in figure 9-1.

Now, verify that the correct version of the XILINX software is present on your system: XILINX Embedded Development Kit 6.1.2, Platform Studio Version 6.1.03i. Newer versions may also work. If the software is not present as needed, install or update it.

Check that the system environment variable %XILINX_EDK% is present and points to the location where EDK indeed is installed. Also check that %XILINX_EDK% is included in the %PATH% variable.

The ATX power supply, the JTAG download cable, a VGA monitor capable to display 1024 $\times$ 768 pixels at least, and a PS/2 keyboard (preferably with US American layout) must be connected. The

[1] unfortunately, no other operating system is currently supported. The XILINX software is available for Unix and Linux too, but with these systems, the configuration via JTAG is not supported.

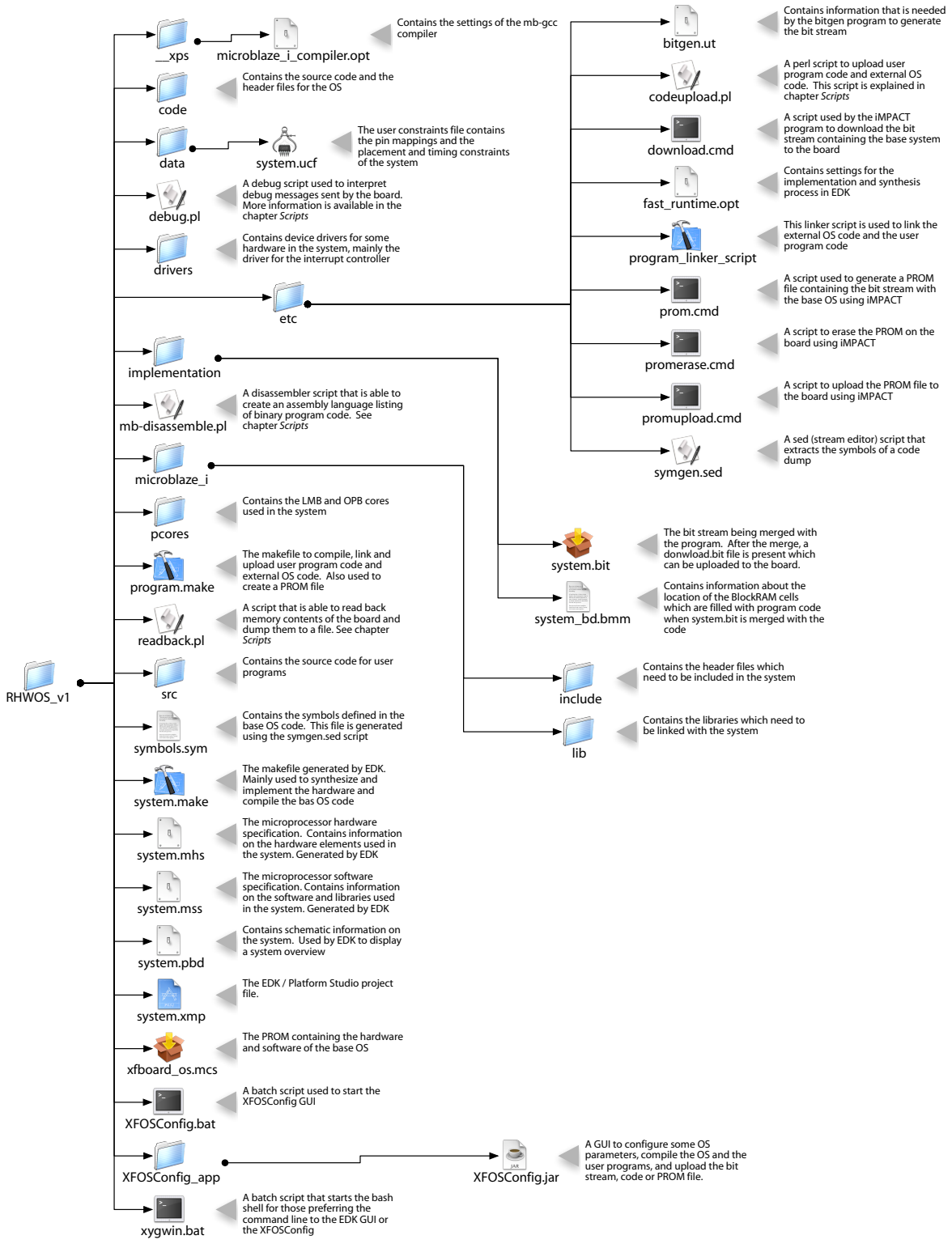


Figure 9-1: System Directory Overview. *This figure provides an overview of the directory containing the XFBOARD OS including a short description of the files and folders.*

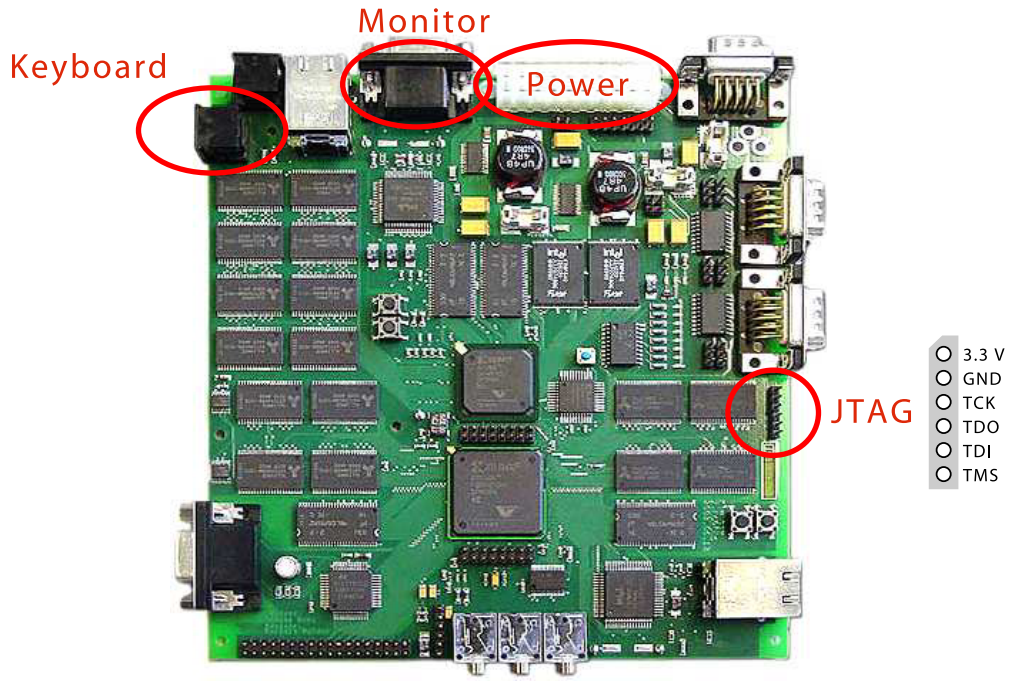


***Figure 9-2: XF-Board Basic Connections**. To run the XFBOARD OS, the ATX power supply, a VGA monitor with a minimal resolution of 1024 × 768 pixels, a PS/2 keyboard and a JTAG download cable must be connected.*

locations of the connectors used are highlighted in figure 9-2.

Now that all preliminaries are met, the system is ready to be built using one of three approaches: the Platform Studio, from the command line using a makefile, and using a Java GUI.

Build Using Platform Studio Open the project `system.xmp` either by double-clicking the document in the explorer or using the menu ***file*** →***Open Project*** in Platform Studio. As soon as the project is open, choose ***Tool*** →***Download*** to build the system and upload it to the board. This approach is recommended for everyone new to the EDK flow.

Build Using the Command Line For advanced EDK users, a bash shell can be started by double-clicking the batch file `xygwin.bat`. In this shell, type `make -f system.make download` to build the system and upload it to the board.

Build Using the Java GUI A Java GUI to configure the OS and upload it to the board has been designed. Java must be installed on the host computer to run this GUI. The GUI can be started by

double-clicking the batch file `XFOSConfig.bat`. However, this GUI is in beta state and not fully tested. Refer to appendix D for details about this GUI.

After having successfully built and uploaded the OS, your VGA monitor connected to the board should look like figure 9-3.

***Figure 9-3: Screen After Startup**. This figure is a screenshot of the monitor connected to the XFBOARD. Your screen should look similar after sucessfully building and uploading the OS.*

9.2 How to Write User Functions

This section shall offer the knowledge needed to write your own C functions for the XFBOARD OS and add them as commands to the shell. It does not explain how the code is compiled and linked, because this topic is offered a separate chapter in appendix B.

There is no special rule on how to write functions to be run on the XFBOARD except for the functions that need to be accessed from within the shell. Functions to be installed into the shell are expected to have the following signature:

```
1  Xuint32 functionName(char** argv, Xuint32 argc);
```

The argument `argc` will contain the number of arguments entered in the shell, `argv` will hold the list of arguments entered in the shell. `argc` and `argv` can be omitted if the function does not take any arguments.

To install this function in the shell, the OS needs to be given some additional information. A string is needed which will be entered in the shell to start the function. Then, a pointer is needed that holds the

address of the start of the function. The stack size of the function must be defined, as the scheduler will try to allocate stack memory for the function when it is started as a process (section 4.2.1.2). The number of time quanta must be included in the information on the function too. Finally, a flag must be set if the function is able to suspend the shell to get keyboard input.

Be sure to correctly calculate the amount of memory needed for the stack when implementing your own processes. If these numbers are not appropriate, the system might kill your process as described in the next section! This warning is already mentioned in section 5.2.2. However, due to its high importance, it may be a good idea to repeat it here.

The information on the functions to be added to the shell needs to be entered in a list using a predefined macro, `XF_COMMAND_LIST`. The constant `XF_NUMBER_OF_COMMANDS` needs to be defined to the number of functions added to the shell. An example for one such function is given in the following code snippet, which can be found in the file `CD/Projects/RHWOS_v1/src/system.c` on the CD:

```
1  #include <xbasic_types.h>
2  #include "scheduler.h"
3  #include "vga.h"
4
5  Xuint32 helloWorld(char** argv,Xuint32 argc);
6
7  // this is how to add your functions to the command user interface
8  #define XF_NUMBER_OF_COMMANDS 1
9  XF_COMMAND_LIST =
10   {
11    {"hello",                // string to be typed in the shell
12     (FunctionPointer)&helloWorld, // pointer to the function to be executed
13     512,                    // minimal stack size needed by function (in bytes)
14     SCH_DEFAULT_QUANTA,     // number of scheduler time quanta
15     XFALSE                  // xtrue if command suspends shell, xfalse otherwise
16    }
17   };
18
19 Xuint32 helloWorld(char** argv,Xuint32 argc){
20   Xuint32 i;
21   printf("hello %hworld%n!\n");
22   printf("you entered %d arguments\n",argc);
23   for(i=0;i<argc;i++){
24     printf("argv[%d] = %s\n",i,argv[i]);
25   }
26   return 0;
27 }
```

This represents a *hello world* program which can be started from the shell using the string `hello`. Everything that is entered after the string but before hitting the *return* key is interpreted as arguments to the function. These arguments are then dumped to the screen, one by one.

For the strings to be started the commands with, be sure that they are unique in the system. No built-in command should be present that is started with the same string, as your function will never be called; built-in commands are searched first when a string is entered in the shell. Also be sure that your commands are sorted alphabetically to allow the command completion mechanism (section 6.1.1) to work properly

9.3 How to Write an OPB Core

This section shall give you the hints needed to create your own OPB core in a minimum of time. This explanation shall help you to navigate around all the pitfalls I fell in. XILINX offer a tutorial on writing custom OPB cores [26], but I still want to offer an explanation in my own words, being able to emphasize the points I found to be important.

9.3.1 VHDL Module

Let's assume that you want to create a core called `corename` that connects to the OPB and has one bidirectional, 32 bits wide port and an additional output (see figure 9-4). When a read access is performed on this core, data are read from this bidirectional port. When a write access is performed, data are written to this bidirectional port, and the additional output is pulled to 1.

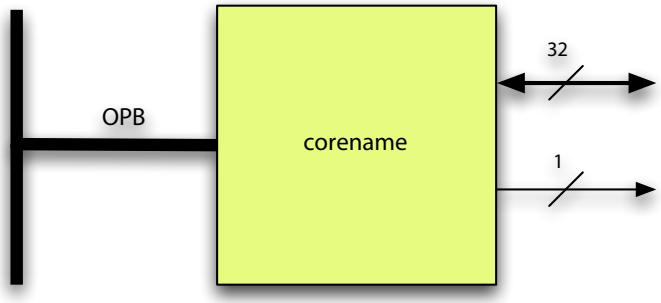


***Figure 9-4: Example OPB Core**. This figure schematically depicts the OPB core to be designed in this section. The core connects to the OPB and has a 32 bits wide bidirectional port and another 1-bit output.*

In addition to the mandatory generic parameters `C_BASEADDR` and `C_HIGHADDR`, this core uses another parameter, `C_INVERT`, which decides whether the data are written inverted or regular. These generic parameters are resolved at synthesis time and are therefore fixed in the implemented design.

C_BASEADDR and C_HIGHADDR denote the address range in which the core shall respond to accesses. This means, that address decoding logic is required. XILINX offers a helper module which includes this decoding logic[31], the IPIF core, but as this logic also includes many other features which are not needed for simple OPB cores, it is far too complex. Also, it infers an additional latency of two clock cycles for every read and write access. Therefore I suggest to implement the decode logic using compare functions provided by VHDL. These comparisons do not result in a large hardware overhead as long as you spend some seconds on finding good choices for the base and the high address.

An example: if you want to design a core that needs 2^n addresses, the base address of this core should have the n least significant bits to zero. The address relative to the base address then consists of theses n least significant bits; stripping these n bits does not require a lot of hardware resources.

A VHDL file that performs all tasks of such a core is given in the following code listing. However, as only one OPB address is used, the decoding is trivial.

```
1  library IEEE;
2  use IEEE.numeric_std.all;
3  use IEEE.std_logic_1164.all;
4  use IEEE.std_logic_misc.all;
5  library Unisim;
6  use Unisim.all;
7
8  entity corename is
9    generic
10     (
11       -- the standards for every OPB core
12       C_BASEADDR : std_logic_vector(0 to 31) := X"FFFFFFFF";
13       C_HIGHADDR : std_logic_vector(0 to 31) := X"00000000";
14       -- an additional generic parameter
15       C_INVERT : boolean           := true
16       );
17   port (
18     --required OPB bus ports, do not add to or delete
19     OPB_ABus    : in std_logic_vector(0 to 31); -- OPB address bus
20     OPB_BE      : in std_logic_vector(0 to 3); -- OPB byte enable
21     OPB_Clk     : in std_logic; -- OPB clock
22     OPB_DBus    : in std_logic_vector(0 to 31); -- OPB data input
23     OPB_RNW     : in std_logic; -- OPB direction, read /not write
24     OPB_Rst     : in std_logic; -- OPB reset
25     OPB_select  : in std_logic; -- OPB bus select
26     OPB_seqAddr : in std_logic; -- OPB sequential address
27     Sln_DBus    : out std_logic_vector(0 to 31); -- core data output
28     Sln_errAck  : out std_logic; -- core acknowledge
29     Sln_retry   : out std_logic; -- core retry
30     Sln_toutSup : out std_logic; -- core timeout suppress
31     Sln_xferAck : out std_logic; -- core transaction acknowledge
32     --user Ports
33     -- bidirectional data signal . the buffer is then inserted by EDK
34     DataxDIO_i : in std_logic_vector(31 downto 0);
35     DataxDIO_o : out std_logic_vector(31 downto 0);
36     DataxDIO_t : out std_logic;
```

```
37     -- regular output
38     WriteEnablexEO : out std_logic
39     );
40 end corename;
41
42
43 architecture rtl of corename is
44 begin -- rtl
45
46   read_or_write : process (DataxDIO_i, OPB_ABus, OPB_DBus, OPB_RNW, OPB_select)
47   begin
48     -- defaults
49     Sln_DBus <= (others => '0'); -- don't drive the bus
50     Sln_xferAck <= '0';          -- don't acknowledge
51     DataxDIO_t <= '1';           -- 1: read , 0:  write
52
53     -- nondefaults
54     if OPB_ABus = C_BASEADDR and OPB_select = '1' then
55       if OPB_RNW = '1' then      -- read
56         Sln_DBus <= DataxDIO_i; -- read data
57       else                       -- write
58         if C_INVERT = true then
59           DataxDIO_o <= not(OPB_DBus); -- write inverted data
60         else
61           DataxDIO_o <= OPB_DBus; -- write regular data
62         end if;
63         DataxDIO_t <= '0';       -- enable buffer  for  writing
64         WriteEnablexEO <= '1'; -- enable external component for writing
65       end if;
66       Sln_xferAck <= '1';        -- acknowledge the  transfer
67     end if;
68   end process read_or_write;
69
70   Sln_retry <= '0';              -- no retry
71   Sln_errAck <= '0';             -- no error
72   Sln_toutSup <= '0';            -- no timeout suppress
73
74 end rtl;
```

It possibly would be a better approach to buffer all input and output data signals using registers, but this would be a question of good coding style and not a question of how to create a minimal OPB core.

Some remarks on the purpose of the OPB signals: OPB_ABUS is the OPB address to which the request is made. OPB_BE is the byte enable signal that tells the core on which bytes of a 32 bit word the operation has to be performed. OPB_DBUS holds the data being written to the core. OPB_RNW is used to tell whether a read or write process is about to occur. OPB_SELECT is used to select the OPB bus. When the OPB bus is selected, every core attached to the bus is expected to check if an access to its address range is about to occur. These signals are valid and constant until SLN_XFERACK is seen high at a rising clock edge. If no such acknowledge is seen within 15 clock cycles, the bus times out. For write accesses, the acknowledge can be set as soon as the core has read the data from the bus. For read accesses, the acknowledge should be set as soon as the data are written to the bus by

the core. Data and acknowledge should stay high until the transaction is completed by the μBlaze by releasing OPB_SELECT.

Your core should under no circumstances assign values other than zero to the OPB bus when either the OPB address is not inside the range defined by C_BASEADDR and C_HIGHADDR or the bus is not currently selected (OPB_SELECT = 1) as this *will* prevent the system as a whole from working as expected!

If you intended to use sequential logic in your core, the OPB bus provides a clock signal (OPB_CLK) and an active high reset input (OPB_RST). Your core may also include additional VHDL modules.

The waveforms of an example read and write access are shown in figure 9-5.

9.3.2 Additional Files: `*.mhs` and `*.pao`

The `*.mhs` file is used to inform EDK about the ports available in this core and the generic parameters used. At the beginning of the file, declaration of the core's name and the bus interface is located. Then, a list of all parameters, their default values and the data types of these parameters is included:

```
BEGIN corename
OPTION  IPTYPE = PERIPHERAL
OPTION  IMP_NETLIST=TRUE

BUS_INTERFACE BUS = SOPB, BUS_STD = OPB, BUS_TYPE = SLAVE

## Generics for VHDL or Parameters for Verilog
PARAMETER c_baseaddr     = 0xFFFFFFFF, DT = std_logic_vector,\
                                       MIN_SIZE = 0x4
PARAMETER c_highaddr     = 0x00000000, DT = std_logic_vector
# --USER-- Add user core parameters
PARAMETER c_invert       = true,       DT = boolean
```

The line break in the declaration of `C_BASEADDR` is inserted for typographical reasons only, you should not insert such line breaks in the file. Now, a list containing all signals on the VHDL entity is to be supplied. Cores that connect to a signal already present in the system are assigned to this signal using the *equals* sign, i.e. the signals of the OPB bus. Signals that can be connected to a user defined system are assigned using " ". Then the direction of the signal must be given, and, if it's a vector, the range of the signal. Signals belonging to a special group of signals, such as the OPB signals, need to be defined accordingly:

```
## Ports
```

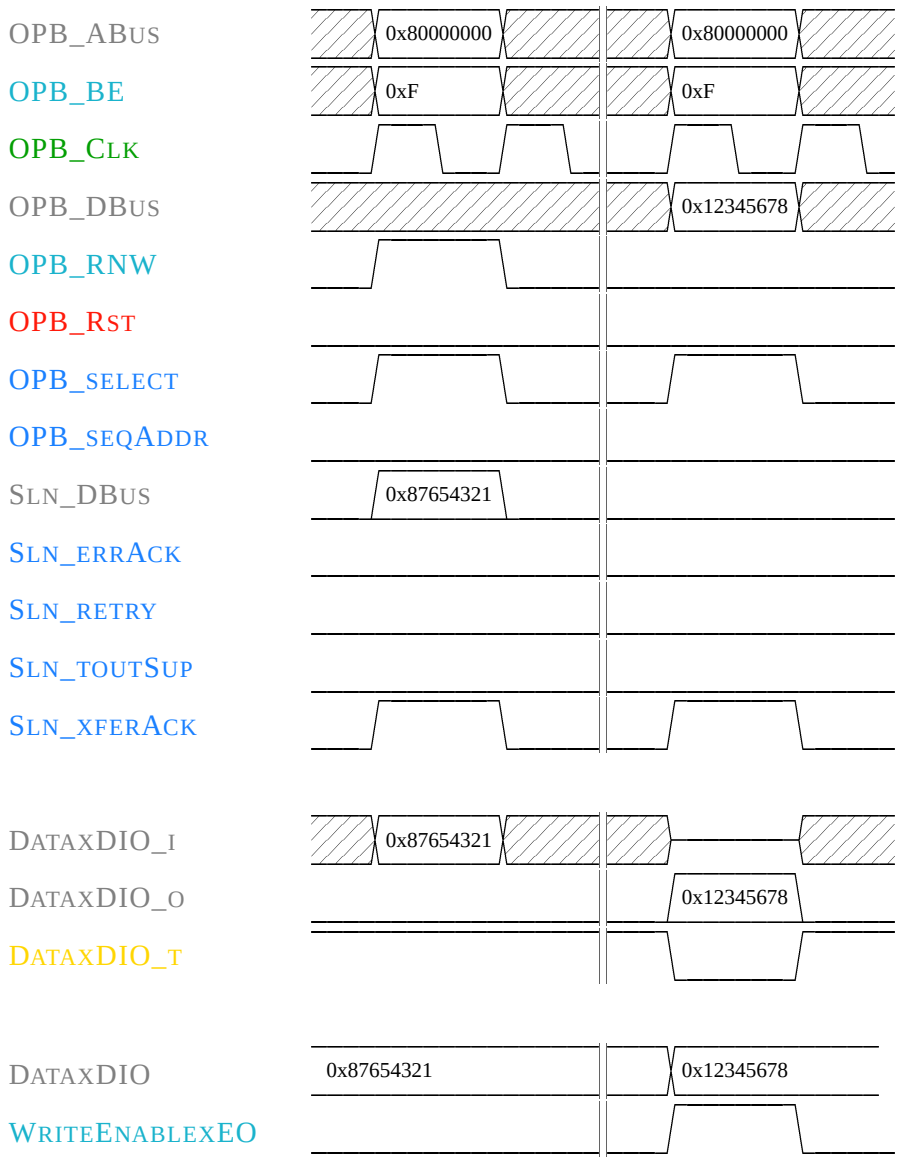


***Figure 9-5: OPB Core Example Waveforms**. This figure provides example waveforms for the `corename` OPB core for a read access (left) and a write access (right). The core is assumed to have a base address of `0x80000000`. `0x87654321` is read from the bus, `0x12345678` is written to the bus. The data being written are not inverted, i.e. the parameter `C_INVERT` is set to **false**. Most of the OPB signals are not used, but they are included in the diagram for sake of completeness and to eliminate possible questions on the values of these signals. Most of them are unlikely to be used in designs of moderate complexity. The only additional signal I ever used for the cores I designed was the byte enable.*

```
PORT opb_abus    = OPB_ABus,    DIR = IN, VEC = [0:31], BUS = SOPB
PORT opb_be      = OPB_BE,      DIR = IN, VEC = [0:3],  BUS = SOPB
PORT opb_clk     = "",          DIR = IN, SIGIS = CLK,  BUS = SOPB
PORT opb_dbus    = OPB_DBus,    DIR = IN, VEC = [0:31], BUS = SOPB
PORT opb_rnw     = OPB_RNW,     DIR = IN,               BUS = SOPB
PORT opb_rst     = OPB_Rst,     DIR = IN,               BUS = SOPB
PORT opb_select  = OPB_select,  DIR = IN,               BUS = SOPB
PORT opb_seqaddr = OPB_seqAddr, DIR = IN,               BUS = SOPB
PORT sln_dbus    = Sl_DBus,     DIR = OUT, VEC = [0:31],BUS = SOPB
PORT sln_errack  = Sl_errAck,   DIR = OUT,              BUS = SOPB
PORT sln_retry   = Sl_retry,    DIR = OUT,              BUS = SOPB
PORT sln_toutsup = Sl_toutSup,  DIR = OUT,              BUS = SOPB
PORT sln_xferack = Sl_xferAck,  DIR = OUT,              BUS = SOPB
# --USER-- change to user core ports
PORT DataxDIO       = "", DIR = INOUT, VEC = [31:0]
PORT WriteEnablexEO = "", DIR = OUT
END
```

The `*.pao` file is needed to inform EDK about the order in which the VHDL files are to be compiled and loaded. For every VHDL file, a line in the following form has to be included in the `*.pao` file.

```
lib corename_v1_00_a corename
```

Be sure to list your files in the correct order! First list the files representing your submodules, and at the end, the top file of your core must be listed.

As a general hint I recommend to copy the files of an existing core and adapt them to your design.

9.3.3 File Names and Directory Structure

As EDK is rather intolerant regarding file names and directory names, the directory structure and naming conventions expected by EDK are given here. An illustration of this structure can be found in figure 9-6. The base directory is named using the core's name and the version number (that you can freely assign) of this core. An example would be `corename_v1_00_a` designating a core named `corename` with version v1.00.a. This is the version number finally shown in the EDK. This naming convention is mandatory. This directory then contains 3 subdirectories:

doc holds the documentation on this core, which should be named `corename.pdf` to be accessible from within the EDK GUI.

hdl contains directories for the hardware description language used: `vhdl` for VHDL files, `verilog` for verilog files. Assuming a VHDL workflow, the `vhdl` directory is expected to hold a top level description of your core named `corename.vhd`. It this top architecture includes other entities, these may also be put in this directory.

data holds two files that are needed by EDK. The file `corename_v2_1_0.mpd` contains information on the input and output signals of the core, the `corename_v2_1_0.pao` contains information for the synthesis of the core.

The version number of the `*.mpd` and `*.pao` files is *always* 2.1.0, regardless of the version number of your core. The version number of these files designates the version of the parser tool used to read the files. This is not obvious at all, and it took me one afternoon to get behind it.

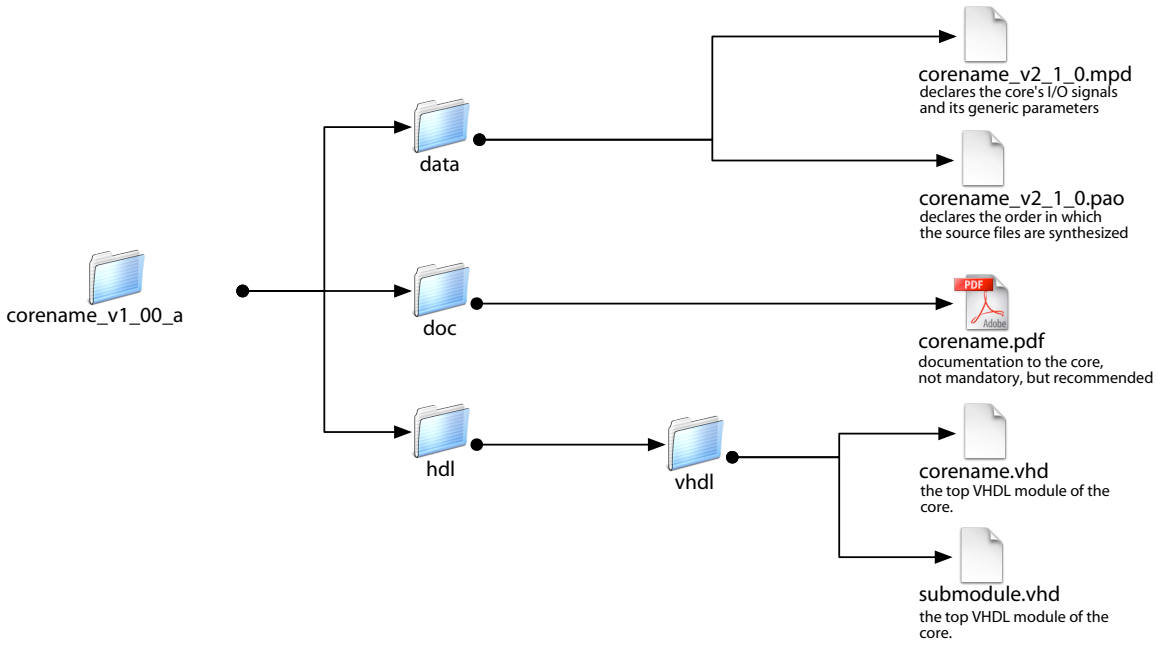


Figure 9-6: Directory Structure of OPB Cores. *This directory structure is mandatory when writing custom OPB cores. EDK will fail loading the core if you don't follow this structure, sometimes even without an error message!*

To include the core in your EDK design, create directory named `pcores` at the same level your EDK project `system.xmp` resides. Copy the directory `corename_v1_00_a` into `pcores`. Now, your OPB core should show up in the *Add/Edit Cores...* dialog. If your project is opened, it must be closed and opened again to make EDK search the `pcores` directory. When you change the contents of the `*.mpd` or `*.pao` files, you also need to close and reopen the project.

10 Outlook and Acknowledgements

In this short chapter, some final remarks are due. I want to give some hints for future improvements of the XFBOARD OS, tell what I learned during my work, and thank all those people that gave me a hand during this master's thesis.

10.1 Future Work and Improvements

Generally, some optimization steps could be performed. There exist elements in the system where a quick and straightforward approach has been used to get things working as soon as possible.

The **scheduler** could be analyzed for efficiency. Is a round-robin scheduler based on time slices appropriate? It possibly would be better to introduce process priorities to allow for real-time behaviour. The memory requirements of the process control blocks an therefore the process list as a whole are immense and could be optimized. Does each and every register need to be saved when task switches occur? Is there a better way to handle the arguments for the process than having them stored in the PCB? Instead, memory for the arguments could be dynamically allocated.

Memory protection could be improved, e.g. by making a difference between user process permissions and kernel process permissions. Some time should be spent on the FlashRAM on the board. This memory technology has not been used yet, but it would be great to store configuration data beyond power interruptions.

The **shell** could be improved regarding useability. Editing of the strings entered in the command line could be improved, possibly using shortcuts as known from other systems. The backspace key is not implemented correctly: when backspace is hit in the middle of a word, the cursor is moved left, deleting the character to the left, but the text to the right of the cursor is not moved left too. So editing is possible only in the *overwrite* mode.

The **OS bridge** could be optimized for speed as it currently runs with a lackadaisical timing. The effort

spent on the OS bridge will be very profitable as the OS bridge is the bottleneck of the communication between C-FPGA and R-FPGA.

Configuration of the R-FPGA could be sped up using a DMA controller; this DMA controller is currently being developed in [4].

The OPB and LMB cores for this system could be improved; most of them have a separate *outlook* section in their documentation in section 7. Refer to these section for details on what could be done better in future. A similar statement applies for the OS software functions: you may want to refer to section 8.3.1 on future work. The OS configuration GUI mentioned in appendix D is far from being finished and needs to be completed.

Special services for **dynamic partial reconfiguration** should be developed. Such services include a hardware scheduling scheme, a hardware resource manager, and a task preparation unit that saves and restores the context of hardware tasks.

Of course, **application** that fully use the features made available by the OS must be implemented.

10.2 Acknowledgements

First of all, I want to thank Herbert Walder, my advisor, for guiding me through this project. His ideas and his experience with reconfigurable OS and the corresponding hardware have been very helpful.

Many thanks go to other assistants that have been bothered with my concerns.

I want to thank the students working in the same room who helped in resolving minor and major problems and offered useful tips for working with the software environment. The *brain-pool* formed by the “inhabitants” of the ETZ G-69[13] room was amazing.

A thank you is aimed at the Support Group (Dienstgruppe) for helping with and installing the computer environment. Luckily, they perform a daily incremental backup of the home directories...

I wish to thank Prof. Dr. Lothar Thiele for being my supervisor and for his confidence in this project.

Appendix A

xfintc Driver

A.1 File Documentation

A.1.1 xfintc_l.c File Reference

A.1.1.1 Detailed Description

This file contains low-level driver functions that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. The contents of this file are derived from xintc_l.c by meinelte/Xilinx Inc.

Author:
Samuel Nobs

Date:
2004-02-18
Revision
1.2

Functions

- void xfintc_defaultHandler (void *unusedInput)
  Default interrupt handler.
- void xfintc_enableInterrupts (Xuint32 mask)
  Enable interrupts.
- Xuint32 xfintc_getEnabledInterrupts ()
  Get enabled interrupts.

A.1.1.2 Function Documentation

void xfintc_defaultHandler (void * unusedInput)

This function is a default interrupt handler for the low level driver of the interrupt controller. It allows the interrupt vector table to be initialized to this function so that unexpected interrupts don't result in a system crash.

Parameters:

unusedInput is an unused input that is necessary for this function to have the signature of an input handler.

void xfintc_enableInterrupts (Xuint32 mask)

This function can be used to enable / disable individual inputs. Calls the macro `XFINTC_SET_ENA_REG` defined in `xfintc_l.h`.

Parameters:

mask is a 32-bit mask that defines the interrupts being enabled / disabled

Xuint32 xfintc_getEnabledInterrupts ()

This function is used to check which interrupts are enabled. Calls the macro `XFINTC_GET_ENA_REG` defined in `xfintc_l.h`.

Returns:

a 32-bit mask denoting the interrupts being enabled / disabled

A.1.2 xfintc_l.h File Reference

A.1.2.1 Detailed Description

This header file contains identifiers and low-level driver functions (or macros) that can be used to access the device. The user should refer to the hardware device specification for more details of the device operation. The contents of this file are derived from xintc_l.h by meinelte/Xilinx Inc.

Author:

Samuel Nobs

Date:

2004-02-18

Revision

1.2

Data Structures

- struct XFVectorTableEntry
  Entry in the Interrupt Vector Table.

Defines

- #define **XINTC_L_H**
- #define XFINTC_INT_REG_OFFSET 0
  Address offset to access the interrupt register of the `opb_xfintc` *core.*
- #define XFINTC_ENA_REG_OFFSET 4
  Address offset to access the enable mask register of the `opb_xfintc` *core.*
- #define XFINTC_ACK_REG_OFFSET 8
  Address offset to access the acknowledge register of the `opb_xfintc` *core.*
- #define XIntc_In32 XIo_In32
  Wrapper macro for reading a 32 bit word, target is defined in `xio.h`.
- #define XIntc_Out32 XIo_Out32
  Wrapper macro for writing a 32 bit word, target is defined in `xio.h`.
- #define XFINTC_SET_ENA_REG(BaseAddress, EnableMask) XIntc_Out32((BaseAddress) + XFINTC_ENA_REG_OFFSET, (EnableMask))
  Enable specific interrupts in the interrupt controller.
- #define XFINTC_GET_ENA_REG(BaseAddress) XIntc_In32((BaseAddress) + XFINTC_ENA_REG_OFFSET)
  Get enabled interrupts in the interrupt controller.
- #define XFINTC_ACK_INT(BaseAddress, AckMask) XIntc_Out32((BaseAddress) + XFINTC_ACK_REG_OFFSET, (AckMask))
  Acknowledge interrupts in the interrupt controller.
- #define XFINTC_GET_INT_REG(BaseAddress) XIntc_In32((BaseAddress) + XFINTC_INT_REG_OFFSET) \
  Get asserted interrupts in the interrupt controller.

Functions

- void xfintc_defaultHandler (void *UnusedInput)
  Default interrupt handler.
- void xfintc_enableInterrupts (Xuint32 mask)
  Enable interrupts.
- Xuint32 xfintc_getEnabledInterrupts ()
  Get enabled interrupts.
- void xfintc_lowLevelInterruptHandler (void)
  Low level interrupt handler.

Variables

- XFVectorTableEntry xfintc_interruptVectorTable [ ]
  Declaration of the interrupt vector table.

A.1.2.2 Define Documentation

#define XFINTC_ACK_INT(BaseAddress, AckMask) XIntc_Out32((BaseAddress) + XFINTC_ACK_REG_OFFSET, (AckMask))

Acknowledge interrupts in the interrupt controller

Parameters:

BaseAddress is the base address of the device

AckMask is the 32-bit value to write to the acknowledge register. Each bit of the mask corresponds to an interrupt input signal that is being acknowledged using its acknowledge output signal if the bit for this acknowledge is set

Returns:

none

#define XFINTC_GET_ENA_REG(BaseAddress) XIntc_In32((BaseAddress) + XFINTC_ENA_REG_OFFSET)

Get enabled interrupts in the interrupt controller

Parameters:

BaseAddress is the base address of the device

Returns:

a 32-bit mask denoting the interrupts currently being enabled in the interrupt controller

#define XFINTC_GET_INT_REG(BaseAddress) XIntc_In32((BaseAddress) + XFINTC_INT_REG_OFFSET) \

Get asserted interrupts in the interrupt controller

Parameters:

BaseAddress is the base address of the device

Returns:

a 32-bit mask denoting the interrupts currently being asserted high at the input of the interrupt controller

#define XFINTC_SET_ENA_REG(BaseAddress, EnableMask) XIntc_Out32((BaseAddress) + XFINTC_ENA_REG_OFFSET, (EnableMask))

Enable specific interrupts in the interrupt controller

Parameters:

BaseAddress is the base address of the device

EnableMask is the 32-bit value to write to the enable register. Each bit of the mask corresponds to an interrupt input signal that is connected to the interrupt controller (INT0 = LSB). Only the bits which are set in the mask will enable interrupts

Returns:

none

A.1.3 xfintc_lg.c File Reference

A.1.3.1 Detailed Description

This file contains the generated configuration data for the low level driver of the interrupt controller. Basically, it is used as a template by `libgen` and gets populated with the interrupt vector table. The contents of this file are derived from `xintc_lg.c` by jhl/Xilinx Inc.

Author:

Samuel Nobs

Date:

2004-02-18

Revision

1.2

Variables

- XIntc_VectorTableEntry XIntc_InterruptVectorTable [XPAR_INTC_MAX_NUM_INTR_INPUTS]
  Declaration of the interrupt vector table.

A.1.4 xfintc_lowLevelHandler.c File Reference

A.1.4.1 Detailed Description

This file contains the low level interrupt handler.

Author:
Samuel Nobs

Date:
2004-02-18
Revision
1.2

Functions

- void xfintc_lowLevelInterruptHandler (void)
  Low level interrupt handler.

A.1.4.2 Function Documentation

void xfintc_lowLevelInterruptHandler (void)

This function is an interrupt handler for the low level driver of the interrupt controller. It must be connected to the interrupt source such that it is called when an interrupt of the interrupt controller is active. It will resolve which interrupts are active and call the appropriate interrupt handler, starting with the interrupt that is connected to the LSB input of the interrupt controller. It acknowledges the interrupt after it has been serviced by the interrupt handler.

This function assumes that an interrupt vector table has been previously initialized by `libgen`. It does not verify that entries in the table are valid before calling an interrupt handler.

A.2 Data Structure Documentation

A.2.1 XFVectorTableEntry Struct Reference

Data Fields

- XInterruptHandler handler
  Function pointer to the interrupt handler.
- void * callBackRef
  Pointer to the OPB address of the core that raises this interrupt.

Appendix B

User Code Generation

This chapter explains how user programs are merged with program code intended for external memory and then uploaded to the XFBOARD. All tasks are performed by the makefile `CD/Projects/RHWOS_v1/program.make` and the linker script `CD/Projects/RHWOS_v1/etc/program_linker_script`. Table B-1 contains a list of the targets available in this makefile. Following are the steps that are executed when the `upload` target is made, i.e. `make -f program.make upload` is entered in the command prompt of the host computer. You may want to refer to figure B-1 for an overview of this process.

First, the source files for the user programs and the source files of the OS code are compiled. From the source code for the OS, only the code sections within the `#ifdef EXTERNAL_OS_CODE ... #endif` pragma is being compiled as only this code is intended for use in external memory.

When adding your source files with user code to the system, be sure to edit the makefile accordingly. The `PROGRAM_SOURCES` variable should include all your source files. The files are separated by a blank. Also update the list of object files, `PROGRAM_O`.

Now the object files are linked with the libraries. As the user programs may use functions and variables that are defined in the code already present in the OS, these symbols are extracted from the OS code using the script `CD/Projects/RHWOS_v1/etc/symgen.sed` and written to a file which is linked with the user program code. During the linker process, the linker script `CD/Projects/RHWOS_v1/etc/program_linker_script` cares about the memory location of the various code sections. It generates a list of the user commands and the external OS commands at the beginning of the SRAM memory used for external program code (see figure 5-1). First, a magic keyword (`badger␣` plus string terminator) is written, then the number of user commands is appended, and then the lists of the external OS commands and the user commands is appended. Now the actual code and variables follow. Refer to section 9.2 on the structure of the list containing user commands.

target	description
`upload`	Uploads the user program code and the external OS program code. If this code is not present yet, it is generated first (→ `program`)
`upload_os`	Compiles and links and merges the code to the bit stream, which is then uploaded using impact. This is a standard EDK flow and is not explained any further
`prom`	Generates a PROM file from the OS bit stream. If the bit stream is not present, it is generated first
`upload_prom`	Uploads the OS PROM file. If it is not present, it is generated first (→ `prom`)
`erase_prom`	Erases the PROM on the XFBOARD
`program`	Compiles and links the source code for the user programs and the external OS code and then generates the according MOTOROLA SREC file
`programclean`	Cleans up by deleting the SREC file and all temporary and intermediate files related to the `program` target

Table B-1: Makefile Targets. *This table contains the main targets that this makefile is able to build. A target is built by entering* `make -f program.make <target>` *in the command prompt of the host computer. This table does not include all intermediate targets and does not claim to be an explanation on writing makefiles. [3] is an example for a good resource to get information on makefiles and the* `make` *program.*

The list with the external OS commands uses the same structure.

When the object files are linked into the `program.elf` file, this file gets dumped into a MOTOROLA SREC file (see figure B-2 for a description of the records in such a file). This file is then uploaded using the `CD/Projects/RHWOS_v1/etc/codeupload.pl` script.

To successfully upload the program code, be sure that the `BOARD_IP` variable in the makefile is set to the actual IP address of your board. Additionally, the `PERL` variable should be set to the path where your Perl program is installed.

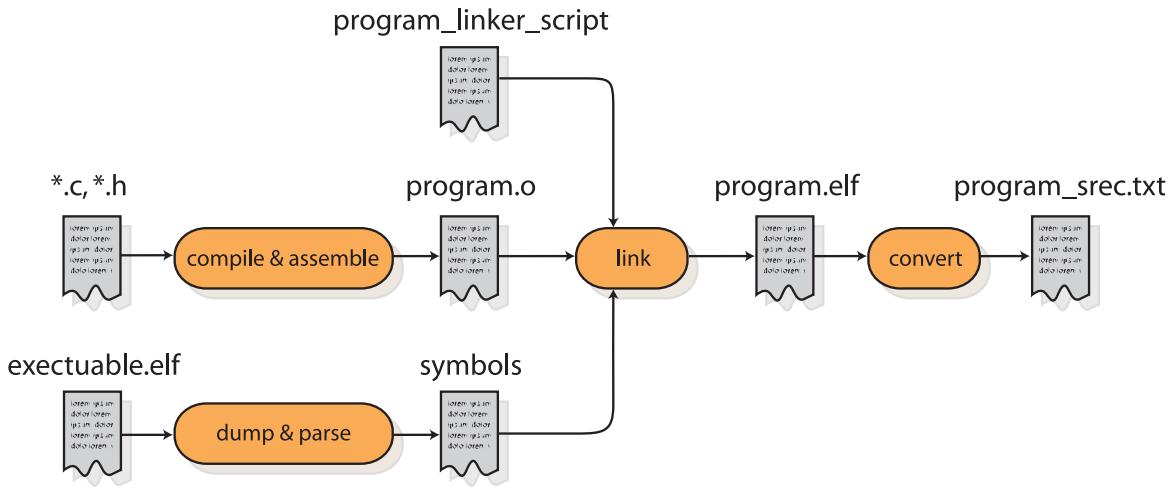


Figure B-1: User Code Generation. *This is a simplified visualization of the code generation flow performed by the makefile, depicting the steps from the C-sources to the final SREC file that may be uploaded to the* XF**BOARD**.

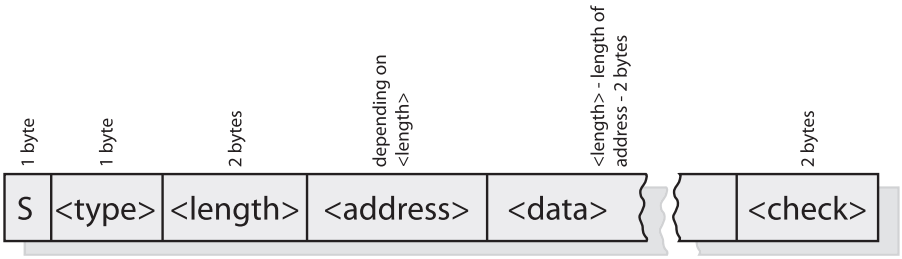


Figure B-2: SREC File Format. *An S-record line always starts with the character 'S', followed by a character '0'-'9' defining the type of the record. Then the record length is given in two hex characters. This length includes the following address, the data, and the 2-byte checksum at the end of the record. The length of the address depends on the type; the type being most interesting in our context is the type '3' which contains a 32-bit address encoded in 8 characters. The address specifies in which memory location the data has to be loaded into. The S-record line is terminated by 2 bytes representing the one's complement of the 8-bit checksum.*

Appendix C

Scripts

This section explains some scripts being useful when working with the XFBOARD OS. Some of them are in a very experimental state and mentioned only for sake of completeness. However, they are still a good starting point to write more sophisticated procedures.

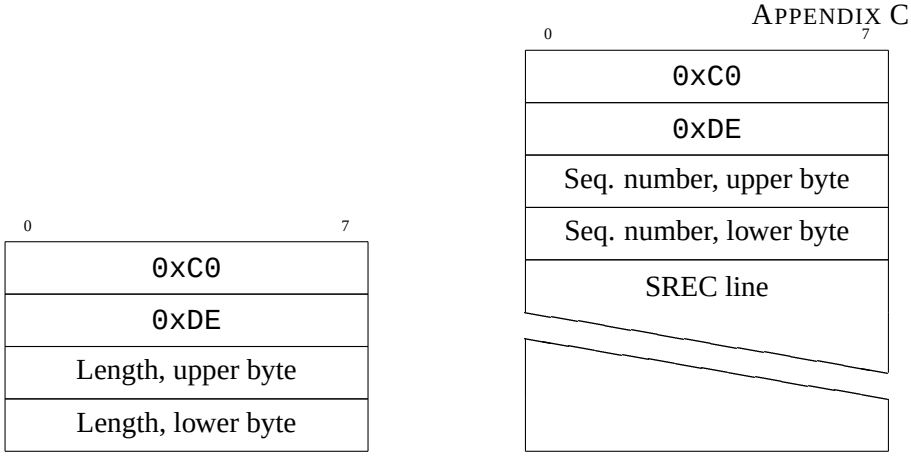
C.1 codeupload.pl

This Perl script, located at `CD/Projects/RHWOS_v1/etc/codeupload.pl`, is used to upload program code in MOTOROLA SREC format to the XFBOARD. It's synopsis is as follows:

```
perl codeupload.pl -a|--address|--host <BOARD_IP> <SREC_FILE>
```

If no IP for the board to send the code to is given, the default value will be used, which is `192.168.1.111`. The data are sent to port number `0xF0CE` of the board.

First, the length of the SREC code about to be transmitted is sent. The data in the in `SREC_FILE` are then sent line by line, each line forming a separate UDP packet. See figure C-1 for the format of these packets. These packets are assigned a sequence number which has to be sent back by the board to acknowledge the reception of the packet. When this acknowledge is not received within one second, the packet is sent again. After three such retries, the program dies. As soon as the acknowledge is received, the next packet is sent, containing the next SREC line. When all lines are sent, the program exits.



(a) Length Packet

(b) SREC Line Packet

Figure C-1: Packets Sent by `codeupload.pl`. *Displayed here are the payloads of the UDP packets sent by the script. First, the length of the SREC code to be transmitted is sent (a), then the code is sent line by line (b), alongside a sequence number which has to be sent back by the board.*

C.2 `debug.pl`

This Perl script, located at `CD/Projects/RHWOS_v1/debug.pl`, is used for debugging purposes. It acts as a daemon to receive debug messages from the XFBOARD. It is started by simply typing `debug.pl` and then waits for such debug messages.

Currently, debug messages are only sent when a process has been killed by the OS due to a memory access error. These debug messages are sent to port `0xFDE8` and contain the value of the program counter.

As soon as such a debug message is received, the program counter is read in. Then, the `*.elf` files of both the OS code and the user program code are dumped, parsed into HTML format and opened in a browser, currently MOZILLA.

Any browser that can be started from the command line, taking the file to be opened as an argument, could be used. The path to the browser is hard-coded; please adjust `$browserpath` to the one fitting your needs. I know that hard-coding variables is bad coding style, but this script is a dirty hack anyway...

The line corresponding to the position pointed at by the program counter is highlighted and scrolled to.

The script mainly consists of a huge number of regular expressions which highlight the assembly code to make it more readable and create links to the addresses in the code.

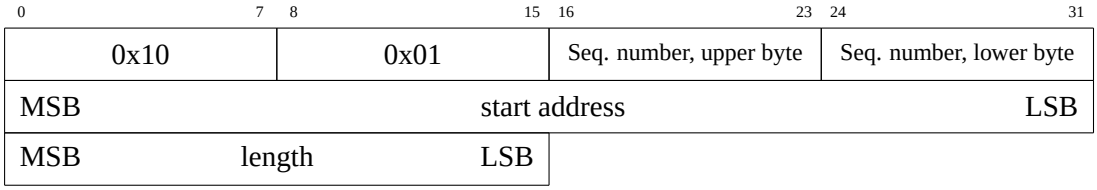


Figure C-2: Packets Sent by `readback.pl`. *The script sends UDP packets that start with `0x1001`, which is the command to read back memory contents. Next, the address where to start memory readback is sent, then the number of bytes to be read from memory is sent.*

C.3 `readback.pl`

This Perl script, located at `CD/Projects/RHWOS_v1/readback.pl` can be used to read memory contents back from the XFBOARD. Its synopsis is as follows:

```
perl readback.pl -a|--address|--host <BOARD_IP>
                 -s|--start <START_ADDR>
                 -l|--length <LENGTH>
                 -p|--packlen <PACKET_LENGTH>
                 <OUT_FILE>
```

`BOARD_IP` is the IP address of the XFBOARD, `START_ADDR` is the address in memory where to start the readback, `LENGTH` is the number of bytes to be read back, `PACKET_LENGTH` is the length of the packets the memory contents are sent in. `OUT_FILE` is the path to the file to be written and is the only mandatory argument. For the other arguments, the following defaults apply:

`BOARD_IP`:	192.168.1.111
`START_ADDR`:	0
`LENGTH`:	1000
`PACKET_LENGTH`:	10000

All numbers must be entered as decimal values.

The code is requested from the XFBOARD OS by sending a request packet to port `0xF0CE` and, as soon as it is received, written in binary format to the file `OUT_FILE`. The packet format of these requests is given in figure C-2.

Most often, program code is read back using this script. As binary program code is not very informative, the script described in section C.4 can be used to disassemble the code.

C.4 mb-disassemble.pl

As no utility was provided by XILINX that was able to dump binary program code into an assembly language listing and no such program was found on the Internet either, I decided to write such a tool by myself. The synopsis of this Perl script is as follows:

```
perl mb-disassemble.pl [-r|--include-raw]
                       [-l|--include-line-numbers]
                       <BIN_FILE>
```

The `-include-raw` switch tells the script to include the raw instructions in the dump, `-include-line-numbers` prepends the offset in the file to the lines. The disassembly is printed to standard output.

The disassembler supports all μBlaze instructions that are officially documented. It can be found at `CD/Projects/RHWOS_v1/mb-disassemble.pl` on the CD.

Appendix D

XF OS Configuration GUI

To easily configure the parameters of the XFBOARD OS, a Java GUI has been developed. Basic scheduler networking and scheduler parameters can be adjusted, and the command user interface can be customized to a certain degree. Apart from that, the following tasks can be started from within the GUI, which simply calls the according makefiles:

generate Config generates a filed named `config.h` placed in the `code` directory of the system. This file will be used to compile the system source files.

upload OS sends the base OS including the according hardware to the XFBOARD. If no such hardware is present, it is generated first.

upload PROM stores the PROM file in the on-board EEPROM. If no such PROM file exists, it is generated first.

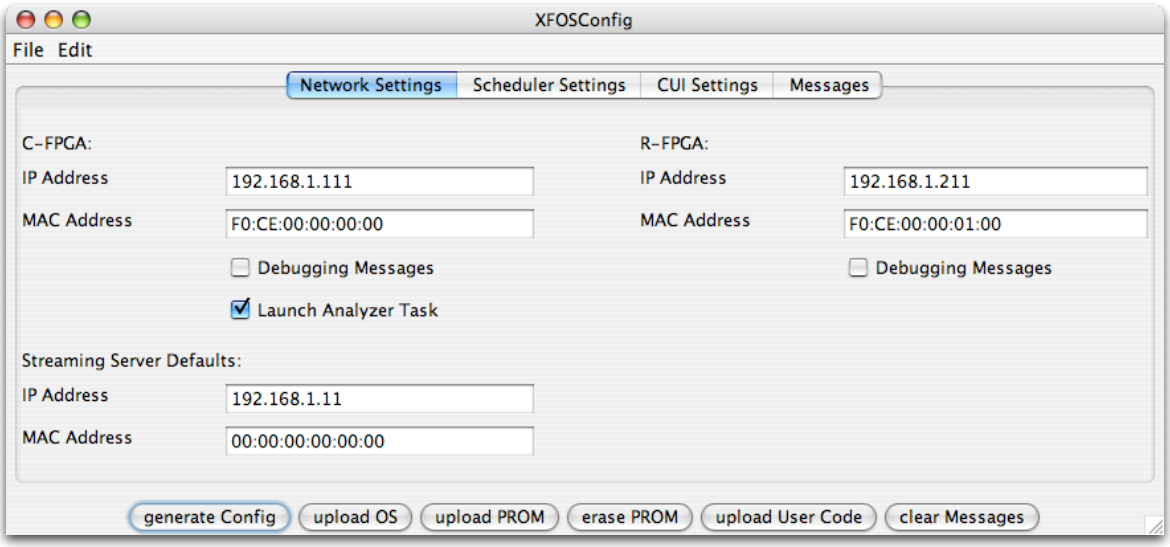
erase PROM is used to remove the PROM file currently present in the EEPROM.

upload User Code compiles and links the user code and the external OS code and then uploads it to the board.

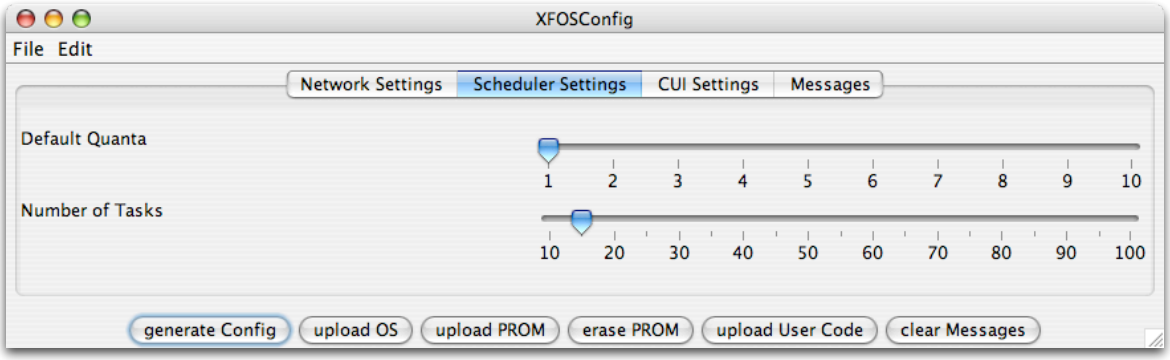
clear Messages erases all messages displayed in the messages tab.

The GUI is mentioned in the appendix only as it is far from being finished. Only a subset of the parameters can be adjusted, and the whole program is suspended when a background process is being executed. This means that no messages show up in the messages tab until a process has been completed. This problem could be solved by adding multithreading to the program.

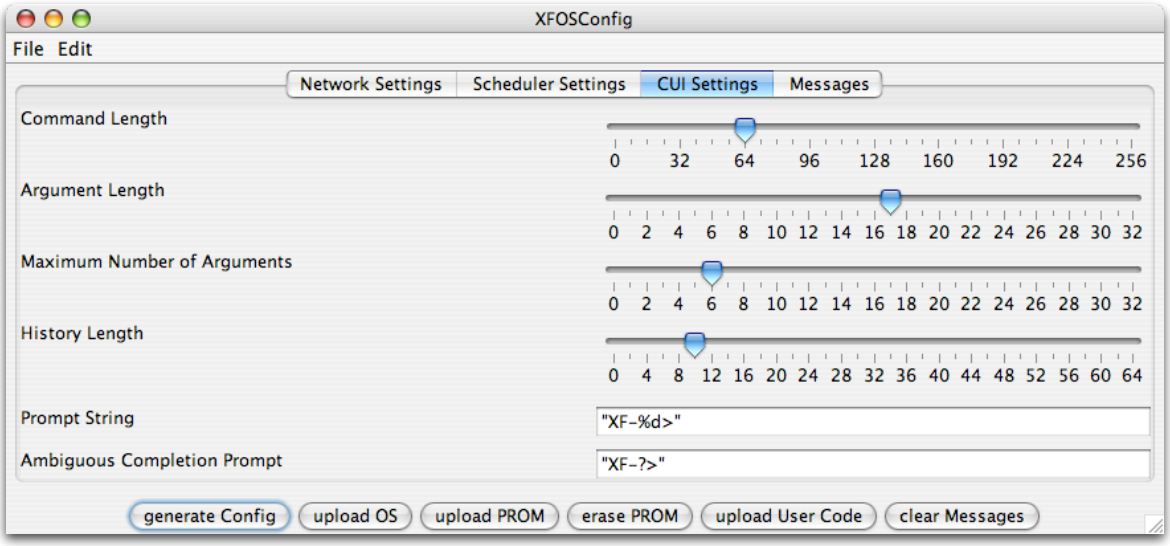
The sources of the GUI can be found on the CD: `CD/Projects/XFOSConfig/`.



(a) Network Tab



(b) Scheduler Tab



(c) User Interface Tab

Figure D-1: XF OS Configuration GUI*. Using this GUI, some key parameters of the OS can be adjusted. Build and upload processes can be started too using this GUI.*

Appendix E

R-FPGA MMU Draft

To manage the various memory technologies in the R-FPGA and to offer the hardware tasks services to access these memory modules as shared memory or FiFos, the need for a memory management unit (MMU) arose. I have combined the requirements for such a memory management unit and stated a draft on how to implement this MMU in hardware. This draft has been meant as a suggestion and has not been implemented by myself; the work has been delegated to Kristofer Jonsson who has been doing his master's thesis in the same project group. Due to the fact that I only suggested the draft and that it was sort of off-topic[1] in my thesis, I decided to put it into the appendix of this report. Overmore, the actual implementation may differ from the specifications stated in the draft.

Following is the paper I submitted to a team meeting we held on the MMU. The paper has been composed in note form, therefore I had to extend the formulations to include it in the report.

E.1 Available Resources

The MMU has to organize and manage the following memory technologies:

SRAM: 4 MB in total, organized as 2^{20} words $\cdot$ 16 bits $\cdot$ 2 banks

BlockRAM: 216 kB, organized as 2^{14} words $\cdot$ 16 bits

The SDRAM also being available has been neglected as it seems not being suitable for building fast FiFos due to its more complicated access protocol in column and row addresses.

[1] the MMU belongs to the OS elements on the R-FPGA

E.2 Basic Structure

To simplify the allocation and control hardware, a minimum and a maximum FiFo depth are defined. All Fifos can be a multiple of this minimum depth in size. This minimum depht is termed *block size*:

- minimum FiFo depth *block size*
  $2^8 \cdot 16 = 0.5$ kb
- maximum FiFo depth
  $2^{14} \cdot 16 = 32$ kb, which is enough for buffering audio data (2ch, 16 bit, 44.1 kHz) for 0.186 s

To organize the access to the FiFos, the connections between tasks and FiFos are listed in the VFDL (virtual FiFo description list) containing the following information:

TID (task ID)	TRFID (task-relative FiFo ID)	Direction	PFID (physical FiFo ID)
3 bits	3 bits	1 bit	4 bits

The task-relative FiFo ID is a virtual FiFo number in contrast to the physical FiFo ID. The width of the information in this table is based on the assumptions that a maximum of 5 tasks can be present at the same time and each task will use no more than 8 FiFos. 4 bits for the PFID results in a maxmimum of 16 FiFos being managed by the MMU.

In a second table, the MMU holds some details about the physically present FiFos:

FiFo ID	Type	BaseAddr	Size	Rd Ptr	Wr Ptr
4 bits	3 bits	12 bits	6 bits	14 bits	14 bits

The FiFos are implemented using circular memory. For every write access, a write pointer is being incremented. For every read access, a read pointer is being incremented. The FiFo is empty when the read pointer points to the same address as the write pointer. The FiFo is full when the write pointer incremented by 1 points to the same location as the read pointer.

3 bits for the type are needed as we have 5 types of FiFos:

- single-ported FiFo in SRAM bank 1
- single-ported FiFo in SRAM bank 2
- dual-ported FiFo in BlockRAM
- single-ported FiFo in BlockRAM accessible for read (directly accessed for write by a driver)
- sinlge-ported FiFo in BlockRAM accessible for write (directly accessed for read by a driver)

The 12 bit base address results from the difference between the SRAM address width 20 and the minimum block size of width 8. The size of the fifo can be described with 6 bits due to the difference of the maximum block size of width 14 and the minimum block size of width 8. The width of the read an write pointers equals the width of the maximum block size.

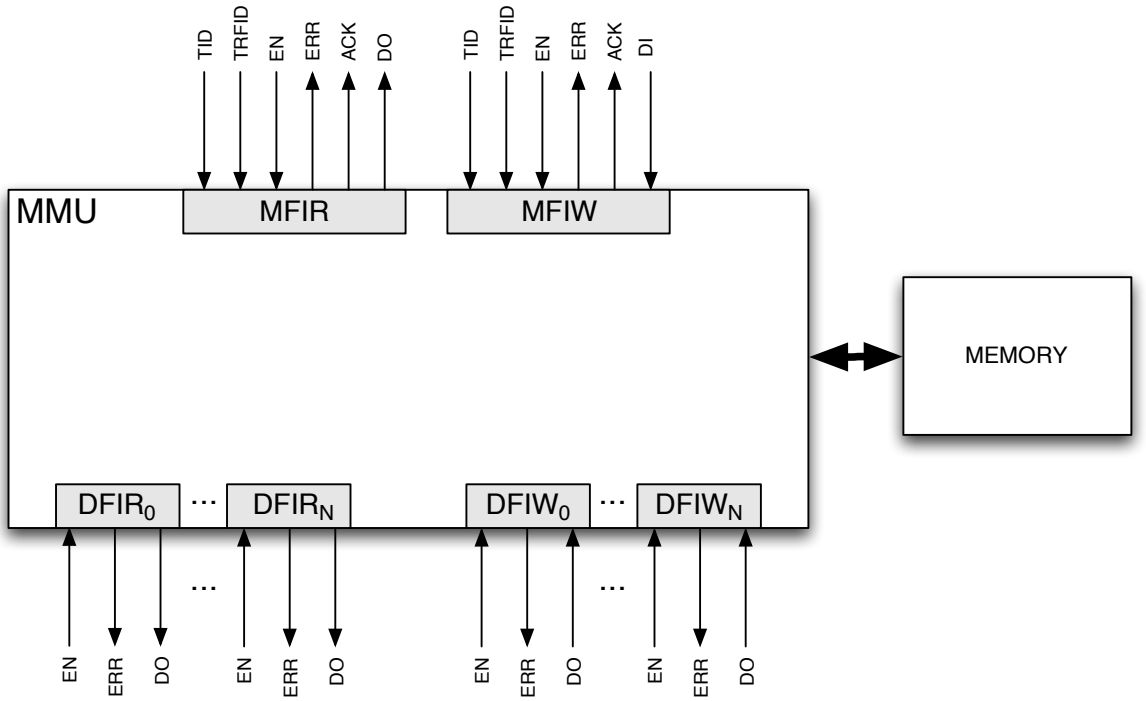


***Figure E-1: MMU Entity**. The ports of the MMU are displayed here. The MMU should provide access to the FiFo using a bus write interface (MFIW) and a bus read interface (MFIR). Accesses over these busses are time-multiplexed, so no two read or write accesses can occur at the same time. As there are device drivers which do not accept latencies when accessing a FiFo, direct read access FiFo interfaces (DFIR) and direct write access FiFo interfaces are dedicated to these drivers.*

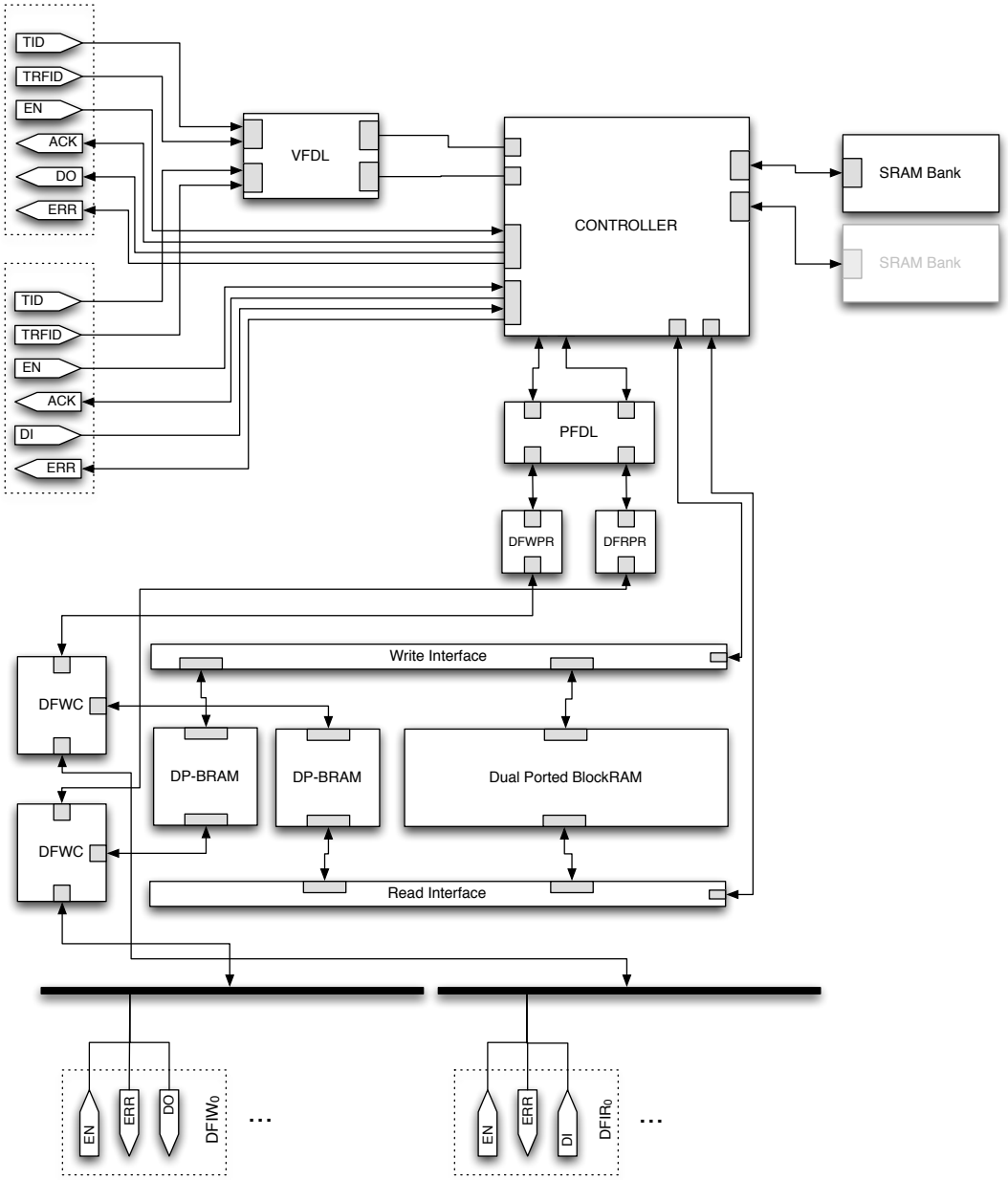


Figure E-2: MMU Block Diagram*. A view of the inside of the MMU is given here. The VFDL and the PFDL are implemented as look-up tables with read/write access. These look-up tables should be dual ported to allow for reasonable speed.*

DFWC: *direct FiFo write controller*	***DFRC:*** *direct FiFo read controller*
DFWPR: *direct FiFo write pointer register*	***DFRPR:*** *direct FiFo read pointer register*
VFDL: *virtual FiFo description list*	***PFDL:*** *physical FiFo description list*

Appendix F

VHDL Issues

F.1 Coding Guidelines

For VHDL coding I have used the coding guidelines of the Microelectronics Design Center of ETH. As a recapitulation I like to give a short overview over the main rules:

Constant Names

- Use *upper-case* letters and "_" only (e.g.,WIDTH, RAM_DEPTH, LFSR_INIT).
- Avoid "_" in *generics* (synthesis attaches generic names to other names with "_" as delimiter).

Signal Names

- Start with an *upper-case* letter.
- Have a *suffix* with syntax "x[CRESDTAZ][IO]?B?" ("[...]" denotes a choice, "?" means optional).
- The suffix part "[CRESDTAZ]" indicates the class of the signal:
- The suffix part "[IO]?" indicates *input* and *output* signals of an entity (e.g., COEFFXDI, FULLXSO)
- The suffix part "B?" indicates active *low* signals.

Class	Char	Example	Description
clock	C	CLKXC	clock
reset	R	RSTXRB	asynchronous reset
enable	E	LOADCNTXE, STARTCTRLXE	trigger some synchronous event
control/status	S	SELINPUTXS, FULLXS	static control signals, status signals
data/address	D	SAMPLEXD, RAMADRXD	data and address signals
test	T	SCANENXT, RAMISOLXT	test signals[a]
asynchronous	A		asynchronous signals
three-state	Z		three-state bus signals

Table F-1: Coding Style DZ for VHDL. *These are the suffixes suggested by the DZ to differentiate between the various signal types*

[a] in contrast to these guidelines, I have used the T suffix for tri-state control signals

Variable Names

- Start with a *lower-case* letter (e.g., temp, i, currentState).
- Have *no suffix* (as opposed to signal names).

Type Names

- Have a *suffix* "Type" or a name that implies a type (e.g. stateType, stdLogicArray).

FSM Names

- Have a *prefix* "st" and a name that implies the state(e.g. ,).

File Names

- Have the same name as the contained design unit (possibly with the first or all letters in lower case).
- Have file suffix ".vhd" or ".vhdl".

F.2 VHDL Error Hotlist

When intensely modelling circuits in VHDL, the following pitfalls are commonly encountered:

1. default assignments

   It must not be possible to go through a VHDL process statement without having all signals assigned a defined value. The most secure approach is to make a default assignment for all signals used at the beginning of the process.

2. reset polarity

   Some systems use active high reset signals, others use active low signals. If you don't respect the actual reset polarity, you may risk your system stuck in the reset state. And, *no*, this is not easily seen in the behaviour of the system!

3. sensitivity list

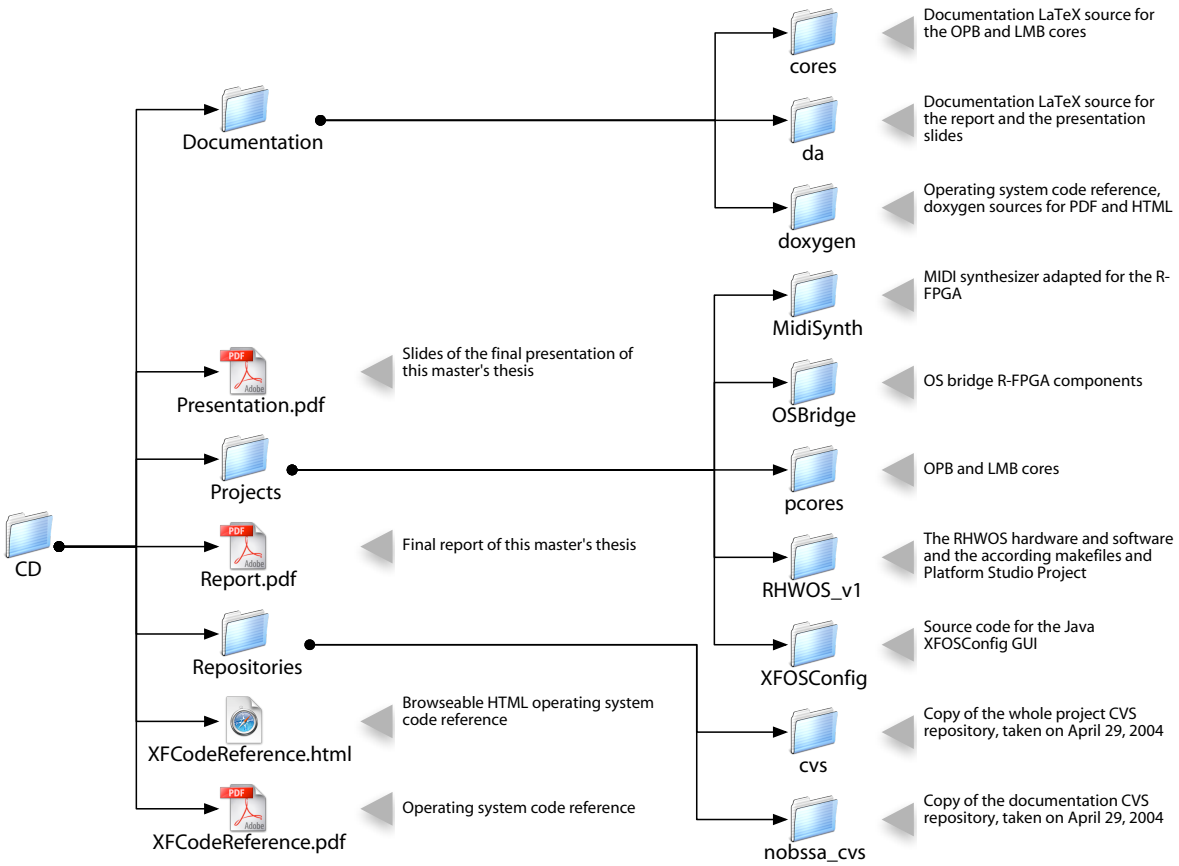
   The sensitivity list is a syntax element of minor relevance for the synthesis process. It is, however, very important for the simulation tool. The simulation can only be guaranteed to behave as the synthesized circuit does if attention is payed on correct sensitivity lists.

4. correct user constraints file (`*.ucf`)

   Having the wrong `*.ucf` file assigned to the project is fatal: often, no error message is spawned during the implementation process, but the design might still not work as the pins are not located as expected.

Appendix G

Contents of the CD



Appendix H

Glossary

ASIC Application-Specific Integrated Circuit. An integrated circuit with functionality customised for a particular use, rather than being for general-purpose use.

ATX An industry-wide specification for a desktop computer's motherboard. The standard also defines the connector for the power supply.

C-FPGA CPU FPGA. The FPGA on the board which is used as a CPU

CLB Configurable Logic Block. A block that can be used to implement some logical and state machine functionality. It consists of a number of look-up tables, multiplexers, and flip-flops. FPGAs are basically an array of CLBs.

CoDec Coder/Decoder. Device being DAC and ADC at a time.

CPLD Complex Programmable Logic Device. A combination of a fully programmable AND/OR array and a bank of macro-cells. CPLDs are non-volatile.

CPU Central Processing Unit. The central unit in a computer system containing the logic circuitry that performs the instructions of a computer's programs.

EDK Embedded Development Kit. A software environment provided by XILINX to develop embedded systems for their FPGA products.

FiFo First-In, First-Out. A list of data items implemented in a way that the oldest item is returned next when a read access occurs. A FiFo is often used as a buffer, e.g. to exchange data between tasks which process data at a different speed. FiFos can be implemented either in hardware or in software.

FlashRAM Flash Random Access Memory. Retains data bits in memory even when power is removed. It is organized so that a section of memory cells is erased in a single action or "flash".

The protocol to read and write data is rather complex. FlashRAM is slower and more expensive than SRAM.

FPGA Field-Programmable Gate Array. An integrated circuit that can be programmed in the field after manufacture. FPGAs are volatile.

GPIO General Purpose I/O. Inputs and outputs that can be used for arbitrary data transfer.

GUI Graphical User Interface. An interface used for human-computer interaction that is based on graphical rather than textual elements, i.e. buttons and menus.

ICAP Internal Configuration Access Port. An FPGA configuration method that is available to the internal logic resources after the device is configured. The ICAP block emulates the SelectMAP configuration protocol.

ISE Integrated Software Environment. A software environment provided by XILINX to synthesize and implement designs for their configurable logic devices (FPGAs, CPLDs). It also includes tools to perform timing analyses and to view the designs graphically.

ISEF Instruction Set Extension Fabric. A software-configurable data-path based on proprietary programmable logic defined by Stretch, Inc. Isef can be used to extend a processor instruction set and define new instructions using C/C++ code.

JTAG Joint Test Action Group. IEEE Standard 1149.1-1990 circuitry that may be built into an integrated circuit to assist in the test, maintenance, and support of assembled printed circuit boards. In the FPGA context, this circuit is used to download configuration data (in-system programming) and to perform debugging actions. The circuit is named after their developers.

LED Light Emitting Diode.

LMB Local Memory Bus. The bus used in μBlaze systems for fast access to on-chip BlockRAM.

MAT Memory Allocation Table. The table holding the information which blocks in the memory are currently being used for which purpose.

MMU Memory Management Unit. A hardware device performing the run-time mapping from virtual to physical addresses.

OPB IBM CoreConnect On-chip Peripheral Bus. A bus structure used to access on-chip peripherals such as memory controllers and I/O device drivers.

OS Operating System. The program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. These other programs, the applications, are offered services by the OS, e.g. management of memory and access to attached devices.

PCB Process Control Block. Representation of a process in the operating system. The PCB holds information about the process' state, register contents, scheduling information, I/O status information etc.

PROM Programmable Read-Only Memory. Memory that can be modified once by a user. As this leaves no margin for error, most PROM chips are so called EPROMS which are erasable and reprogrammable.

R-FPGA Reconfigurable FPGA. This FPGA is used as the reconfigurable resource in the system

RHWOS Reconfigurable Hardware Operating System. A specialised operating system (OS) that deals with the resource allocation and scheduling problems that come with a system involving reconfigurable hardware (FPGA's).

RISC Reduced Instruction Set Computer. A CPU that is designed to perform a smaller number of types of computer instruction so that it can operate at a higher speed. Since each instruction type that a computer must perform requires additional transistors and circuitry, a larger set of computer instructions tends to make the CPU more complicated and slower in operation.

SDRAM Synchronous Dynamic Random Access Memory. Retains data bits in memory as long as power is being supplied. Bits are stored in small capacitances. Due to charge loss in these capacitances, the cells need to be refreshed periodically. This memory technology is slower but cheaper than the SRAM technology. The protocol to access the data is more complicated than the one used for SRAMs.

SRAM Static Random Access Memory. Retains data bits in memory as long as power is being applied. Bits are stored in a register-like structure. Fast but expensive memory technology.

UART Universal Asynchronous Receiver/Transmitter. Controller of a computer's interface to its attached serial devices. Specifically, it provides the computer with the RS-232 interface so that it can "talk" to and exchange data with modems and other serial devices.

VGA Video Graphics Array. The accepted minimum standard for PC monitors. Introduced by IBM in 1987.

VHDL VHSIC Hardware Description Language. Design-entry language for FPGAs and ASICs in electronic design automation.

VHSIC Very-High-Speed Integrated Circuit. A type of very fast digital logic used for current designs.

Appendix I

Bibliography

[1] Microblaze uClinux. http://www.itee.uq.edu.au/~jwilliams/mblaze-uclinux.

[2] Alliance Semiconductor. *AS7C4096/AS7C34096, 5V/3.3V 52K x 8 CMOS SRAM*, v.1.8 edition, March 2002. http://www.alse.com/pdf/sram.pdf/fa/AS7C34096.pdf.

[3] Free Software Foundation, Inc. *GNU Make*. http://www.gnu.org/software/make/.

[4] Kristofer Jonsson. Component and Services for Reconfigurable OS. Master's thesis, ETH Zürich, May 2004.

[5] Marco Kuster. Firmware for Reconfigurable Hardware OS Platform. Master's thesis, ETH Zurich, Computer and Networks Lab, December 2003.

[6] Maxim Integrated Products. *Remote/Local Temperature Sensor with SMBus Serial Interface*, March 1998. http://pdfserv.maxim-ic.com/arpdf/MAX1617.pdf.

[7] Memec, Inc. http://www.memec.com.

[8] MIDI Manufacturers Association. MIDI Protocol Specification. http://www.midi.org/about-midi/specshome.shtml.

[9] Mind NV. http://mind.be.

[10] Samuel Nobs. Prototype Board for Reconfigurable OS. Term thesis, ETH Zurich, Computer and Networks Lab, July 2003.

[11] Samuel Nobs and Daniel Engeler. VLSI Design Project: MIDI Synthesizer. Term thesis, ETH Zurich, Integrated Systems Laboratory, February 2003.

[12] RedBoot. http://sources.redhat.com/redboot.

[13] Katja Reider and Silvio Neuendorf. *Wald-Detektiv Dario Dax*. Coppenrath Verlag, 2004.

[14] Michael Ruppen. Reconfigurable OS Prototype. Master's thesis, ETH Zurich, Computer and Networks Lab, 2003.

[15] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*, chapter 4 and 6. John Wiley & Sons, Inc., 1 edition, 2000.

[16] Abraham Silberschatz, Peter Galvin, and Greg Gagne. *Applied Operating System Concepts*, chapter 9. John Wiley & Sons, Inc., 1 edition, 2000.

[17] Christoph Steiger, Herbert Walder, and Marco Platzner. Heuristics for Online Scheduling Real-time Tasks to Partially Reconfigurable Devices. In *Proceedings of the 13rd International Conference on Field Programmable Logic and Application (FPL'03)*, pages 575–584. Springer, September 2003.

[18] Stretch, Inc. http://stretchinc.com/.

[19] Herbert Walder, Samuel Nobs, and Marco Platzner. XF-BOARD: A Prototyping Platform for Reconfigurable Hardware Operating Systems. Submitted to ERSA 04, 2004.

[20] Herbert Walder, Samuel Nobs, and Marco Platzner. XF-Board: Prototype Platform for Reconfigurable Hardware Operating System. Technical Report TIK Nr. 193, Swiss Federal Institute of Technology (ETH), Zurich, March 2004.

[21] Herbert Walder and Marco Platzner. Online Scheduling for Block-partitioned Reconfigurable Devices. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 290–295. IEEE Computer Society, March 2003.

[22] Herbert Walder and Marco Platzner. Reconfigurable Hardware Operating Systems: From Design Concepts to Realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287. CSREA Press, June 2003.

[23] Herbert Walder and Marco Platzner. Reconfigurable Hardware OS Prototype. Technical Report TIK Nr. 168, Swiss Federal Institute of Technology (ETH), Zurich, April 2003.

[24] Herbert Walder, Christoph Steiger, and Marco Platzner. Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS) / Reconfigurable Architectures Workshop (RAW)*, page 178. IEEE Computer Society, April 2003.

[25] Xess Corp. *XSV Board V1.1 Manual*, May 2001. http://www.xess.com/manual/xsv-manual-v1_1.pdf.

[26] Xilinx, Inc. *Designing Custom OPB Slave Peripherals for MicroBlaze*, February 2002. http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/opb_tutorial.pdf.

[27] Xilinx, Inc. *MicroBlaze Hardware Reference Guide*, March 2002. http://www.xilinx.com/ipcenter/processor_central/microblaze/doc/hwref.pdf.

[28] Xilinx, Inc. *LMB BlockRAM Interface Controller*, 1.5 edition, November 2003.

[29] Xilinx, Inc. *MicroBlaze Processor Reference Guide*, 6.1 edition, September 2003.

[30] Xilinx, Inc. *OPB Ethernet Lite Media Acess Controller*, 1.8 edition, November 2003.

[31] Xilinx, Inc. *OPB IPIF Architecture*, 1.3 edition, January 2003. http://www.xilinx.com/ipcenter/catalog/logicore/docs/opb_ipif.pdf.

[32] Xilinx, Inc. *Virtex-II Platform FPGA User Guide*, 1.8 edition, April 2004. http://www.xilinx.com/bvdocs/userguides/ug002.pdf.