Prototype design of an FPGA-based Ethernet server for STS detector readout for Compressed Baryonic Matter experiment at GSI facility

by

Pawel Kusmierski

A Thesis presented to the
Faculty of Physics, Astronomy and Applied Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science

JAGIELLONIAN UNIVERSITY
KRAKOW, POLAND

December 2009

Advisor: dr Adam Czermak
# Contents

1 Introduction ........................................... 1
   1.1 Objectives ........................................ 1
   1.2 Compressed Baryonic Matter experiment (CBM) .......... 2
   1.3 Silicon Tracking System (STS) ...................... 3
   1.4 n-XYTER ASIC ..................................... 4
   1.5 Virtex-4 FPGA chip architecture .................... 5
   1.6 Test board architecture ............................ 6

2 Principle of Operation ................................. 10
   2.1 Hardware ......................................... 10
   2.2 Software ......................................... 10

3 Tools .................................................. 13
   3.1 Xilinx ISE ......................................... 13
   3.2 Xilinx EDK ........................................ 15
   3.3 Cross compiler ..................................... 18
   3.4 Servers ............................................ 19

4 System Realization .................................... 21
   4.1 Linux PowerPC kernel ............................... 21
   4.2 Bootloaders ........................................ 28
   4.3 Operating system ................................... 37
   4.4 I²C bus ............................................. 41
   4.5 LabView integration .................................. 47

5 Conclusions ............................................ 51

A Files .................................................. 52
   A.1 Terminal output of a successful system run .......... 52
   A.2 DHCP files ......................................... 55
   A.3 EDK files ......................................... 55
   A.4 I²C files .......................................... 57

Bibliography ............................................. 60
## List of Figures

1.1 CBM Experiment side view[1] .............................. 2
1.2 Detectors’ slices placement[3] .............................. 3
1.3 Image of a central Au+Au collision and rare “open charm” decay. ... 4
1.4 n-XYTER channel architecture[5] .......................... 4
1.5 Detailed schematic of an n-XYTER channel[4] ............... 5
1.6 A simplified diagram of a Xilinx Virtex-4 FPGA slice.[8] ........ 6
1.7 The hard PowerPC processor core diagram[9] .................. 7
1.8 Virtex-4 FX12 Mini Module board[10] ........................ 7
1.9 Virtex-4 FX12 MM plugged into it’s baseboard[10] ............. 8

4.1 A schematic of an exaplme I²C bus setup. ..................... 42
4.2 Picture of the I²C-attached 7-segment LCD display. ........... 44
4.3 Oscilloscope images of captured I²C signals. .................. 45
4.4 Increasing voltage threshold value as set over the I²C interface. 47
4.5 The front panel of the n-XYTER test setup LabVIEW visual instrument 48
4.6 Call library function dialogue and respective functional element below (the one with orange top). ......................... 49
Acknowledgements

The author is indebted to dr Adam Czermak, who supervised the thesis, for introducing him to the world of silicon detector front-end electronics. Author would also like to express his gratitude towards two other members of the Department\textsuperscript{1}, Bogdan Sawicki and Barbara Dulny for helpful criticism and advice. Additional thanks should go to dr Pawel Staszel at the Institute of Physics of the Jagiellonian University for facilitating internship at the Department which the origin of this thesis.

\textsuperscript{1}Department of the Structure of Atomic Nucleus of the Institute of Nuclear Physics of The Polish Academy of Sciences in Krakow, Poland
Chapter 1

Introduction

This thesis was completed at the Institute of Nuclear Physics in Krakow, Poland as a part of preparatory work on a front-end electronics detector read-out. The thesis’ focus was on the building a hardware and software system for the n-XYTER ASIC\textsuperscript{1} VLSI\textsuperscript{2} chip which is proposed to be used in reading out strip detectors positioned inside Silicon Tracking System (STS). The STS is going to be used in Compressed Baryonic Matter (CBM) experiment which will take place at the Facility for Antiproton and Ion Research (FAIR) at GSI Institute in Darmstadt, Germany. A FPGA-based device, featuring a PowerPC processor core, running Linux OS was used as a partial solution.

1.1 Objectives

Silicon Tracking System is planned as an array of many position-sensitive silicon detectors, in total consisting of about 1.5 million channels. These detectors need to be able to relay large amounts of data for storage and later analysis. The data link used in the previous project at the Department of the Structure of Atomic Nucleus of the Institute of Nuclear Physics was USB 2.0. It’s point-to-point topology does allow relatively high transmission rates (over 30MBps [240Mbps] throughput was achieved) at the sacrifice of flexibility. Optimal results were obtained with only one device connected to the PC with the cable length no longer than few meters. At this scale this is no longer acceptable. An Ethernet network topology is more suitable, together with its current speeds topping 10Gbps. With cable lengths up to 100 meters and flexibility of ”many to many” logical topology, the systems can be based on existing network and server infrastructure or can be designed using widely available and very affordable (compared to proprietary equipment) network hardware.

In order to store the massive amounts of data produced by the n-XYTERs we need devices capable of high speed logic-level and differential signaling to interface the chips and network protocols aware device to work with the servers. A Virtex-4 FX12 device featuring a PowerPC processor core was used (together with an 1Gbps

\textsuperscript{1}Application Specific Integrated Circuit, a type of integrated circuit (IC) designed for a specific task, as opposed to for example, general-purpose CPUs.

\textsuperscript{2}Very-Large-Scale Integration, a technique for integrating a very large amount of transistor-based circuits into a single chip.
Ethernet controller) for its unique composition of speed of the FPGA fabric and flexibility of a general-purpose CPU.

Linux Operating System was chosen as a software layer providing an enormous set of useful system tools as well as facilities to interface diverse hardware with a common Application Programming Interface (API), enabling the use of well-known programming languages such as C. The open source nature of its code and its portability allowed to modify it to facilitate the needs of the Memec Virtex-4 FX12 MM hardware.

1.2 Compressed Baryonic Matter experiment (CBM)

In the CBM experiment, the scientists will study the properties of highly compressed baryonic matter, produced in high energy nucleus collisions. A central collision of two atoms of gold accelerated to relativistic velocities yields a total of \( \sim 1000 \) particles. Different particles can then be identified basing on time of flight, energy loss or track curvature. Capturing and analysing their traces and energies will help to understand properties of the dense matter, forming neutron stars and, according to the current state of knowledge, the origin of the Universe. It will also allow to study the Quantum Chromo Dynamics’ (QCS) fundamental properties, such as color confinement and chiral symmetry.

![CBM Experiment side view](image)

Figure 1.1: CBM Experiment side view[1]

The CBM Experiment consists of several layers of detectors positioned behind a dipol magnet (see figure 1.1), which deflects charged particles towards the detectors. The accelerated atom nuclei enters the reaction chamber and hits the one resting inside the magnetic field. Still inside the magnet the first set of detectors is placed, called the Silicon Tracking System (described in next chapter). Next, Ring Imaging
Cherenkov (RICH) detectors are placed together with Transition Radiation Detectors (TRD). The RICH detectors will observe electrons and mesons, and the array of three TRD will measure higher momentum electrons. Resistive Plate Chambers (RPC), positioned next will perform time of flight measurements. Other particles should deposit their energies inside the electromagnetic calorimeter (ECAL) which is the last instrument. The information was taken from the CBM Letter of Intent[1] and the CBM Introduction[2].

1.3 Silicon Tracking System (STS)

The STS is the most sophisticated component of the experiment. It will be used to capture track and momentum of charged particles. It consists of two main parts (see figure 1.2), the Micro-Vertex Detector (MVD), consisting of two silicon pixel detectors, and proper STS, consisting of eight silicon strip detectors.

![Figure 1.2: Detectors’ slices placement][3]

Two Monolithic Active Pixel Sensors (MAPS) building the MVD will have a pixel size of 40x40 μm, and will be 100 μm thick. They will feature a 3 μm resolution, which is more than 10 times smaller than the pixels itself, due to the charge distribution over several contiguous pixels. The STS will consist of 8 detector sets based on double-sided silicon strip detectors, featuring 60 μm strip pitch and being 300 μm thick. Signals from the detectors will be sent over capton micro-cables to the readout electronics. Because the MVD and the STS readout electronics will reside inside the dipol magnet, and can be hit by ionized particles, a radiation hard devices must be used. Providing chips with sufficient readout speeds, as well as the network architecture enabling the relay of the information will be a great challenge. A particle stream of about $10^{10}$ per second is expected. The proposed solution allows for momentum determination with an accuracy of about $\Delta p/p = 1%$[3].

1.4 n-XYTER ASIC

The n-XYTER ASIC is a mixed signal, position-sensitive, silicon detector readout chip. It consists of 128 analog input channels with two signal shapers per channel, and the digital readout circuitry (see figure 1.4).

![n-XYTER Architecture](image)

Figure 1.4: n-XYTER channel architecture[5]

Each analog channel consists of preamplifier fed into a fast shaper with time-walk compensation, which is used for timestamp latching, and a slower one with peak detection for signal amplitude capture. The timestamp comes from an internal timestamp generator which can be fed from a source common to a number of chips.
and allows to correlate signals captured by different detectors. The signal’s captured timestamp data can be read out over a 8-bit parallel digital interface, and the respective voltage representation of the signal amplitude can be sampled on an analog pin as well. Each channel features a digital and analog FIFOs capable of storing 4 consecutive hits. The FIFOs outputs are connected to a token ring data bus, so that busy channels are granted access.

![Figure 1.5: Detailed schematic of an n-XYTER channel][4]

Total digital information is 4-bytes wide, consisting of the Grey encoded timestamp (14 bits) and channel ID (7-bits for 128 channels) as well as other signals such as FIFO overflow and signal pile-up flags. The chip also features various testing and calibration circuits which can be configured over it’s I²C interface. An entire 4 bytes together with the analog signal can be read at the frequency of 32 MHz. More information can be found in the n-XYTER manual[4] or at the n-XYTERs wiki page at CBM[5].

### 1.5 Virtex-4 FPGA chip architecture

The modern day integrated circuits, offering the highest speed of parallel data processing, whilst maintaining the ability to be re-programmed are the Field Programmable Gate Arrays. They allow design of a custom logic that can be used in such areas as real-time video compression, high-speed cryptography and signal processing. They offer clock speeds approaching 550MHz and resources corresponding to 330k logic cells. Their configuration can be specified using a hardware description language like VHDL or Verilog, much in the same way as for regular ASICs. It allows to design any kind of combinatorial or sequential logic using abstraction as simple as logic gates together with more sophisticated one like latches, registers, multiplexers and so on.

Virtex family logic cells are grouped in structures called slices (see figure 1.6). A Virtex-4 FPGA slice consists of two 4-input LUTs (Look-Up Tables) that can implement any 4-input boolean function, two dedicated user-controlled multiplexers for combinational logic (MUXF5 and MUXFX), dedicated arithmetic logic and two
1-bit registers that can be configured either as flip-flops or as latches. The input to these registers is selected by YMUX and XMUX multiplexers[8]. Virtex-4 FX12 device consists of 5 472 slices corresponding to around 12k logic cells. It has 648 Kbits of a Block RAM memory (∼81KB).

Virtex 4 was the second Xilinx chip manufactured in the 90nm technology. It was the first chip to introduce the Advanced Silicon Modular Block technology (ASMBL), allowing a different mixes of Logic, DSP\(^3\), connectivity and embedded processing domains in a single chip. The common features of the platform are the power and ground routing, and the clocking signals. Then distinctively, Virtex-4 LX features the highest logic density, the SX version — highest DSP performance, whereas Virtex-4 FX (the chip used in preparation of this thesis) excels in the processing domain, providing embedded processor cores and high-speed serial I/Os (based on [7]).

Virtex-4 FX12 chip includes general-purpose programmable fabric, an embedded 32-bit PowerPC processor (with a maximum clock of 450MHz) together with a Auxiliary Processing Unit (APU) allowing the processor to execute custom instructions designed in the FPGA fabric, and of two 10/100/1G Ethernet Media Access Controllers (see figure 1.7)[9].

1.6 Test board architecture

The hardware best meeting the thesis' subject requirements is a Virtex-4 FX12 Mini Module board by Memec, which was available at the time of writing this thesis at

\(^3\)Digital Signal Processing, accelerating the common transformations of a digital representation of an analog signal.
the Institute on Nuclear Physics. The Virtex-4 FX12 MM includes all components an embedded system needs. The processor is the PowerPC core embedded in the Virtex-4 chip. There are both high-speed volatile (DDR RAM) and slower non-volatile (Flash) memories on board. The most interesting feature is a 1 Gigabit Ethernet controller, with it’s higher-level logic (EMAC core) contained inside the Virtex 4 FX12 and the physical layer chip located on the board.

The board measures 30x70 mm including the Ethernet socket. It connects to the baseboard, or any user-designed board with a two dual-row 64 pin headers (see figure 1.8).

The Mini-Module can be bought together with a baseboard (see figure 1.9),
providing power and data connectors as well as LEDs and buttons. It is an important resource in development as the user is free of designing a board just for the testing purposes.

The Mini-Module offers the following features:

- Virtex-4 FX12 FPGA
- Two clock sources (100MHz and 25MHz)
- JTAG-accessible Platform Flash Memory for FPGA configuration storage.
- 64 MB of RAM (organized as 32M x 16)
- 4 MB of Flash Memory (organized as 2M x 16)
- 10/100/1000 Mbit/s Ethernet transceiver PHY chip (BCM5461 by Broadcom) with three link status LEDs
- An FPGA controlled LED
- A RJ45 Ethernet Port
- Two 64 pin (2x32) I/O headers that can be directly soldered into a motherboard or plugged into the baseboard socket.

Additionally, the baseboard provides:

- A JTAG Port
- A serial port DB9-F connector with a RS232 driver IC and a serial-to-USB transceiver with a USB-B socket.
- DIP/Push Switches and LEDs and a 2x16 LCD panel accessible from the FPGA.
• A 5V input and switching power supplies for 3.3V and 2.5V
• User and SAM/IO pin headers
• A socket for the user FPGA clock and a power switch

The information presented above was based on User Guide for the Memec Virtex-4 FX12 Mini-Module[10].
Chapter 2

Principle of Operation

Although the author obtained the following knowledge in a painful process of trial-and-error, (the description of which is spread throughout the entire thesis), he feels the need to briefly explain the mechanisms by which the final, working system operates. It is his belief that the reader, acquainted with the final solution will be more receptive to the description of the way in which the result was achieved. Therefore, the author’s understanding of what a processor-based detector system undergoes since it’s early power-on stages until reaching nominal operating conditions, follows.

2.1 Hardware

After the power is applied to the board the FPGA is a tabula rasa and it’s configuration is being forced upon it by the Platform Flash memory, using a high-speed parallel data bus. When the process is complete, in our case the “DONE” LED on the base board lits up, and the FPGA comes to live. The PowerPC processor starts executing instructions stored in it’s on-chip Block RAM\(^1\) and the FPGA fabric performs it’s assigned roles. Interfaces often implemented on the FPGA side include the RAM memory controller, serial port, \(^{1}\)IIC interface and Flash memory controller, since the PowerPC core is only the CPU itself.

2.2 Software

When the hardware part of the chip is initialized, it’s the software turn to take over the control of the PowerPC core. The image of the first stage bootloader is already in the Block RAM, and the CPU, kept in the reset state until now, begins executing it. Note, that because of the bigger the space constrains of the boot level, the lesser the knowledge of the hardware of the respective bootloader can be. The first stage bootloader has to fit together with the hardware description into the Platform Flash memory, the second stage — into a 16 KBytes of Block RAM (it measures just over 9 KB), and the third — into a 4MB of the external flash memory (the SREC file size is slightly less than 512KB).

\(^1\)The BRAM contents can be initialized by the Platform Flash Memory during the configuration phase
First stage bootloader — the bootstrapping

The bootstrapping\(^2\) process is the description of the processor’s ability to build an operating system environment, from it’s “empty”, power on state. The first level bootloading is actually the Platform Flash initial process of putting the second stage bootloader into the Block RAM memory. In case of the system discussed it loads a program called “Simple SREC Bootloader” provided by the Xilinx EDK. It’s operation is described in the following section.

Second stage bootloader — “Simple SREC Bootloader”

This bootloader is also provided by the Xilinx company, and can be generated using the EDK software. It’s principle operation is to load a third-stage bootloader, stored in the Flash memory in the SREC (String Record)\(^3\) format. It needs to fetch the image into the RAM memory, translate it to an executable format (SREC contains ASCII characters representing binary data) and begin the execution of it. It is therefore mandatory for the program to know how to interface the Flash controller. This differs it from the first stage bootloader which is aware only of the resources of the physical chip itself.

Third stage bootloader — U-boot

U-boot, the Universal Bootloader\(^11\) is the final boot tool used to fetch the Linux kernel over the Ethernet network. The U-boot first accesses the Flash memory to load it’s environmental variables, such as the MAC address it assigns to the Ethernet controller. It then obtains it’s network configuration using the DHCP\(^4\) protocol. The received configuration includes the board’s own IP address, and the IP address of the server that runs the TFTP service. Now the boards tries to contact the server using the TFTP\(^5\) protocol over in order to download the description of the hardware it is being run on (a so-called device tree), and the Linux kernel. Then it executes the downloaded kernel passing it as an argument the device tree, and the (hopefully working) kernel takes over.

Operating system

Every operating system can be divided in two parts: the kernel and the userland. Kernel runs in a special part of the memory that no process can read or write. Kernel can directly interface to the hardware as only kernel “sees” the real memory layout. It provides the user processes with memory allocation and other services such as opening and reading files, network sockets and so on. The userland is everything else that runs on a system. This includes the system libraries, services and regular user processes. They cannot see the hardware directly, it is hidden behind the kernel.

---

\(^2\)The [bootstrapping] term is often attributed to Rudolf Erich Raspe’s story, “The Surprising Adventures of Baron Munchausen”, where the main character pulls himself out of a swamp, though it’s disputed whether it was done by his hair or by his bootstraps, as one can read on Wikipedia.

\(^3\)It is a format developed by Motorola for storing binary images as strings of ASCII data, including a checksum or the image integrity verification.

\(^4\)The Dynamic Host Configuration Protocol, described in section 3.4 on page 19

\(^5\)The Trivial File Transfer Protocol, described in section 3.4 on page 20.
They can however take full advantage of a modern operating system features such as simultaneous operation on a single CPU or TCP/IP stack and network abstraction layer.

Linux kernel

After the last stage bootloader, kernel now executes, initialises all the devices it’s aware of, and tries to mount it’s root filesystem. It knows where to look for because of an root= option passed to it from device tree. Then it tries to load and execute the init userland process, which is responsible for starting all the other operating system services. In our case kernel tries to mount the root filesystem over the NFS\textsuperscript{6}.

Linux userland

The first process that begins execution in the user mode is the init process, often residing in the /sbin directory. init in our case if a part of the busybox\textsuperscript{12} package. It’s behaviour is controlled by the configuration files stored under /etc directory, most notably /etc/inittab, and system services startup scripts located inside /etc/init.d/ directory. inittab instructs the system to spawn a terminal login session on the serial port, so we can log into the system using a terminal connected to it. A complete serial port output of our system can be viewed in Appendix A.1, Terminal output of a successful system run, on page 52.

System services, or as per the Unix terminology, “daemons” in our case are:

- The SSH daemon\textsuperscript{7} (called dropbear\textsuperscript{13})
- The WWW server\textsuperscript{8} (called thttpd\textsuperscript{14})
- and The NTP client\textsuperscript{9} (called ntpdate\textsuperscript{15}).

All of which are described in their respective subsections in section 4.3, Operating system on page 37.

Normal operation

At this stage the system is declared up and running, and can perform it’s assigned tasks. The author now feels justified in explaining the process in which he arrived at the working system, and provide an example of it’s use.

---

\textsuperscript{6}The Network File System, protocol, described in section 3.4 on page 20
\textsuperscript{7}The Secure SHell daemon, enabling a remote login sessions, described in section 4.3 on page 39
\textsuperscript{8}The World-Wide Web server, serving HTTP requests, described in section 4.3 on page 40
\textsuperscript{9}The Network Time Protocol client, used to obtain current time over the Internet, described in section 4.3 on page 41
Chapter 3

Tools

In order to build and run Linux on the PowerPC processor inside the Virtex-4 device, two kinds of tools are needed. The first kind is used to establish an embedded platform, then configure and build the bootloaders, Linux kernel and system binaries. The second kind of tools is used to provide services to the board over the network, such as provide it with an IP address and all the files it needs for operation. To accomplish that, one needs following development tools:

3.1 Xilinx ISE

The Xilinx ISE (Integrated Synthesis Environment) allows to design and synthesise the FPGA logic. Hardware description files can be written in VHDL or Verilog. This tool is prerequisite for Xilinx EDK, since all the devices added by the EDK are in the form of VHDL modules, and they need to be synthesised anew every time a component is changed. This is a commercial tool, which I used under the Institute of Nuclear Physics license. There is a free version available with a limited functionality, called the Xilinx WebPack, which allows to design logic on Xilinx CPLD and FPGA devices. Both ISE and WebPack can be installed and run on Linux.

The VHDL design cycle

For reader not acquainted with the VHDL language, or the way in which it controls the hardware, an example source code and it’s life cycle is shown. The function is simply to blink an LED. The purpose of this demonstration is for the reader to better understand the process of synthesis and possibly get a little feel of the bare metal that all the sophisticated software runs atop.

The author familiarised himself with the described process by writing a number of simple VHDL modules, and even making a small PCB featuring an Xilinx XC9536XL CPLD\footnote{Complex Programmable Logic Device, a smaller, less sophisticated brother of the FPGAs} device for greater understanding of the software-described-hardware paradigm.

The VHDL language will be used in building an IP (Intellectual Property) core embedded into the FPGA fabric for reading out the nXYTER detector data. The peripheral which will emerge this way will also be accessed by the PowerPC processor over the shared memory bus interface, to send the data over the network. The
author attempted to familiarise himself with the detector readout, but the amount of study required increased dramatically. As such, this part of the system was left for another team member, a full-time researcher who has already wrote a nXYTER readout system for the Virtex-II family.

The VHDL source usually consists of two main parts, the definition of the input and output signals, and their expected correlation, a so-called “behaviour”:

```vhdl
entity led is
  Port (clk100_in : in std_logic;
        LED : out std_logic);
end led;
```

This section defines an input, `clk100_in`, that we will later connect to a 100 MHz clock located on the Mini Module, and an output, `LED`, that we will later connect to the only user-controllable LED on the MM. Now comes the functional part:

```vhdl
architecture Behavioral of led is
  signal cnt : std_logic_vector(31 downto 0);
  constant speed : integer := 23;
begin
  LED <= cnt(speed);
  process(clk100_in)
  begin
    if rising_edge(clk100_in) then
      cnt <= cnt+1;
    end if;
  end process;
end Behavioral;
```

It defines the `cnt` as a vector, which we can treat as an integer variable (being, all in all, a bit array), and sets the `speed` constant to 23. We will use the `cnt` variable as a counter, incrementing it by one until it overflows, (at the value of $2^{31}$) and then again and again. After the `begin` statement, we connect (the “<=” symbol) one bit of a variable (addressed by parenthesis) to an output signal, `LED`. The bit is the 23rd bit of our counter, the `cnt` variable, as indicated by the `speed` current value, 23. This, effectively divides our input signal clock of 100 MHz, by $2^{23}$, giving us $\sim 12$ Hz at the output (the `LED` signal). Next comes the actual increment statement, that adds one to the `cnt` variable on it’s every invocation. An interesting thing to note is that a VHDL source can have many `process` statements, and each of them is “executed” in parallel, with respect only to it’s input clock, which must always be specified. There is no defined order of execution between processes, as this concept does not apply here.

Apart from the VHDL file, a User Constrains File (.ucf) is needed. It maps the inputs and outputs declared in the `Port` section of an entity to the physical pins or pads on the actual device. In our case it looks like this:

```ucf
NET "LED" LOC = M5 | IOSTANDARD = LVCMOS33;
NET "clk100_in" LOC = B13 | IOSTANDARD = LVCMOS25;
```

\(^2\)It’s actually $\sim 6$ Hz because the counter is only incremented at the rising edge, which basically halves the input signal frequency. It was left as 12 Hz for the clarity, just to show the principle of operation.
Apart from the pads’ addresses (the Virtex4 FX12 we are using comes in an sf363 BGA\(^3\) package, featuring 363 pads under the device [with an area of less than 3cm\(^2\)!]) the UCF file also specifies the logic level voltage standard. We use LVCMOS25 for the clock input since the clock buffer circuitry is designed to run at 2.5V. A UCF file can often be generated using a tool called PACE (included in the ISE Suite). It allows the user to drag the signal names to the graphical representation pins or pads and then generate the corresponding UCF.

Now three steps take us to a working device: Synthesis, Generating the programming file, and Configuring the Target device, all of which have their corresponding actions in the Project Navigator of Xilinx ISE. The output file, with the .bit extension, is downloaded to the device using a tool called Impact\(^4\) over a JTAG interface by which the PC running ISE and the Virtex4 device are interconnected. If all goes well, we have a blinking LED. However, we only configured the running device, so the FPGA contents will be lost together with the presence of the power supply voltage. To make the configuration “permanent”, we need to generate a .mcs file, again, using the Impact tool, and program the Platform Flash Memory (located on the board) with it, using, again JTAG. On power-on, the Platform Flash will configure the FPGA device according to its own contents.

### 3.2 Xilinx EDK

The Xilinx EDK (Embedded Development Kit) allows to design embedded systems consisting of selected VHDL-described components and hard cores, perform the synthesis of the platform (using the ISE Foundation software suite) and to write software for it. The user interface is provided by the Xilins Platform Studio (XPS) program. This tool also lets the user design its own peripheral that can be added to the embedded platform, and later interfaced by the software. It also facilitates the software debugging, using the XMD\(^5\) and gdb\(^6\) debuggers. EDK also provides a big help in creating the second-stage boot loader, by providing a source code for it, which only needs to be slightly customized in order to work.

### Running under Linux

Installation under Linux was straightforward and presented no problems. There were however some preparations to run the tools and program the attached devices. One such thing was that Xilinx tools were trying to access the JTAG programmer over a proprietary kernel module called win drv r. The module was intended for older Linux kernel versions (<2.6.18), and does not work with the current ones. A workaround was found by Michael Gernoth, who wrote a library called \libusbdriver\[^{16}\]. Despite its name, it works for the parallel-connected devices as well. It’s path has to be placed in the \LD_PRELOAD\ environmental variable, so that it overloads the library functions the tools would normally use to access the programmers, either

---

\(^3\)Ball Gate Array, a package used in modern space-constrained application, where signals are relayed to the PCB by a two-dimensional array of pads which offer smaller overall package of the chip, compared to conventional “pinned” packages

\(^4\)Impact comes together with the Xilinx ISE Foundation software

\(^5\)The XMD - Xilinx Microprocessor Debugger, described in section 3.2 on page 18

\(^6\)The Gnu DeBugger
over the parallel or the USB port. For simpler execution of the tools, a wrapper
scripts were created, one for every tool, such as Xilinx Project Studio, ISE, XMD,
Imapct and so on. They have exactly the same body, the only difference is the last
line, containing the \texttt{exec} statement followed by the tool’s name. To test different
EDK and ISE versions, at times the paths were also altered, so I was able to have
different versions installed at the same time. A complete script for starting the
Xilinx Project Studio from EDK 10.1 is located in Appendix A.3, \texttt{startedk shell
script} on page 55.

\section*{Obtaining a working platform}

The XPS features a wizard for creating an embedded platform. For a given board,
it’s components together with their settings (like the integrated circuit’s pin-to-pin
interconnection paths, etc.) are included in an \texttt{.xbd} (Xilinx Board Description) and
an \texttt{.ucf} (User Constrains File, discussed in the previous chapter) files supplied by
the board manufacturer. Xilinx board’s description files come with the EDK suite.
For the board the thesis was based, they had to be downloaded from the Memec site
and placed under the \texttt{EDK/board/} hierarchy. It is interesting to note that the file’s
contents change from EDK one version to another, and they need to be updated
together with the software. The author learned it the hard way, when after an
update he wasn’t able to get a new setup working, and was receiving a somewhat
mysterious behaviour in return.

Having the board support files in place, one can easily create an embedded
platform by simply following the wizard suggestions. To get a system on which
Linux will run, few things need to be set. The user has to instruct every peripheral
such as a serial port or the Ethernet interface, to use interrupts. Otherwise they
might not function properly. Also a 16KB of On-Chip Memory has to be chosen,
instead of the default 8KB, since the second-level bootloader is slightly over 9KB in
size and will not fit in the 8KB during the “Initialise BRAM” phase, and the user
will be served with a not very useful error message. Also to utilize the full potential
of the Ethernet hardware provided on the board, the TriMode\_MAC\_GMII\textsuperscript{7} device
should be chosen with XPS\_LL\_TEMAC\textsuperscript{8} peripheral using the Scatter-gather DMA
mode. None of these settings are chosen by default, and figuring out have caused
the author a lot of grief. One of such examples is, that to be able to even enable the
DMA mode, there has to be a MPMC (Multi-Port Memory Controller) controller
available in the system, and it refused to appear automatically because of a version
mismatch between the \texttt{.xbd} Board Description files and the EDK environment.

It was found by the author, that a detailed description of running the Base
System Builder wizard together with a screen shot of every step can be found Steven
Kauffmann’s page at [17].

Now that the wizard setup has created the system, the user is free to add new
peripherals to it or change the settings of the existing ones. One example a such
new peripheral might be the P\textsuperscript{3}C interface, but it’s addition to the system in this

\textsuperscript{7}It can be de-ciphered as following: Three modes refer to 10, 100 and 1000 Mbit speeds, MAC
is the Media Access Controller, and GMII stands for Gigabit Media Independent Interface. The
Tri-Mode Ethernet MAC is called TEMAC for short.

\textsuperscript{8}XPS stands for Xilinx Project Studio, and LL is the dedicated, high-speed Local Link interface,
by which the TEMAC and the PowerPC core are connected, in addition to the standard PLB
(Processor Local Bus) connection
way is rather lengthy and error-prone. As such it was circumvented by the author in another way, described in section 4.4 on page 42. An example of a property change is using the TX and RX checksum computation offload to the FPGA fabric, that can be chosen for the Ethernet interface.

Now is also the time to experiment with the programs running on the “bare” hardware, that is without the operating system. With some customisation of the Xilinx provided routines one can use the serial terminal as the input/output device. An example of a program which should be added and compiled at this time is the bootloader that will take care of bringing up the u-boot bootloader placed in the Flash memory, as described in the section 2.2 on page 11. This bootloader however, was not added to the solution until after the Linux kernel and the U-boot bootloader started operating properly, so a corresponding description is located under section 4.2, The SREC bootloader on page 36. Another example of a program which user might want to run is the peripheral test example program which tests the input, output (to the serial terminal), and can test user LEDs and GPIOs with buttons or switches connected.

If no changes are desired, we can proceed with generating the bitstream (the output file of the hardware synthesis, with a .bit extension). The process can take from 45 minutes (on a modern, multicore PC) to several hours, depending on the performance of the workstation. The bitstream will then be ready to be downloaded to the Virtex-4 over JTAG. After the generation, we can initialize the bit-stream file with software, if we created any. We can also proceed directly to downloading the bit-stream into the FPGA device, using a dedicated button on the GUI. When the download is complete, we have an embedded platform featuring the PowerPC processor core that we can run programs on. We can shorten the software development process. We don’t have to initialize the bit-stream file with the executable every time it’s changed, and then download the updated bitstream to the device. We can utilize the XMD debugger to directly load the program into the processors memory, once the hardware has been configured.

Software Platform Settings

Under the Software menu, there is an option called Software Platform Settings. It allows the user to choose for which software platform should the supporting files, describing the hardware, be generated. Three different options needed to be used during preparation of the thesis. The first mode is standalone. It doesn’t assume any operating system will be run, and it generates a set of libraries that can be used directly by the software written under the Xilinx Platform Studio. This mode is used for compiling the Simple SREC Bootloader. The second mode is called linux_2_6. It will generate, among other things, the xparameters.h file that will be used for U-boot bootloader configuration. I also used it in the first phase of research, because the arch/ppc kernel tree needed this sort of C-header, file-based configuration. The last mode is called device-tree. It is not present on the list by default. Instructions on how to install the files needed for it’s operation can be found at [18]. It’s function is described in the next subsection. It’s important which platform is chosen and was generated, as only one kind can be active and present in the project at a given time. Therefore, after a change in hardware which would affect any of the software components, all necessary libraries have to be re-generated.
Generating the Device Tree

After we have generated the system’s bitstream, describing the hardware, we need a way of presenting it to the Linux kernel. Since all the devices are accessed by their memory-mapped register access, the kernel has to know which device occupies which memory range, and what version of the peripheral to expect. On regular PCs this is normally done by the Plug and Play mechanism, but it’s too complicated for an embedded platform, so we have to do it some other way. The old method was to generate the Board Support Package (BSP) files that included source code and header files for the drivers inside the Linux kernel. Also, a memory address of every peripheral was specified inside those files. This however, among other things, required the BSP to be re-generated after a change as simple as changing the peripherals order, and therefore, their addresses. After that, kernel had to be recompiled. The reconfiguration and recompilation was a lengthy and painful process, which, much to author’s delight, was superseded by the Xilinx Device Tree Generator\[18\]. After applying the instructions described under the reference, a single file is generated, containing in it’s tree structure all the devices and their addresses present on the system. The device tree is then either attached to an already compiled kernel, or can be downloaded by the U-Boot bootloader as a separate file, to be later passed as an argument to a booting kernel process. The freshly generated device tree is a text file (.dts) containing a tree-like (hence the name) that specifies the peripherals’ addresses, their general types (serial port, network controller) and various properties, such as the kernel command line\(^9\). Since the board non-volatile memory is a valuable resource, the file is often converted into a binary form.

XMD - Xilinx Microprocessor Debugger

The XMD attaches itself to a special part of the PowerPC core dedicated to debugging over the JTAG interface. We can invoke the debugger from the GUI, and in order for it to connect, we simply type into it’s window: connect ppc hw. Other choices are to connect to a Xilinx “soft” MicroBlaze processor (mb), or to connect to a simulator (sim) instead of hardware (hw), but these settings here are just provided for reference. Then we download the binary in to the processor by typing dow executable.elf and run it by simply typing run. Downloading a fairly large executable (like the Linux kernel, over 1MB in size) can take relatively long time, measured in minutes. The author started experimenting with running the Linux kernel in this way, but becoming increasingly anxious of the outcome of the download during the lengthy process, he resolved to different ways of doing so, which are explained under section 4.1, Booting Methods on page 23.

3.3 Cross compiler

All described preparatory work was done on an Intel processor-based PC. Since PowerPC is a different type of processor, featuring different instruction set, programs compiled for one processor will not execute properly on the other. Being able to compile a program to be run on a different architecture requires a special

---

\(^9\)The kernel command line is mentioned in the section 4.1, Configuration on page 21.
kind of compiler called a cross-compiler, and if any features of an operating system will be used, also a set of libraries. Creating a cross-compiler by hand can be done and it has been used by the author for several platforms (such as arm or avr architectures) before, but this time he decided to use a readily available tool, named simply Crosstool\textsuperscript{[20]}. Crosstool takes care of downloading all necessary software packages (the GCC compiler, the binutils package and the Gnu C Library — glic), versions of which can be specified by the user. After the lengthy build, they’re usually placed in /opt/crosstool directory, with the next directory tree levels representing the software versions and the architecture, like gcc-4.0.2-glibc-2.3.6/powerpc-405-linux-gnu. Here finally resides the bin directory where the proper compiler binary is stored, named after the architecture and operating system (in our example: powerpc-405-linux-gnu-gcc), so it’s easy to distinguish our own platform’s compiler from the PowerPC’s one. In order to make use of the new compiler suite, it’s a good idea to put the lengthy directory path into the system’s PATH environmental variable, and when configuring software that will be compiled, mention to it the prefix by which the proper compiler can be identified, that is powerpc-405-linux-gnu. An example of usage is presented in section 4.2, Configuration on page 28.

### 3.4 Servers

Finally, the only tools left are the servers. As described in Chapter 2, Principle of Operation on page 10, the board relies on network resources for providing it with the network configuration, kernel image and filesystem access. In that order, we need to configure: The DHCP server, the TFTP server and the NFS server, which setups are described below:

#### Dynamic Host Configuration Protocol (DHCP)

The DHCP protocol allows to remotely assign network settings to a machine, such as the IP address, default gateway etc. Apart from those, it provides the board with the information about who will be serving the files it needs to boot. A popular choice of software to run as the server is the DHCP package provided by the Internet Software Consortium\textsuperscript{[21]}. It’s configuration, specific to the setup we are using consists of declaring a host statement, together with the board’s Ethernet MAC address. The declaration specifies the next server the board will contact to obtain the kernel image (over TFTP). The host declaration used looks like this:

```
host b0 {
  option host-name "b0.nz22.ifj.edu.pl";
  option root-path "/tftpboot";
  next-server 192.168.0.1;
  hardware ethernet 00:de:ad:be:ef:00;
  fixed-address 192.168.0.60;
}
```

This configuration also sets the hostname, and the path of the root of the filesystem for the board, respective to the servers filesystem hierarchy. For convenience, all files concerning the board runtime environment will be stored under /tftpboot directory.
on the server. The explanation of all the options included in the configuration file is outside of the scope of this thesis. A complete working file however is included in the Appendix A.2 on page 55. It mostly includes details about the network the device is located inside, and some parameters for the use of the DHCP daemon.

**Trivial File Transfer Protocol (TFTP)**

TFTP is a very simple protocol. There’s no logging in, no username or password, just files one can download. It also works over UDP, a less sophisticated protocol than TCP. TFTP is used by the U-Boot, the final bootloader, to download the kernel image to the board’s memory. A software package called tftp-hpa[22] was used. The server’s configuration is rather simple, and is actually limited to setting the “root” directory path, the highest-level directory the tftp clients will have access to. Just as in the DHCP section, it is pointing to `/tftpboot`:

```
INTFTPD_PATH="/tftpboot/
```

**Network File System (NFS)**

NFS is a very old protocol, a standard among Linux machines for sharing files over network. After Linux kernel has been brought up, the NFS will provide the diskless board with a remote filesystem access. Since the FX12MM board does not include a suitably sized non-volatile storage for an operating system, a networked storage was a natural choice. First a local, System Ace (Compact Flash-based) card was tested, but the NFS storage is much more flexible, because all the updates and modifications to both the kernel and other software can be made on the server PC disk on-the-fly, without the need to remove and update additional media, and then to reboot the board. The NFS software needs support in the Linux kernel, but it often can be loaded as a kernel module, without the need of recompilation. It’s central point of configuration is the `/etc/exports` file, inside which a directory path and corresponding IP address allowed to access it should be defined. A sample configuration follows:

```
/tftpboot 192.168.0.0/24(rw)
```

The `rw` option in the parentheses tells the NFS server to allow writing to the filesystem.
Chapter 4

System Realization

4.1 Linux PowerPC kernel

The main Linux kernel development takes place in the kernel.org repositories. The kernel version offered there can be run on a number of different architectures. But because of the scale of the project new patches and device drivers are not integrated at once and need to undergo testing and quality assurance procedures beforehand. This has led some organizations and companies to create their own repositories, to which they can freely apply any corrections they like. Therefore, apart from the “stock” Linux kernel from kernel.org, there are versions more adapted to running on Virtex-based systems. They often contain more drivers for Xilinx-specific devices, like the Xilinx TEMAC, or Xilinx I²C interface. The first such resource that the author tried was the Secret Lab\[23\] repository located at git://git.secretlab.ca/git/, and the final one that yield best results was the Xilinx repository, located at git://git.xilinx.com/linux-2.6-xlnx.git.

Configuration

The Linux kernel has a multitude of options to be configured. For PowerPC processors it even has two separate architecture types, which complicates things a little. One of them resides in arch/ppc, the other one in arch/powerpc. As for the PowerPC405 cpu, arch/ppc directory is not being actively developed, however, it is mentioned in most of the available (like [25], [26] or [27]), aging manuals on how to build a working Linux setup. The author’s first attempts at building and running a kernel on the board were based on the older branch, as suggested by the Secret Lab[23] how-to document[24]. These efforts succeeded as far as the kernel is concerned. Linux was unable to boot into userland from the kernel because of a dynamic-linking problem\(^1\). The configuration of arch/powerpc is mentioned in sources such as Xilinx Open Source Wiki[28] and upon finding such a useful piece of information I proceeded along this way. The author was able to compile and run a working kernel and, after solving the linking problem mentioned above, to run init process and boot into busybox[12]-based distribution as well.

Settings such as PCI, firewall, in-kernel cryptography, NLS (National Language Support), miscellaneous file systems, frame buffer or mouse and keyboard support

\(^1\)Which ultimately was my fault. Please see section 4.1, /sbin/init and dynamic linking problem on page 25 for more information.
were found not to be needed and upon disabling made the kernel ~300KB smaller.

The first thing that needs to be done is to tell the Makefile which architecture and compiler we will be using. It can be either done by setting the environmental variables inside the system shell, or by putting them into the makefile. The values used for this thesis were:

```
ARCH = powerpc
CROSS_COMPILE = powerpc-405-linux-gnu-
```

One should also remember about adding to the PATH environmental variable the directory where the cross-compiler resides. Of course it can be placed in the user log-in script, like ~/.bashrc.

Then, the kernel configuration command line interface can be invoked just like for a regular kernel build, by typing: make menuconfig.

The arch/ppc kernel configuration required much time and figuring out the right options, and it needed to be in sync with the Board Support Package source and header files that changed each time any part of the embedded platform changed even a bit. However, the Xilinx version of the configuration of kernel source was almost ready to use, with only few customizations needed. One of the options that had to be discovered by trial-and-error was an option called “Xilinx LLTEMAC PHY Support” in the Device Drivers -> Network device support -> Ethernet (1000 Mbit) section. There were three choices, all related to Marvell 88E1111 PHY chip. It is the chip used in Xilinx boards. However, the FX12 MM features a Broadcom device. The choices were: RGMII, GMII and “MII or other PHY”. They represent the interface between the MAC part (a core inside the Virtex4-FX12) and the PHY chip (a Broadcom device). From the pin assignments table in the FX12 MM it was inferred that the proper choice would be GMII. However, This setting introduced increased delay during the kernel startup and worked only with 10Mbit speed, unable to negotiate anything faster. The RGMII option refused to work entirely, but the “MII or other PHY” one worked just fine.

One of the more important parameters is the kernel command line option. It defines such settings as the location of the root filesystem (for example root=/dev/nfs) or the console device (a serial terminal can be specified by using console=ttyS0). A command line used by the author most of the time was:

```
console=ttyS0,9600 ip=on root=/dev/nfs rw
```

It sets the baud rate of the console device (being the first serial port) to 9600 baud and lets the kernel obtain it’s IP settings over the DHCP\(^2\). Together with the NFS server’s address (ip=on) and sets the root to the network location obtained by the DHCP as well (root=/dev/nfs), in the read-write mode (rw).

**Compilation**

In the older version of the kernel tree, the source was compiled by simply typing make. It produced a binary file located under arch/ppc/boot, named zImage.elf that was ready to be executed on the target system. The binary included the not only the support for certain devices, but also their memory-mapped register addresses.

\(^2\)Other options provided by the DHCP are included in section 3.4, under Dynamic Host Configuration Protocol (DHCP) on page 19
The Device Tree method offered by the arch/powerpc is more flexible, but since the kernel does not know neither the number nor the location of the supported devices, it needs the device tree file. It can be either attached to it to form a stand-alone executable .elf file, or stored under the /tftpboot directory to be later fetched by the U-boot bootloader and passed on to the kernel.

**Booting Methods**

Before the kernel can be executed from the board’s RAM memory, it needs to be placed there in some way. All three methods described were used in development of the final result.

**JTAG**

The first, most natural way is to place the binary image directly into the memory using the JTAG interface. It is the method used by the EDK environment for running programs written and compiled in Xilinx Project Studio. After the hardware description gets loaded, one can proceed to initialize the memory that the processor uses, with it’s debug section that can be remotely controlled. The Xilinx Microprocessor Debugger is used for that purpose. More information on this too can be found in section 3.2, under XMD - Xilinx Microprocessor Debugger on page 18. As mentioned there, downloading and running a Linux kernel this way is a time-consuming process, but it can be realized without the need of any additional hardware apart from the JTAG programmer.

After the board is powered up, the FPGA fabric needs to be configured first. If the Platform Flash does not contain a proper image, we can download it over JTAG. The author found a way of doing this straight from the command line by typing:

```
impact -batch etc/download.cmd
```

After the download is done, one should invoke the XMD debugger, by typing xmd and input the following commands:

```
connect ppc hw
download zImage.elf
run
```

Where the *zImage.elf* is the kernel executable image file to be downloaded, as produced in the compilation step.

**System Ace**

A faster startup time can be realized by using a System Ace card. It is a Compact Flash-based add-on card together with supporting circuitry. It plugs into the baseboard SAM-IO header, and takes control over the board’s JTAG pins. It then relays the CF contents to the Virtex-4 FPGA. After the hardware is configured, an optional binary file can be loaded into the embedded system memory, and executed. There is also a peripheral that can be added to the platform than can interface the System Ace card, and utilise the Compact Flash memory partitions as a regular non-volatile storage. As such it can be used by Linux as a file system storage device, just like a regular hard disk drive in a PC.
The file that contains both the hardware description (.bit) file and the executable (.elf) that will be used to bring the board to operation is called system.ace. To create this file, a set of options need to be specified to the XMD program. They can be saved to a file and passed as a parameter to the XMD. To create the system.ace file one should navigate to the XPS project directory using the terminal and type:

```
xdm -tcl genace.tcl -opt xupGenace.opt
```

where xupGenace.opt is the name of the options file. It’s contents can be viewed in the appendix A.3 on page 56. This produces the expected file that needs to be copied over to the Compact Flash card using a suitable card reader. It is important to note, that the module expects a FAT16 filesystem. To format a partition in such a way under Linux, mkdosfs command can be used. An example would be `mkdosfs -F16 /dev/sdc1`, where /dev/sdc1 is the first partition on the CF card. Then, the card should be placed in the System Ace module, the module should be inserted into the SAM-IO socket and the board should be powered up. Hopefully there should be text output on the serial terminal at the baud rate selected in the configuration process.

To also use the CF card as a local filesystem storage, it’s System ACE support should be compiled into the kernel. The suitable option is located under Device Drivers->Block Devices kernel configuration tree. The exact option name is XIL-INX_SYSACE. The device should be recognized by the kernel and should appear in the system as /dev/xsa. We should also tell the compiled kernel where to look for the root partition. If the root filesystem is located on the third partition, we should use a parameter: root=/dev/xsa3 in the kernel boot options. The output from the kernel, so called dmesg (debug message) for a successful local System ACE root mount would look like this (note that this dmesg is from the arch/ppc kernel tree, as can be seen in the first line):

```
Linux/PPC load: console=ttys0,9600 root=/dev/xsa3 ip=off
...
xsysace xsa: Xilinx SystemACE revision 1.0.12
xsysace xsa: capacity: 250368 sectors
xsa: xsa1 xsa2 xsa3
Xilinx SystemACE device driver, major=254
...
VFS: Mounted root (ext2 filesystem).
```

While offering greater loading speed than both the JTAG and Network boot methods, System ACE solution has one major drawback. Any changes in the system, either to the kernel, or to the software or it’s configuration are much slower. One has to power down the board, remove the CF card from the System ACE module (often unplugging the SysACE module from the baseboard as well), place the CF card in the reader, power the reader on, apply all the changes, and then reverse the procedure. Theoretically, a faster work cycle can be obtained by mounting the filesystem containing the system.ace file on the board, and access it over the network. But if we do have a network connectivity, we can avoid using the local filesystem storage altogether, which is what the next section describes.
Network

The most flexible option for both kernel booting and filesystem storage is the network. In order for a kernel to be able to access remote filesystems, one has to add few options to the kernel configuration. In the usual menu, accessible by typing `make menuconfig`, there is a `File systems` submenu, under which `Network File Systems` option should be checked. This option contains a submenu, where options such as `NFS client support` should be checked, and under it, `Root file system on NFS` option, since we’re planning to have no local storage whatsoever. Therefore even the root filesystem will be mounted over the network. And for that we should pass a kernel booting option like `root=/dev/nfs`. This however is not sufficient, as the kernel has to be able to make a connection to the server that contains the filesystem. This can be done in two ways. We can either pass a complete configuration in the kernel command line, or instruct it to use DHCP protocol, and obtain the settings automatically. The second approach was taken, and the option is simply `ip=on`. A successful attempt by the kernel to obtain the IP configuration and then to mount it’s root filesystem should look like this (as copied from a serial terminal):

```
Kernel command line: console=ttyS0 root=/dev/nfs rw ip=on
...
Sending DHCP requests ., OK
IP-Config: Got DHCP answer from 192.168.0.1, my address is 192.168.0.60
IP-Config: Complete:
  device=eth0, addr=192.168.0.60, mask=255.255.255.0,
  gw=192.168.0.1,
  host=b0.nz22.ifj.edu.pl, domain=nz22.ifj.edu.pl,
  nis-domain=(none),
  bootserver=192.168.0.1, rootserver=192.168.0.1,
  rootpath=/tftpboot
Looking up port of RPC 100003/2 on 192.168.0.1
Looking up port of RPC 100005/1 on 192.168.0.1
VFS: Mounted root (nfs filesystem).
```

/sbin/init and dynamic linking problem

When kernel completes it’s initialization process and mounts it’s root filesystem, it is ready to pass control to userland. It does so, by executing the `init` program. My first attempts at booting into the userland failed with a following message:

```
VFS: Mounted root (nfs filesystem) readonly.
Freeing unused kernel memory: 104k init
Warning: unable to open an initial console.
Kernel panic - not syncing: No init found. Try passing init= option to kernel.
Rebooting in 180 seconds..
```

It looked as if the kernel was unable to find the `init` program. I was sure it was placed properly under `/sbin/init`, where the kernel should be looking for it. I was sure the kernel can actually see the NFS mounted filesystem, because I was
able to get rid of the “Warning: unable to open the initial console” line by creating a device node for /dev/console.

This was probably the most time-consuming issue in the whole thesis. I was extremely happy I finally configured, built and ran a working kernel, but I was unable to make even one step further. I tried placing init in different folders, changing it’s name, passing it’s different location as a run= parameter to the kernel. I also wrote a small and simple program which should just output a text to the console and exit, and replaced /sbin/init with it. Nothing helped. I started to investigate the problem by searching the web, and looking at the kernel source, because (unfortunately\(^3\)) I had enough C-language programming skills to understand it well enough. I found a place from which the “No init found” is being print out. It was the main.c file inside the init/ directory. The code looked like this:

```c
run_init_process("/sbin/init");
run_init_process("/etc/init");
run_init_process("/bin/init");
run_init_process("/bin/sh");
panic("No init found. Try passing init= option to kernel.");
```

The run_init_process function looks like this:

```c
static void run_init_process(char *init_filename)
{
    argv_init[0] = init_filename;
    kernel_execve(init_filename, argv_init, envp_init);
}
```

The idea is that the execve function tries to execute the file passed to it as a parameter, and if it succeeds, it terminates the old execution path. Therefore, it does not return to the original instruction flow, so the init process is not run four times, as the source could suggest, but the first run_init_process function that “fires”, takes over. If, however, the execve system call fails for some reason, it does return to it’s calling program, carrying an error number which can help to identify the cause of the failure. Going this way, I altered the run_init_process subroutine slightly, to indicate the error cause. I found out that the error codes being returned by the kernel system calls are the same as for the userland versions, only being negative. So I captured the kernel_execve return error code and printed it out. Later I also found the error messages corresponding to the numbers, and the final debug version of the run_init_process function looked like this:

```c
if ((ret =
    kernel_execve(init_filename, argv_init, envp_init)) < 0) {
    printk(KERN_WARNING "Failed to execute %s: %d ",
            init_filename, ret);
    if (ret == -ENOENT) {
        printk(KERN_WARNING ": No such file or directory.\n");
```

\(^3\)As it will be indicated at the end of the subsection, the problem was not buried that deep. This investigation however taught me a lot about the Linux kernel.
After running this version, a following output was obtained (The <4> characters are generated by the printk function, and are used for message importance level classification. Here they happen to be output because they’re not the first characters in the line of text):

Failed to execute /sbin/init: -2 <4>: No such file or directory.
Failed to execute /etc/init: -14 <4>: Bad address.
Failed to execute /bin/init: -14 <4>: Bad address.
Failed to execute /bin/sh: -14 <4>: Bad address.

An interesting thing to note was, that if I moved the /sbin/init to /etc/init, the first error still would be No such file or directory (ENOENT, as in Error NO ENTRY, indicating that there is no such directory entry), then the second one, for an existing file, would be the same, and the rest would be Bad address regardless of whether the file was present or not.

By searching the internet, I learned that the ENOENT can also be caused by not existing dynamic library that the program is linked against. To verify that, I tried statically linking the simple program I wrote, by passing -static flag to the gcc compiler from the cross-compiler tool-chain. However, something must’ve gone terribly wrong, because the problem persisted. I even tried compiling a program written in assembly, but sure enough instead of writing to the hardware directly it was using the standard output, and as such it was depending on an external library. Since this gave not result either, I left the problem alone for some time. When I tried again to statically link the simple C program and place it instead of the init, it just worked! Last time, I must have been copying a different file than the output of the compilation, and this is quite possible, since my irritation was beyond comprehension. Having solved that issue, I statically linked the busybox-provided init, placed it where the kernel expected it, and it just worked! Oh the mirth. This was definitely one of the happiest moments in the preparation of thesis. Later I found that I would have to statically link every single program I would like to run on the board. This resulted in the executables being bigger and loading slower. Finally, I copied over the /opt/crosstool/gcc-4.0.2-glibc-2.3.6/powerpc-405-linux-gnu/ powerpc-405-linux-gnu/lib directory to the board’s /lib and that just simply worked. I was able to run executables dynamically linked against glibc, compiled using the cross-compiler. A successful console output looked like this:

VFS: Mounted root (nfs filesystem).
Freeing unused kernel memory: 164k init
init started: BusyBox v1.12.4 (2009-01-28 00:13:20 CET)
starting pid 768, tty ’’: '/etc/init.d/rcS'
Initializing random number generator... done.
Starting network...
ip: RTNETLINK answers: File exists
Welcome to the Virtex-4 FX12 MM Linux system.
b0 login:

4.2 Bootloaders

Until now, the board software setup was hybrid in a way, consisting both of local and networked booting methods. The hardware configuration and the kernel executable were being loaded using the System ACE solution. The kernel was taking over, obtaining its IP address and mounting its root filesystem over the network. The board contained a Platform Flash that could hold a hardware description, and enough general-purpose Flash to hold some kind of bootloader, it was time to remove the supplementary System ACE card, which was not going to be a part of the final solution.

U-boot bootloader

U-boot, the Universal Bootloader was chosen as the final stage bootloader, as suggested on Xilinx Wiki[28]. Getting it to work proved to be troublesome, as there was no existing configuration for our specific setup. However, by trial and error in selecting different options and editing the sources and recompiling, I was able to find the right mixture.

Configuration

Xilinx version of U-boot bootloader including the setup files specific for the ml405 and ml507 development boards can be downloaded from the Xilinx repository using git command:

\[
git\ clone\ git://git.xilinx.com/u-boot-xlnx.git
\]

For either of these platforms, U-boot can be simply configured by setting the environmental variables according to our desired architecture:

\[
export\ CROSS\_COMPILE=powerpc-405-linux-gnu-
\]

\[
export\ ARCH=powerpc
\]

and by typing, for example:

\[
make\ ml507\_config\make
\]

However, the Xilinx U-boot tree does not include a ready configuration for our setup. I later found out that the U-boot’s own repository contains a setup for the Virtex-4 FX12 MM board, but with a different configuration than the one used in thesis. I decided to customize the Xilinx U-boot software. I copied the configuration for ml507 board, and renamed it to fx12mm. This included an fx12mm Makefile target, a fx12mm directory under board/xilinx structure and a configuration file, fx12mm.h inside the include/configs directory.
Because U-boot has to know the exact description of the hardware it will be run on, such as where to look for the Ethernet or Flash controller, and does not support the device tree for the configuration phase, we have to generate a `xparameters.h` file that was used by older versions of kernel configuration setup. This file contains all the peripheral addresses and parameters. It can be generated from the XPS. First one needs to open the `Software` menu and click `Software Platform Settings`. There, under `OS & Library Settings 1inux_2.6` should be chosen, and the dialog window should be closed by clicking `OK`. Then, again from the `Software` menu `Generate Libraries and BSPs` should be clicked. It also has a dedicated button on the GUI. A more detailed description of the `Software Platform Settings` can be found under section 3.2, `Software Platform Settings` on page 17.

The `xparameters.h` file forms the basis of the `board/xilinx/fx12mm` directory, and is the only major difference between it and `board/xilinx/ml507`. Because the early versions didn’t have the I²C interface which is included in the ml507 design, I had to comment out a line which is trying to read the Ethernet MAC address from the I²C connected EEPROM memory chip, and also disable the driver code generation, by setting the ` XPAR_IIC` variable in the `xparameters.mk` file to ‘n’.

The `include/configs/fx12mm.h` file is the actual U-boot configuration file. It is a C programming language header file, which contains definitions of devices’ addresses and other options. Among other things, the serial port baud rate (`CONFIG_BAUDRATE` variable), Ethernet MAC address (`CONFIG_ETHADDR`) or the inclusion of Flash memory-related commands (`CONFIG_CMD_FLASH`) can be set. A complete file is rather long and was included on the compact disk attached to the thesis.

**Ethernet drivers problem**

After I managed to obtain a successful build, I tried to run U-boot on the board. After downloading hardware description to the FPGA, I loaded the U-boot executable and ran it by opening XMD and typing:

```
connect ppc hw
dow u-boot
run
```

Because I had to download the U-boot image to the board quite a few times, and for a clear device setup a power-cycling was sometimes required, I developed a script consisting of a set of often-repeated commands. I put the above few lines into a script I named `dow-uboot.xmd`, and wrote a following shell script:

```
#!/bin/sh

echo -n "Impact... "
(cd ~/project_dir; startimpact -batch etc/download.cmd > /dev/null)
echo done.
echo -n "Xmd... "
startxmd < dow-uboot.xmd > /dev/null
echo done.
```

After loading and running U-boot, on the serial port, the U-boot prompt appeared:
I issued\(^4\) the command for obtaining the network configuration over DHCP, but the board appeared no to do anything. I enabled the debugging mode, and ran it once more. This time I was able to see where the problem might be:

```
=> setenv ethaddr 00:0a:35:01:02:03
=> dhcp
LLTEMAC Initialization Started
Initializing DMA...
Setting the MAC address...
XTemac: PHY detected at address 4.
XLIITemac: speed set to 10Mb/s
LLTEMAC Initialization complete
BOOTP broadcast 1
Waiting for frame to be sent
```

Since I had the Linux kernel running, I was able to verify that the PHY address indicated (4) was correct. However, the network should be 100Mbit, not 10. It happened to the kernel before, when the PHY chip type was not configured properly. I identified the speed detection function to be the \emph{set\_mac\_speed()} in the \texttt{board/xilinx/xilinx\_lltemac/xlltemac\_adapter.c} file. I compared the function to the corresponding one (also called \emph{static void set\_mac\_speed()}, in the \texttt{xlltemac\_main.c} file inside \texttt{drivers/net/xilinx\_lltemac}) in the Linux kernel, since it works there. The U-boot’s function was exactly the same as a part of the function in Linux kernel. It was however only for a type of MAC which shipped with Xilinx boards. So I copied over the part that the kernel was using for the detection to U-boot, and, after fixing some dependencies, it worked, and the speed was being set properly. However, U-boot was still waiting for the frame to be sent, so the speed was not the case here.

```
XTemac: PHY detected at address 4.
XLIITemac: Not able to set the speed to 1000 (status: 0x7949)
XLIITemac: We renegotiated the speed to: 100
XLIITemac: speed set to 100Mb/s
LLTEMAC Initialization complete
```

\(^4\)In the following examples, the convention is that everything that appears after the U-boot prompt (the => characters) is user-entered. Anything else is generated by the board’s software, mostly U-boot bootloader, but can also be the kernel, when it starts booting.
I enabled even more in-depth debugging, by defining the \texttt{xdbg.printf} function in the board/xilinx/xilinx_llitemac/xlltemac.c file, but it still revealed nothing past the \texttt{Waiting for frame to be sent} message.

I started reading various system sources and Xilinx documentation, and found out that the TEMAC peripheral can work in two different modes for accessing data. They are FIFO and DMA modes. In our setup, the FIFO mode was chosen, as it is chosen by the board setup wizard by default. However, U-boot supports only the DMA mode. Having learned that I created a number of different hardware designs in XPS, but I wasn’t able to synthesise all of them due to lack of required IP (Intellectual Property) licenses. I finally managed to obtain a setup with TEMAC on EDK version 9.1. I was unable to use the UART peripheral, which license probably expired, and had to resort to it’s free alternative, UARTlite. This required some re-configuration of the U-boot \texttt{fx12mm.h} file. On this setup however, the U-boot did not work for some reason. I choose to upgrade the software, since new version of both EDK and ISE were available. I tried to re-create my previous setup, this time with the LL TEMAC peripheral in the DMA mode. XPS was refusing to complete this setup, claiming that there are no DMA providers. I tried under both 9.2 and 10.1 versions, with the same result. I found out, that the DMA supplier would be the Multi Port Memory Controller, or MPMC for short. This was a standard peripheral that the EDK would usually add automatically, so something was wrong. I finally found out the board description files, .xbd, were outdated, because they were good only for 9.1 EDK version. I downloaded appropriate .xbd files and I was able to complete the setup wizard. However, during the synthesis one of the processes stopped claiming a failed library dependency. This is a relatively common error when binaries are moved from one system to another. The name of the executable was xilperl. I located it in ISE/bin/lin/unwrapped/ directory, and ran \texttt{ldd} command on it, which lists all the dynamic libraries the software depends on:\footnote{The convention here is that lines preceded with a `#` character are input to the terminal by the user. Other lines are the output of the commands}

\begin{verbatim}
# ldd ./ISE/bin/lin/unwrapped/xilperl
linux-gate.so.1 => (0xffffe000)
libnsl.so.1 => /lib/libnsl.so.1 (0xb7f15000)
libdb-4.1.so => not found
libdl.so.2 => /lib/libdl.so.2 (0xb7f11000)
libm.so.6 => /lib/libm.so.6 (0xb7eeb000)
libc.so.6 => /lib/libc.so.6 (0xb7dbb000)
libcrypt.so.1 => /lib/libcrypt.so.1 (0xb7d8d000)
libutil.so.1 => /lib/libutil.so.1 (0xb7d88000)
/ld-linux.so.2 (0xb7f45000)
\end{verbatim}

The \texttt{libdb-4.1.so} library comes from the Berkeley DB 4.1 package. I had BDB 4.3 installed in the system. The simplest solution was to place a symbolic link to the 4.3 library and see if it works.

\begin{verbatim}
# ln -s /usr/lib/libdb-4.3.so /usr/lib/libdb-4.1.so
\end{verbatim}
The library was located, and the synthesis ran fine. I reverted all the changes I did while trying to debug the Waiting for frame to be sent phase, and it worked!

=> dhcp
BOOTP broadcast 1
*** Unhandled DHCP Option in OFFER/ACK: 28
*** Unhandled DHCP Option in OFFER/ACK: 28
DHCP client bound to address 192.168.0.60
TFTP from server 192.168.0.1; our IP address is 192.168.0.60
Filename '/kernel'.
Load address: 0x400000
Loading: ########################################################

The kernel file was downloaded automatically over TFTP because there was a filename statement pointing to it, in the DHCP server configuration file.

Running U-boot

A complete set of commands available for the current compilation can be obtained by entering help after the U-boot prompt. Description of a specific command can be obtained by adding it’s name after the command, like help dhcp. If the DHCP server does not specify the file to use, we can first obtain just the network configuration (the dhcp command) and then download the file we need, also specifying the address in the memory under which to store it. We need two files (as described in chapter 2, Principle of Operation), the Linux kernel binary (it’s version for U-boot, called ulimage, which contains some additional information), and the device tree blob (a binary .dtb file, being a compiled device tree ASCII file). We can actually pass arguments to the dhcp command, like dhcp 0x1000000 fx12mm.dtb, where 0x1000000 is the mentioned offset, and second argument is the filename, with respect to the /tftpboot directory on the server in our case. A set of commands to download the files and boot the kernel would look like this:

=> dhcp 0x1000000 fx12mm.dtb
BOOTP broadcast 1
*** Unhandled DHCP Option in OFFER/ACK: 28
*** Unhandled DHCP Option in OFFER/ACK: 28
DHCP client bound to address 192.168.0.60
TFTP from server 192.168.0.1; our IP address is 192.168.0.60
Filename ’fx12mm.dtb’.
Load address: 0x1000000
Loading: #
done
Bytes transferred = 12288 (3000 hex)
=> tftp 0x1C00000 kernel
TFTP from server 192.168.0.1; our IP address is 192.168.0.60
Filename 'kernel'.
Load address: 0x1c00000
Loading: ########################################################
##############################################################
done
Bytes transferred = 1601665 (187081 hex)

The `tftp` command can be used to download the specified file to the specified address in the RAM memory after the network is configured. Kernel image residing in the memory can be booted by using the `bootm` command.

```sh
bootm 0x1c00000 - 0x1000000
```

The arguments are the memory addresses of: kernel, initrd (Initial RAM disk, not used in our setup, hence '-') and device tree images. I used the addresses specified on the Xilinx Open Source Wiki[19]. My first attempts to boot the kernel were not successful.

```sh
=> bootm 0x1c00000 - 0x1000000
## Booting kernel from Legacy Image at 01c00000 ...
   Image Name: Linux-2.6.28-rc6
   Image Type: PowerPC Linux Kernel Image (gzip compressed)
   Data Size:   1601601 Bytes = 1.5 MB
   Load Address: 00000000
   Entry Point: 00000000
   Verifying Checksum ... OK
## Flattened Device Tree blob at 01000000
   Booting using the fdt blob at 0x1000000
   Uncompressing Kernel Image ... OK
   Loading Device Tree to 007fa000, end 007fffff ... OK
Unable to update property /plb/opb:clock-frequency,
   err=FDT_ERR_NOTFOUND
Unable to update property /plb/ebc:clock-frequency,
   err=FDT_ERR_NOTFOUND
Unable to update property NOR mapping, err=FDT_ERR_NOTFOUND
## Transferring control to Linux (at address 00000000) ...
   Booting using OF flat tree...
zImage starting: loaded at 0x00000000 (sp: 0x00196f1c)
Allocating 0x365618 bytes for kernel ...
Insufficient memory for kernel at address 0! (_start=00000000,
   uncompressed size=00342ff0)
```

I compared the few last lines with the output of the kernel booting from a Compact Flash card, it was following:

```sh
Allocating 0x365618 bytes for kernel ...
```
gunzipping (0x00000000 <- 0x0040d000:0x0059452e)...done 0x342ff0 bytes

Linux/PowerPC load: console=ttyS0 root=/dev/nfs rw ip=on
Finalizing device tree... flat tree at 0x5a2300
Using Xilinx Virtex machine description

I tried loading the kernel under a different address, and also found out that the uImage can be obtained by simply typing `make uImage`, instead of doing it manually. I found the working memory address to be 0x900000, so the set of instructions altogether was: dhcp 0x1000000 fx12mm.dtb; tftp 0x900000 kernel; bootm 0x900000 - 0x1000000. In U-boot, the commands can be input on the same line, if semicolon-separated. This line therefore constitutes the only line that would have to be input to the U-boot to download the device tree and kernel and boot into Linux.

**Flash setup**

Since I had a working U-boot setup, I was vitally interested in not having to load it every time over JTAG. After enabling Flash memory support in the `fx12mm.h` config file and rebuilding U-boot, I was able to check if the Flash memory is detected by using the `flinfo` command:

```bash
=> flinfo

Bank # 1: CFI conformant FLASH (16 x 16) Size: 4 MB in 71 Sectors
AMD Standard command set, Manufacturer ID: 0x1F, Device ID: 0xC8
Erase timeout: 4096 ms, write timeout: 1 ms

Sector Start Addresses:
FF800000 FF810000 FF820000 FF830000 FF840000
FF850000 FF860000 FF870000 FF880000 FF890000
FF8A0000 FF8B0000 FF8C0000 FF8D0000 FF8E0000
FF8F0000 FF900000 FF910000 FF920000 FF930000
FF940000 FF950000 FF960000 FF970000 FF980000
FF990000 FF9A0000 FF9B0000 FF9C0000 FF9D0000
FF9E0000 FF9F0000 FFA0000 FFA10000 FFA20000
FFA30000 FFA40000 FFA50000 FFA60000 FFA70000
FFA80000 FFA90000 FFAA0000 FFAB0000 FFAC0000
FFAD0000 FFAE0000 FFAF0000 FFBO0000 FFBI0000
FFB20000 FFB30000 FFB40000 FFB50000 FFB60000
FFB70000 FFB80000 FFB90000 FFBA0000 FFBB0000
FFBC0000 FFBD0000 FFBF0000 FFBE0000 FFBF0000
FFBF4000 FFBF6000 FFBF8000 FFBF90000 FFBBFC0000
FFBFE000

=>
I was able to copy regions of RAM memory into Flash memory by using the `cp.b` command. However, a Flash not Erased error appeared if I tried to write anything past the 0xFF802000 address, even though the memory was freshly erased. So I just jumped over the first sector, and started with the 0xFF850000. I was able to write the entire U-boot image there, so it was now that I obtained a Xilinx’s SREC bootloader working setup. I wrote the U-boot image in the Flash memory by doing:

```plaintext
=> dhcp 0x1000000 u-boot
...  
=> cp.b 0x1000000 0xFF860000 0x9ca1e  
Copy to Flash... done
=> cmp.b 0x1000000 0xFF860000 0x9ca1e  
Total of 641566 bytes were the same
```

Just to be sure the contents were really non-volatile, I power-cycled the board, and issued:

```plaintext
=> dhcp 0x1000000 u-boot
...  
Bytes transferred = 641566 (9ca1e hex)
=> cmp.b 0x1000000 0xFF860000 0x9ca1e  
Total of 641566 bytes were the same
```

Since I was able to place the U-boot bootloader in the Flash, I now started working on how to actually boot it on power up. What now followed is described below, in section The SREC bootloader on page 36. After I had both bootloaders working, I looked into another problem:

**Permanent Ethernet MAC address problem**

Because the final board’s application would consist of many devices, it made sense to use the board’s Ethernet MAC address as it’s identifier. It also made sense not to have to compile a separate kernel and U-boot binary for every board, having it’s MAC address embedded inside. Usually either a network interface card in the PC, or some Xilinx development boards have their MAC addresses written in an on-board EEPROM, but our board does not include such a device. Therefore, both U-boot and Linux kernel were up to generating a MAC address on their own, obviously different from each other. Among other things, this required a two DHCP entries, one for every address, apart from the address, being the same. It also meant every board used with this software would have the same MAC address, which would be in violation of Ethernet standards. The place that could comfortably hold the MAC address was of course the Flash memory. The MAC address can be set in one of the U-boot variables, of which the so-called U-boot environment consists.

It’s variables can be set by using the `setenv` command, and saved to the Flash memory using `saveenv` command. Setting the board’s MAC address can be done in this way: `setenv ethaddr 00:de:ad:be:ef:00`. After `saveenv`, U-boot uses this MAC address to obtain the network configuration over DHCP, but the kernel still uses 02:00:00:00:00:00. I found this address in the device tree that is being downloaded by U-boot and gets passed to the kernel. After deleting the device-tree node which holds this address, the kernel response was:
Device Tree Probing 'ethernet'
xilinx_lltemac 81c00000.ethernet: No MAC address found.
xilinx_lltemac 81c00000.ethernet: MAC address is now c0:33:63:f0:ff:ff

So it looks like the kernel is actually using the device-tree address and needs it. By browsing the U-boot source I found that U-boot should replace the device-tree specified MAC address with it’s own stored in the ethaddr variable. To see if the function actually gets invoked, I added a simple printf function to it (it was fdt_fixup_ethernet(void *fdt) function declared in the common/fdt_support.c file inside U-boot directory. The text got output, but the MAC address was not replaced. It seemed as if U-boot was unable to find the proper device node. I searched the Internet, and learned of similar problems that were fixed by defining an alias inside the device tree. So I created a following device tree node:

    aliases {
        ethernet0 = "/plb/xps-ll-temac@81c00000/ethernet@81c00000";
    };

And after compiling it to a device-tree-blob (.dtb), placing under /tftpboot and rebooting, it got replaced by U-boot:

    Updating property '/plb/xps-ll-temac@81c00000/ethernet
    81c00000/mac-address' = 00 de ad be ef 00
    Updating property '/plb/xps-ll-temac@81c00000/ethernet
    @81c00000/local-mac-address' = 00 de ad be ef 00

And used by the kernel:

    Device Tree Probing 'ethernet'
xilinx_lltemac 81c00000.ethernet: MAC address is now 0:de:ad:be:ef: 0

The SREC bootloader

I found out that a popular tool used for starting up a Xilinx PowerPC based embedded systems was a bootloader called simply “The SREC bootloader”, available in EDK. It is a small piece of software that can load the next stage bootloader from the flash memory at the specified address. The program itself is loaded into BRAM (Block RAM, a type of memory present inside the FPGA and accessible by the PowerPC core) during the power-on load phase of the FPGA. It’s the program that the PowerPC starts executing after power-on. The bootloader can be created by opening the Device Configuration menu in Xilinx Platform Studio, and then Program Flash Memory. We should check the box next to Create Flash Bootloader Application. The dialog menu also allows us to program the next stage bootloader (U-boot in our case) into the flash memory, and for that we should choose the offset at which to burn it. Currently both ELF and SREC formats are supported. When we close the configuration window, we should have a software project added to our project called bootloader_0. After compiling the software inside EDK, we should set it to be loaded into the chip’s BRAM by the Platform Flash on power-on, by clicking “Initialize to BRAM” button. Then we can update the previously generated bitstream with
software information. We can test the bitstream by downloading it to the FPGA device, and if all works, we can program the Platform Flash memory with it, so the bootloader will run every time the board is powered on. This way after powering up the board, the PowerPC will start loading the contents of the 2x16M Flash memory located on the Mini Module.

Because I was unable to use the Flash memory from its beginning, I had to define an offset of one sector, so 0x00050000 bytes. While I was trying to compile it, the first error was:

(error: undefined reference to `outbyte')

Since it was a linking error, I inferred out that I should make sure to set the **Software Platform Settings**, mentioned in the section **Software Platform Settings** on page 17, to standalone, and generate the libraries. It solved this issue, but a next one appeared, also in the linking phase:

```
powerpc-eabi-gcc -O2 bootloader_0/bootloader.c bootloader_0/srec.c -o bootloader_0/executable.elf -Wl,-defsym -Wl,_START_ADDR=0xFFFFE000 -g -I./ppc405_0/include/ -Ibootloader_0/ -L./ppc405_0/lib/

/opt/Xilinx92i/EDK/gnu/powerpc-eabi/lin/bin/../lib/gcc/powerpc-eabi/4.1.1/../../../../powerpc-eabi/lib/xilinx.ld:273 cannot move location counter backwards (from fffffec0 to 00000ec0)
```

I noticed that the 0xFFFFE000 address is the base of the BRAM and the from fffffec0 to 00000ec0 error looks like an overflow, so I decided to try with a bigger BRAM setup. After creating a new hardware platform and re-compiling, it linked just fine, resulting in a binary of around 12KB:

```
/powerpc-eabi/4.1.1/../../../../powerpc-eabi/lib/xilinx.ld:273
cannot move location counter backwards (from fffffec0 to 00000ec0)
```

However after loading the binary using XMD noting appeared on the terminal. I ran the bootloader using the software debugger, and stopped the execution after a while. The program was waiting in a loop, `while (!XUartNs550.mIsTransmitEmpty (BaseAddress));` The BaseAddress variable, which should point to the memory address of the UART peripheral registers was set to 0. I rebuilt the project, and this time the variable was set correctly, however the program was still waiting for the first character to be transmitted. I decided to turn off the verbosity by commenting out the `#define VERBOSE` line in the `bootloader.c` source file. After recompilation and reload, the U-boot welcome message appeared, so it simply worked. I created the .mcs file from the .bit hardware description one using the Impact tool, and programmed the Platform Flash with it. After power cycling the U-boot appeared again, and at this time I just couldn’t be happier.

### 4.3 Operating system

The “Linux” term actually defines as much as the kernel itself, common to all of it’s variants. Linux distributions differ among each other in system utilities and
programs that they offer. PC-oriented distributions are obviously too large and cpu-intensive to run on the limited resources that we have. Common choice among embedded systems are packages called buildroot[31] for the essential operating system ‘skeleton’ (like /etc/*, the device files and so on) and busybox[12] for the system binaries. Also, an interesting resource for designing a Linux-based platform is the Linux From Scratch project homepage[32].

The installation of the rest of the programs (thttpd, i2ctools etc.) was completed by fetching and unpacking the appropriate source, entering it’s unpacked directory and typing:

```
# ./configure --host=powerpc-405-linux-gnu
```

for the configure script to set the proper type of compiler in the makefile. The environmental variable PATH has to be setup correctly, because the configure script will be looking for a compiler that looks like powerpc-405-linux-gnu-gcc and assembler that looks like powerpc-...-gnu-as and so on. I then compiled each program by typing make and put the binary inside the board root directory, under /bin.

**Core system files: buildroot**

Buildroot is a package that makes it easier to create an operating system environment. It can take care of generating a cross-compiler, operating system tools and devices, as well as the Linux kernel itself. I already had a cross-compiler and a Linux kernel, so I was interested in obtaining just the filesystem with tools and devices. Buildroot configuration menu is entered by typing make menuconfig into the console while inside it’s main directory. I tried both an external toolchain configuration and a buildroot-generated one, and different mix of options in general. I obtained a root ext2 filesystem image that I tried to boot, but I failed. The internal toolchain build failed, and since I used the external one, but didn’t know I should copy the libraries over, I stopped at the /sbin/init boot problem, described in detail in section /sbin/init and dynamic linking problem on page 25. I basically copied over the files from the .ext2 image to the /tftpboot directory on the server, and worked on the busybox setup exclusively, since this is where I expected the problem to be. The buildroot package therefore supplied me only with the filesystem skeleton, onto which I copied the kernel, busybox utilities and the glibc library.

**Core system utilities: busybox**

Busybox[12] is a single executable which emulates behavior of many common UNIX utilities. It replaces such tools as find, grep, more, man and so on. It’s much smaller than all the separate tools would be altogether at the cost of some functionality.

Because it is a single program, an individual tool can be selected in two ways. Let’s assume we want to execute the ls command, for listing files in a directory. We can either do it like this:

```
b0# busybox ls
```

with all the arguments following, or create a symbolic link to the busybox executable:

```
b0# ln -s /bin/busybox /bin/ls
```
and then simply enter:

```
 b0# ls
```

This works because the busybox executable is able to tell by which name it was invoked. If it detects only `busybox`, it expects a command name as a parameter. If it detects that it’s being run under the name of any of commands known to it, it will simply assume it’s behaviour. Obviously busybox creates all the symbolical links automatically. This exercise was shown only to illustrate it’s *modus operandi*.

It’s configuration is entered with the `make menuconfig` command. It offers emulation of different UNIX utilities, which can be added or removed individually. The only option that actually needs to be set is the *Cross Compiler prefix* under *Busybox Settings* -> *Build Options*. An option I would often leave out was the `inotifyd` applet from the *Miscellaneous Utilities* category (the `CONFIG_INOTIFYD` option). It was causing errors in the compilation and I didn’t need it anyhow. After I discovered the dynamic linking problem, I immediately found an option for a static compilation, choose it and built busybox this way. I placed the executable, `busybox` inside the `bin` directory under `/tftpboot` where all the symbolic links were leading to. It worked fine, and prove to be a really nice and small replacement for all the tools.

After I discovered how to use the dynamic libraries, I obtained a dynamic build, and it worked fine as well.

One of the important things would be to set the board’s root’s password which we will use to log into the board, either over the serial terminal, or remotely. It can be done by passing `init=/bin/sh` option on to the kernel command-line (it can be edited over the serial terminal when the kernel is about to boot, or set from the U-boot bootloader). When we will arrive at the prompt, we should use the `passwd` program to set the new password.

**SSH server: dropbear**

The Secure Shell service offers a possibility to execute commands on a remote system just as if the user was logged in locally. It’s a more secure replacement for the venerable `telnet` protocol, which was sending the input and the output in plain text. SSH offers both server identity verification and data encryption. A remote login capability is vital in our setup, since the Ethernet RJ-45 plug will very likely be the board’s only connection. SSH protocol server that comes with most Unix-like operating system, the OpenSSH is too big and has too many features the board doesn’t need. A often-used replacement is called dropbear[^13].

Apart from just installing the software, an important thing to do is to generate the private encryption key for the server. It can be done with the `dropbearkey` tool which is supplied together with the server:

```
 b0# dropbearkey -t rsa /etc/dropbear/dropbear_rsa_host_key
```

After the key is generated, the server can be started simply by typing:

```
 b0# dropbear -k /etc/dropbear/dropbear_rsa_host_key
```

To log in from a Linux box, one should type:

[^13]: The server software was written by an Australian, which should explain the origin of the name. Dropbear is a legendary creature that attacks unaware tourists visiting the continent.
and then enter the previously set root password. Logging in from a Windows-based host can be done using the popular putty program.

**WWW server: thttpd**

Thttpd is “simple, small, portable, fast, and secure HTTP server”[14]. It can be compiled and run on most Unix-like operating systems. I used a www server to be able to serve content from the board in a simple and easy-to-use manner. The big advantage of thttpd is that while still small and portable it allows a user to run CGI scripts. I used this feature to demonstrate remote execution of binaries on the board.

I created a startup script for it and put it under `/etc/init.d` as `S60thttpd`:

```bash
#!/bin/sh

echo -n "Starting thttpd... 

/bin/thttpd -c "**" -d /root/dev/i2c

echo "done"
```

so that the service is started every time the board is powered on.

Script contents allow execution of cgi-scripts of any form (the ** argument) and set the server home to the `/root/dev/i2c` directory so the www client can not navigate outside of it. An example of use is presented in section I²C bus on page 41.

**I²C toolbox: i2ctools**

The i2ctools[33] suite was used for initial I²C bus testing and devices’ addresses discovery. It consists of three main tools, `i2cdetect`, `i2cget` and `i2cset`. First one is used for scanning the I²C bus in search for devices, second one for reading a specific device’s contents, and the third one to write to it. There was nothing special about the installation. As with the thttpd server, the example usage is presented in section I²C bus on page 41.

**TCP/IP performance: iperf**

In the desired environment, one of the boards key parameters will be it’s network throughput. To test it, I installed a package commonly used for this purpose called `iperf[34]`. It requires no configuration whatsoever. Setup consists of two machines, a server and a client. A server can be started like this:

```
server# iperf -s
```

and it will be listening for incoming connections. Then, on our board we instruct `iperf` to connect as a client:

```
b0# iperf -c 192.168.0.1
```

---

This feature is called the 'chroot'
to the 192.168.0.1, which is the server’s IP. The direction of the traffic will be from client to server, so upload will be tested in this way. I also performed some tests using the netperf tool. The results were quite disappointing, because the board’s TCP throughput was around 60Mbps, so 7.5MBps. The UDP results were more encouraging, achieving 120Mbps (15MBps) over a gigabit link, but there is no guarantee of packet reception. This is significantly slower than the USB connection used previously, which could achieve 240Mbps, or 30MBps, which is two times more than the UDP speed result.

Network interface settings: ethtool

The ethtool program can be use to change various settings of an Ethernet controller. I used it to increase the maximum Ethernet frame size (MTU, as in Maximum Transmission Unit), from the Ethernet’s standard of 1500 bytes to a so-called jumbo frame maximum of 9000 bytes, to increase the throughput. However the card on the other side of the gigabit link, Intel PRO/1000 PL inside my laptop suffered from a design flaw and effectively does not support jumbo frames.

Time synchronization: ntpdate

Ntpdate tool is a part of the ntpd[15] project. It allows the user to effortlessly and accurately update the system clock. It’s installation also went smoothly, and I added it to the system startup scripts, to obtain the date and time on every boot. It’s especially important in this computer system the board presents, since there is no battery backed-up hardware clock. On every boot the board assumes the Unix epoch time, January the 1st, 1970. First, the timezone should be set properly, otherwise UTC will be used. An examples timezone string for Poland would be CET-1CEST. It can be placed in the TZ environmental variable or, more permanently in the /etc/TZ file:

```
b0# echo CET-1CEST > /etc/tz
```

To update the time, one has to run the ntpdate tool presenting to it as an argument the time server’s address:

```
b0# ntpdate pool.ntp.org
3 Dec 22:03:44 ntpdate[10965]: adjust time server 149.156.70.5
    offset 0.225779 sec
```

4.4 I²C bus

The I²C (Inter-Integrated Circuit) bus is a serial, slow speed (hundreds of kHz) communication interface. It was designed by Philips around 1980 and gained much popularity since. It’s one of the most common ways to interface digital circuits, and devices supporting I²C range from Ethernet interfaces and digital thermometers, ADCs, DACs to relay arrays and CMOS camera’s configuration interfaces. Currently it’s difficult to buy a modern micro controller which would not support this bus. It was a natural choice for the nXYTER designers. The physical layer consists of two lines, SDA (Serial Data) and SCL (Serial Clock). Data is sent in 8 bit frames. Bus
supports bi-directional communication with acknowledgement and addressing, with up to 127 devices per bus. $I^2C$ can have two types of devices connected: master and slave. Master device always initiates the transfer, so reading from a slave device is by polling for data. The schematic of the bus is shown in figure 4.1.

![Figure 4.1: A schematic of an example I$^2$C bus setup. Image courtesy of Wikimedia Commons](35)

$I^2C$ addresses consist of seven bits. Additional bit sent just after the address distinguishes reading from writing. If a device is present on the bus, it should acknowledge reception of its address by pulling the data line low after the R/W bit. In this way scanning for devices on a bus can be performed.

**EDK setup**

In the upper part of the XPS GUI, there are two main workspaces. The one on the left lets the user choose and add files, applications or peripherals (Project, Applications and IP Catalog). There are also three tabs on the right, which allow setting various parameters of the peripherals already added. These are Bus Interfaces, Ports, and Addresses. Bus Interfaces usually serve to connect the added device to the processor interface, Ports are used to connect various status and signal lines between two peripherals or peripherals and external world, and Addresses allow to set and change the peripherals’ addresses, as seen by the processor on the memory bus.

Browsing through the IP catalog in EDK, under Communication Low-Speed I found XPS IIC interface. I added it to the project. There are three tabs in the middle of the GUI in XPS: Under Bus interfaces I chose the corresponding device, which was xps_iic_0, and connected it to the Processor Local Bus (plb setting in SPLB field). In the Ports tab I also created the interrupt line (IIC2INTC_Irpt), and set the SCL and SDA lines to Make external. I found two free pins on the Baseboard User I/O Header (J22) named IO_18_1 and IO19_1, which were FPGA pads V3 and V4. I opened system.ucf, since that’s where I expected the External Ports declarations to be, and I by looking at other peripherals setup, I added:

```
Net xps_iic_0_Sda LOC=V3;
Net xps_iic_0_Sda PULLUP | IOSTANDARD = LVCMOS33;
Net xps_iic_0_Scl LOC=V4;
Net xps_iic_0_Scl PULLUP | IOSTANDARD = LVCMOS33;
```

Under Addresses I clicked Generate Addresses, locking the addresses of all the other peripherals so they would not be accidentally changed, just in case. I generated the
hardware (around 20 minutes on Intel’s Core2 processor + 2GB of RAM), and the device tree, and ran Linux on them. After boot, nothing about I²C could be seen. However, the old, bad Ethernet MAC address was set. Obviously I forgot to add:

```plaintext
aliases {
    ethernet0 = "/plb/xps-ll-temac@81c00000/ethernet@81c00000";
}
```

to the .dts file. Since nothing about the I²C happened, I decided to try and find some description of the process of adding such interface to the system. I found one on the Xilinx Wikidot site[36], and I performed the setup as described. A wrong kernel driver was used, after recompiling, the `dmesg` contained:

```plaintext
i2c /dev entries driver
Device Tree Probing ‘i2c’
iic 81600000.i2c: no IRQ found.
```

It seemed that although I created the interrupt line, I did not connect it to the Interrupt Controller (INTC). I also did not have the `/dev/i2c/` directory or any device files that would look like the I²C bus device. I downloaded the MAKEDEV[37] script, placed it under `/dev/` and created the I²C bus device node by issuing:

```plaintext
b0# sh MAKEDEV i2c
b0# ls i2c*
i2c-0  i2c-1
```

### I²C-attached test display

Some time ago, experimenting with Atmel AVR microcontrollers I build a simple device, an I²C-controlled 7-segment LCD display (see figure 4.2). It was a big help in debugging this part of the project, since I didn’t have to have the whole detector board connected during the tests. The display was based on the Philip’s (now NXP Semiconductors) PCF8574[38] chip. It is a 8-bit I/O expander, and it has a 7-segment LCD display connected to it. The display is common anode model, so setting a bit inside the PCF to 0 lits up a segment.

I tried to access the display in two ways. I modified a program from the Xilinx Wikidot page[36] which was supposed to read an I²C-connected EEPROM into a program for reading out the PCF chip. I named it `i2cread`, and tried to run it. The full source of the program is presented in the Appendix A.4 on page 57. I used 0x70 as the PCF’s address, since I was able to do it using the Atmel’s device this way, but without success. I also tried to scan the I²C bus for any devices, so that the PCF address would appear automatically. Since `i2cdetect` tool from the `i2c-tools` package could not find any bus, I decided to generate the hardware with the `xps_iic_0 peripherals’ interrupt line connected to the interrupt controller. After the first bitstream generation, the kernel was unable to mount it’s NFS root filesystem, because apparently I forgot to connect the Ethernet’s controller interrupt line. After correcting it, the `dmesg` contained:

```plaintext
i2c /dev entries driver
Device Tree Probing ‘i2c’
81600000.i2c #0 at 0x81600000 mapped to 0xC50A0000, irq=20
```
This time `i2cdetect` noticed the I²C bus, but during the scan it produced a number of errors and hanged the board. I watched the SDA pin on an oscilloscope while the board was booting, and while the scan was in progress, but there was no change in voltage level. The `i2cread` program also did not help.

Since I knew that ML405 board does include an I²C interface, with an EEPROM connected to it, I decided to see how it’s `.xbd` file looks like. I found a suitable fragment, and created a copy of our board’s initial setup:

In the `/EDK/board/Memec/boards/` subdirectory under the EDK tree I copied the `Memec_V4FX12MM_Rev1` directory to `Memec_V4FX12MM_Rev1_i2c_Rev1`. I also added `i2c` to the names of the files inside, and also to the `ATTRIBUTE` fields inside the `.xbd` file. The actual lines that made the difference, were:

```plaintext
# Inter Integrated Circuit (IIC) Bus
# taken from Xilinx_ML405_v2_2_0.xbd
BEGIN IO_INTERFACE
  ATTRIBUTE IOTYPE = XIL_IIC_V1
  ATTRIBUTE INSTANCE = IIC_BUS
  PARAMETER C_IIC_FREQ  = 100000, IO_IS = clk_out_freq
  PARAMETER C_TEN_BIT_ADR = 0,  IO_IS = slave_respond_mode
  PORT SCL = iic_scl, IO_IS = Serial_Clock
  PORT SDA = iic_sda, IO_IS = Serial_Data
END
```

and:

```plaintext
# Inter Integrated Circuit (IIC) Bus
PORT IIC_CLK  = iic_scl, UCF_NET_STRING=("LOC=V3", "SLEW = SLOW", "DRIVE = 6")
PORT IIC_DATA = iic_sda, UCF_NET_STRING=("LOC=V4", "SLEW = SLOW", "DRIVE = 6")
```
added to the .xbd file, the second fragment after the BEGIN FPGA section. In the Base System Builder wizard the I²C bus appeared automatically, and after I finished the Builder the .ucf file was updated with following lines:

```plaintext
Net fpga_0_IIC_BUS_Scl_pin LOC=V3;
Net fpga_0_IIC_BUS_Scl_pin SLEW = SLOW;
Net fpga_0_IIC_BUS_Scl_pin DRIVE = 6;
Net fpga_0_IIC_BUS_Sda_pin LOC=V4;
Net fpga_0_IIC_BUS_Sda_pin SLEW = SLOW;
Net fpga_0_IIC_BUS_Sda_pin DRIVE = 6;
```

After booting the board on this hardware, the `i2cdetect -y 0` (the `-y 0` is specifying the I²C bus id equal to zero) command produced some numbers, and then hanged. On the oscilloscope all was in the low state. I added pull-up resistors as per the I²C specification, but now both lines were in the high state. I tried the `i2cread` program, and this time a clock signal could clearly be seen. However, there was an error:

```
Unable to talk to Device
Wrong Slave address or Slave device Busy
```

Also, quite often the board would just hang, after some manipulation regarding the SCL or SDA pins, even so subtle as connecting or removing an oscilloscope probe. If the board would still be up, an error would appear on running any I²C-related program:

```
81600000.i2c #0: Could not talk to device 0x70 (-1),
business always busy, trying to reset
```

I connected the display to the AVR microcontroller board (atmega48-based) that I originally built the device for, and ran a program writing random data to the display. It worked and the display lit up. I took a photo of the oscilloscope screen showing the SDA and SCL lines (see figure 4.3, image on the left).

![Oscilloscope images of captured I²C signals. Impulses generated by FX12MM board on the left, by the AVR microcontroller on the right.](image.png)
I then connected the display to the FX12MM board. I wrote a program which would try to write a single byte to the display. The source is also presented in the Appendix A.4 on page 58. I also captured the oscilloscope image (also figure 4.3, image on the right) and immediately realized that the SDA and SCL lines were swapped. I reconnected them, the board hanged, but after a reboot I still wasn’t able to write to the display. I went back to take a closer look at the two graphs. The second observation was that the address on the left (0111000 in binary) looks like shifted by one to the left with respect to the address on the right side (00111000). Their respective values are 0x70 and 0x38 in hexadecimal. I replaced the old address with the 0x38 and now I was able to write to the display, and I could also read the written value back. It became clear to me that I made an error when calculating the address of the display, because according to the I²C specification the read/write bit should not be included in the address.

Now I was also able to use the i2ctools package:

```
-b0# i2cset -y 0 0x38 0x00
-b0# i2cget -y 0 0x38
0x00
-b0# i2cset -y 0 0x38 0xa5
-b0# i2cget -y 0 0x38
0xa5
```

Since I already learned how the CGI scripts work and because the thttpd server supported them, I decided to prepare a little demonstration. Arguments can be passed to the CGI program by separating them with a plus sign and appending to the URL using a question mark. The i2cwrite program required two arguments, first being the device’s address, and second being the value to be written to it. Because the thttpd server runs chrooted, which means it can not access files outside a specified path, I had to create a I²C device node inside the chroot path. After this was done, I was able to set the contents of the LCD display remotely, by using a web browser. An example link would be:

```
http://192.168.0.60/i2cwrite?0x38+0x45
```

and just visiting the link would put number 5 on the display. The mapping of the bits to the segments was arbitrary, so numbers were discovered by trial-and-error.

**n-XYTER tests**

I connected the I²C bus of the nXYTER to the FX12MM board and ran the i2cdetect utility. The 0x10 address was identified. To test whether the communication actually works I started writing values to the 18-th register called Vth, which stands for Voltage threshold. A change in the voltage level could then be read out on the pin named y-vth on the experimental board. I also created a program which was writing a value incremented from 0 to 255 to the register, and captured the output. The photograph can be seen in figure 4.4.

---

8Common Gateway Interface, a technique which allows to run programs remotely on a machine with a HTTP server installed.
4.5 LabView integration

This section is intended to show how my work can be used. It is not meant as an integral part of the presented solution, it just provides guidelines for someone who would like to continue development of this system. It is only meant to serve as an indication of direction, because especially the code is mostly testing, and not any final implementation.

I choose to provide a user interface for demonstration. For this purpose software part of the previous n-XYTER setup was used. The whole setup is partially described in [39]. The n-XYTER was interfaced by a Virtex-II FPGA chip which was pushing data to the PC over a USB 2.0-capable microcontroller, Cypress EZ-USB FX-2. The microcontroller also featured an I²C interface, over which the initial n-XYTER’s configuration was loaded. The controlling application was built using the National Instruments Lab View environment. It’s front panel can be seen in figure 4.5.

The visual instrument

The instrument’s default view allows the user to select individual channels that will be masked out by the chip readout circuitry. Contents of all I²C registers can be set using this panel. By clicking the Send button their contents are sent to the board over the USB link. Because LabVIEW does not directly support reading out the data from microcontroller connected over USB, a workaround was created. The EZ-USB FX-2 chip’s manufacturer, Cypress, is distributing a driver for Windows operating systems. Using this driver, functions for reading from, and writing to the FX-2 chip were designed, using example code provided by Cypress in Microsoft Visual
Figure 4.5: The front panel of the n-XYTER test setup LabVIEW visual instrument

C++ Studio. Those functions were enclosed in a DLL\(^9\) library which LabVIEW environment could then interface. My role in this previous project was merely to optimize the USB readout speed, and perform final testing. However small, it provided me with knowledge that I needed to attach my work presented in this thesis to the existing infrastructure.

**The DLL Library replacement**

I decided that the best place to connect to the existing software chain is the DLL library. I completely removed the USB-related functionality from it, and started dumping the data passed over to the `fnUsbWrite()` to a file. The modified function is presented below.

```c
_declspec (dllexport) long fnUsbWrite(TD1Hdl bufout) {
    uInt16 bufo[256];
    int nBytes = 512;

    // copy the data from the LabVIEW structure to a char array
    memcpy(bufo,(*(uInt16 **)(bufout)+2),nBytes);

    dump_to_file((char*)bufo, nBytes);

    return nBytes;
}
```

\(^9\)Dynamic Link Library, Microsoft’s implementation of the shared library concept.
The logic for compiling the data packets in LabVIEW was complicated, and it took me some time to figure it out. Afterwards, I started writing two programs for the testing purposes, the client and the server. The client part would later be enclosed in the DLL library, and the server program would be compiled and run on the FX12MM board. Server would listen for incoming connections, and write the data received over TCP/IP to the I2C-bus connected device, which for testing purposes was the 7-segment display.

Network operation

The dialogue for calling the \textit{fnUsbTcpSend()} library function from LabVIEW is show in figure 4.6.

![Figure 4.6: Call library function dialogue and respective functional element below (the one with orange top).](image)

The respective function, which replaced \textit{fnUsbWrite()} looked like this:

```
_declspec (d1lexport) long fnUsbTcpSend(TD1Hdl bufout)
{
    detectorRemote* det = detectorRemote::getMe();
    uInt16 bufo[256];
    int nBytes = 512;
    
    memcpy(buco,(*(uInt16 **)bufout)+2,nBytes);
```
The explanation of details of the implementation of the `detectorRemote` singleton class and TCP/IP network programming are beyond the scope of this thesis. Nevertheless, full sources together with a Visual Studio project are provided on attached compact disk for reference. The library’s mode of operation was such, that on the first invocation of the `fnUsbTcpSend()` function the detector object would be created, and connection to the board would be initiated. Next, a call to `sendI2Cdata()` would send the buffer passed by LabVIEW to the board. The buffer was 512 bytes wide, and included configuration for two n-XYTER chips, for a dual-side setup. On the board’s side, the server would receive the buffer. Only the first chip’s configuration would be used, and from that the register no. 18 specifically, since this was actually the only changing value. The register’s name is Vth, or Voltage threshold, and it’s used for setting the global threshold voltage for the comparators. It was the same signal that was used in testing previously. It’s value would be output to the display directly. This would not produce any meaningful result, but since all eight bits were represented on the display (the dot being the 8-th segment) a pattern for a binary counter could be observed, since the Vth parameter was incremented from 0 to 255. Also first 32 bytes of the packet were output to the terminal, for testing purposes. The code on the server side looked like this:

```c
while ( (bytes = recv(new_fd, buf, MAXBUF, 0)) > 0) {
    if (bytes == -1) {
        perror("Server-recv() error");
    }

    printf("Read: %d bytes!
", bytes);
    i2c_disp_write(buf[18]);
    for (i = 0; (i < bytes) && (i < 32); i++) {
        printf("0x\%hhx ", buf[i]);
    }
    printf("\n");
}
```

This concluded the testing, and showed that I²C bus control over the TCP/IP network from the existing LabVIEW application level is possible.
Chapter 5

Conclusions

The completion of this thesis required interdisciplinary skills, many of which I initially lacked. Working on it taught me plenty, from basics of the FPGA design, through detailed booting procedures and low-level operating systems design, up to CGI scripts execution, LabVIEW visual instruments and network programming. I was able to utilize and develop my three main interests in computer science, that is computer networks, micro-controllers and software engineering. I feel greatly enriched by the experiences I gathered, and I consider this to be a desirable closing for my Master of Science studies. Although the results will not be immediately useful, they provide a solid grounds for further research and a platform ready for extension and application. The factors that can positively impact future designs would include integrated circuits manufactured in higher density processes, featuring higher executions speeds and faster networking infrastructure. I believe that better hardware would increase the network throughput, as speeds approaching the theoretical bandwidth limit of 1Gbps of Gigabit Ethernet are beginning to be seen in embedded, Virtex-5 based devices.
Appendix A

Files

A.1 Terminal output of a successful system run

U-Boot 1.3.4-00326-g9abed00-dirty (Feb 11 2009 - 20:27:34)

### No HW ID - assuming Virtex-4 FX12 or FX20
DRAM: 64 MB
FLASH: 4 MB
In: serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
BOOTP broadcast 1
*** Unhandled DHCP Option in OFFER/ACK: 28
*** Unhandled DHCP Option in OFFER/ACK: 28
DHCP client bound to address 192.168.0.60
TFTP from server 192.168.0.1; our IP address is 192.168.0.60
Filename 'fx12mm.dtb'.
Load address: 0x1000000
Loading: #
done
Bytes transferred = 12288 (3000 hex)
TFTP from server 192.168.0.1; our IP address is 192.168.0.60
Filename 'kernel'.
Load address: 0x900000
Loading: #
done
Bytes transferred = 1601704 (1870a8 hex)
## Booting kernel from Legacy Image at 00900000 ...
   Image Name: Linux-2.6.28-rc6
   Image Type: PowerPC Linux Kernel Image (gzip compressed)
   Data Size: 1601640 Bytes = 1.5 MB
   Load Address: 00000000
   Entry Point: 00000000
Verifying Checksum ... OK
## Flattened Device Tree blob at 01000000
Booting using the fdt blob at 0x1000000
Uncompressing Kernel Image ... OK
Loading Device Tree to 007fa000, end 007fffff ... OK
Unable to update property /plb/opb:clock-frequency, err=FDT_ERR_NOTFOUND
Unable to update property /plb/ebc:clock-frequency, err=FDT_ERR_NOTFOUND
Unable to update property NOR mapping, err=FDT_ERR_NOTFOUND
Using Xilinx Virtex machine description
Linux version 2.6.28-rc6 (loko@ludolphine) (gcc version 4.0.2) #9 PREEMPT
Wed Feb 11 16:00:32 CET 2009
Zone PFN ranges:
   DMA 0x00000000 -> 0x00004000
   Normal 0x00004000 -> 0x00004000
Movable zone start PFN for each node
early_node_map[1] active PFN ranges
   0: 0x00000000 -> 0x00004000
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 16256
Kernel command line: console=ttyS0 root=/dev/nfs rw ip=on
Xilinx intc at 0x81800000 mapped to 0xfdfff000
PID hash table entries: 256 (order: 8, 1024 bytes)
clocksource: timebase mult[2800000] shift[22] registered
Console: colour dummy device 80x25
Dentry cache hash table entries: 8192 (order: 3, 32768 bytes)
Inode-cache hash table entries: 4096 (order: 2, 16384 bytes)
Memory: 61388k/65536k available (3192k kernel code, 4080k reserved, 128k data, 137k bss, 164k init)
Calibrating delay loop... 199.16 BogoMIPS (lpj=398336)
Mount-cache hash table entries: 512
net_namespace: 636 bytes
NET: Registered protocol family 16
PCI: Probing PCI hardware
NET: Registered protocol family 2
IP route cache hash table entries: 1024 (order: 0, 4096 bytes)
TCP established hash table entries: 2048 (order: 2, 16384 bytes)
TCP bind hash table entries: 2048 (order: 2, 16384 bytes)
TCP: Hash tables configured (established 2048 bind 2048)
TCP reno registered
NET: Registered protocol family 1
Installing knfsd (copyright (C) 1996 okir@monad.swb.de).
msgmni has been set to 120
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
io scheduler cfq registered (default)
Serial: 8250/16550 driver4 ports, IRQ sharing disabled
83e00000.serial: ttyS0 at MMIO 0x83e01003 (irq = 16) is a 16550A console [ttyS0] enabled
Welcome to the Virtex-4 FX12 MM Linux system.
b0 login:
A.2 DHCP files

The DHCP server configuration file (dhcpd.conf)

```plaintext
option domain-name "nz22.ifj.edu.pl";
option domain-name-servers 192.168.0.1;
option routers 192.168.0.1;
option broadcast-address 192.168.0.255;

default-lease-time 7200;
max-lease-time 86400;

authoritative;
ddns-update-style ad-hoc;

subnet 192.168.0.0 netmask 255.255.255.0 {
  range 192.168.0.100 192.168.0.150;
}

host b0 {
  option host-name "b0.nz22.ifj.edu.pl";
  option root-path "/tftpboot";
  next-server 192.168.0.1;
  hardware ethernet 00:de:ad:be:ef:00;
  fixed-address 192.168.0.60;
}
```

A.3 EDK files

startedk shell script

```plaintext
PLATFORM=‘uname -m’

if [ ‘echo $PLATFORM | grep "sun"’ ]; then
  PLATFORM=sol
elif [ $PLATFORM = "i686" ]; then
  PLATFORM=lin
elif [ $PLATFORM = "x86_64" ]; then
  PLATFORM=lin64
else
  PLATFORM=lin
fi

XILINX=/opt/Xilinx101/ISE
export XILINX
XILINX_EDK=/opt/Xilinx101/EDK
export XILINX_EDK
```
source ${XILINX}/settings32.sh

export LD_PRELOAD=/usr/local/lib/libusb-driver.so

if [ -n "$PATH" ]; then
    PATH=${XILINX_EDK}/bin/${PLATFORM}:
        ${XILINX_EDK}/gnu/microblaze/${PLATFORM}/bin:
        ${XILINX_EDK}/gnu/powerpc-eabi/${PLATFORM}/bin:${PATH}
else
    PATH=${XILINX_EDK}/bin/${PLATFORM}:${XILINX_EDK}/gnu/microblaze/
        ${PLATFORM}/bin:${XILINX_EDK}/gnu/powerpc-eabi/${PLATFORM}/bin
fi
export PATH

if [ -n "$LD_LIBRARY_PATH" ]; then
    LD_LIBRARY_PATH=${XILINX_EDK}/bin/${PLATFORM}:usr/X11R6/lib:
        ${XILINX}/bin/lin:${LD_LIBRARY_PATH}
else
    LD_LIBRARY_PATH=${XILINX_EDK}/bin/${PLATFORM}:usr/X11R6/lib:
        ${XILINX}/bin/lin/
fi
export LD_LIBRARY_PATH

myxilinxrc=${HOME}/.qt/xilinxrc

if [ -d "${SYSCONF}/xilinxrc" -a ! -f "$myxilinxrc" ]; then
    cp "${SYSCONF}/xilinxrc" "$myxilinxrc"
elif [ -f "/Xilinx/xilinxrc" -a ! -f "$myxilinxrc" ]; then
    cp "/Xilinx/xilinxrc" "$myxilinxrc"
fi

export DISPLAY=:0
exec xps

XMD genace.tcl options file (xupGenace.opt)

-jprog
-board user
-target ppc_hw
-hw implementation/download.bit
-elf /home/loko/cross/X/zImage.elf
-configdevice devicenr 1 icode 0xf5057093 irlength 16 partname xcf08p
-configdevice devicenr 2 icode 0x01E58093 irlength 10 partname xc4vfx12
-debugdevice devicenr 2 cpunr 1
-ace system.ace

Please note that the idcode parameter is unique for every Xilinx device, and this parameter had to be obtained by the author using the Impact tool, as there was
no documentation available on how to create this file for the FX12 MM board specifically.

A.4 I²C files

Presented sources are a modification of the original source available on the Xilinx Wikidot page on the Open Source Linux I²C driver[36].

i2cread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "i2c-dev.h"

int main(int argc, char** argv) {
    int dev_i2c;
    unsigned int eepromAddr;
    unsigned int argRead, bytesRead;
    unsigned char i2c_buffer[1];

    if(argc != 2) {
        printf("Usage: %s 0x<addr>\n", argv[0]);
        return(-1);
    }
    argRead = sscanf(argv[1],"0x%x", &eepromAddr);
    if(argRead != 1) {
        printf("Couldn’t scan address & val\n");
        return(-1);
    }

    printf("Reading from I2C PCF at 0x%X\n", eepromAddr);

    // Open and read i2c
    if ((dev_i2c = open("/dev/i2c/0", O_RDWR)) < 0) {
        printf("Cannot Open I2C Master Device\n");
        exit(1);
    }

    if (ioctl(dev_i2c, I2C_SLAVE, eepromAddr) < 0) {
        printf("Cannot set I2C address\n");
        exit(1);
    }

    // Read from device
```
byteRead = read(dev_i2c, i2c_buffer, 1);
if(byteRead != 1) {
    printf("I2C read failed. Return code = %u\n", bytesRead);
    exit(1);
}
printf("Read: 0x%x\n", (unsigned char)i2c_buffer[0]);
return(0);

/*
 * i2cwrite.c
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "i2c-dev.h"

int main(int argc, char** argv) {
    int dev_i2c;
    int bytesToRead;
    unsigned int eepromAddr;
    unsigned int value;
    unsigned int argRead, bytesRead;
    unsigned char i2c_ptr[2];
    unsigned char* i2c_buffer[1];

    if(argc != 3) {
        printf("Usage %s 0x<addr> 0x<value>\n", argv[0]);
        return(-1);
    }
    argRead = sscanf(argv[1], "0x%x", &eepromAddr);
    argRead = sscanf(argv[2], "0x%x", &value);
    if(argRead != 1) {
        printf("Couldn’t scan address & val\n");
        return(-1);
    }
    printf("Writing 0x%X to I2C addr 0x%X\n", value, eepromAddr);}

    // Open and read i2c
    if ((dev_i2c = open("/dev/i2c/0", O_RDWR)) < 0) {
        printf("Cannot Open I2C Master Device\n");
        exit(1);
    }

    return(0);"
if (ioctl(dev_i2c, I2C_SLAVE, eepromAddr) < 0) {
    printf("Cannot set I2C address\n");
    exit(1);
}

i2c_ptr[0] = (value) & 0xFF;
if (write(dev_i2c, i2c_ptr, 1) != 1) {
    printf("I2C write failed\n");
    exit(1);
}

return(0);
Bibliography


