CHALMERS

# Transferring MOST data over Ethernet

*Master of Science Thesis in Integrated Electronic System Design*

Holmström, Christoffer
Persson, Martin

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, March 2010

Transferring MOST data across Ethernet

Christoffer Holmström

Martin Persson

Examiner: Arne Linde

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

SE-412 96 Göteborg

Sweden

Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering

Göteborg, Sweden March 2010

# Abstract

The purpose of this Master's thesis is to provide a feasibility study of encapsulating and transmitting data from an FPGA for transfer across Ethernet, with the encapsulation done with a microcontroller. The data rate required is 53 Mbit/s, specified by the MOST50 standard which will be the source of the data in the final product. In addition to the streaming capabilities, basic control functionality should also be implemented. This thesis was initiated by FYI Communication who already has a product, called MCBuster, whose functionality and performance should be enhanced.

It was decided that neither TCP nor UDP would be able to fulfill the transport layer requirements for the project. Therefore, a custom protocol called FYI CETP was developed on top of UDP. For performance reasons, the entire network protocol stack was developed from the ground up.

To transfer data from the FPGA to the microcontroller fast enough, a parallel interface was used, called External Peripheral Interface in the Luminary LM3S9B96 microcontroller used. The FPGA, a Lattice XP2-8, is interfaced in much the same way as an external memory would be, with separate data and address buses.

For the receiving end of the data stream, a heavily threaded logging and control application called PCDump was developed using Java.

The project has fulfilled this goal with a broad margin, achieving a sustained transfer rate of 73 Mbit/s.

# Preface

This document is the final report of a Master's thesis in Electrical Engineering at the department of Computer Science and Engineering at Chalmers University of Technology in Göteborg, Sweden. The examiner and Chalmers' tutor has been Arne Linde at the department of Computer Science and Engineering. The thesis work was conducted by request from and in cooperation with FYI Communication in Göteborg, Sweden. Tutors from FYI have been Björn Bergholm and Tomas Forslund. Office space was graciously supplied by Broccoli Engineering, Göteborg, Sweden.

We would like to give a special thanks to Björn and Tomas at FYI, to Broccoli Engineering for using their offices, to Alain Aublet for valuable input on our PCB design, to ETA for the use of their equipment when designing and constructing the PCB, to Andreas at Broccoli for lending us a computer with a parallel port, and to Arne for his invaluable help and guidance. Last but not least we would like to thank our friends and family for their support and understanding during this time.

# Table of Contents

## Table of Figures

# 1 Abbreviations

| | |
|---|---|
| ARP | Address Resolution Protocol |
| BGA | Ball Grid Array |
| CAN | Controller-Area Network |
| CPLD | Complex Programmable Logic Device |
| CPU | Central Processing Unit |
| CSMA/CD | Carrier Sense Multiple Access with Collision Detection |
| DDR | Double Data Rate |
| DMA | Direct Memory Access |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| EMAC | Ethernet MAC |
| EMAC PHY | EMAC Physical Layer |
| EPI | External Peripheral Interface |
| FIFO | First In First Out |
| FPGA | Field-Programmable Gate Array |
| FYI CETP | FYI  Custom Ethernet Transmission Protocol |
| GUI | Graphical User Interface |
| IDE | Integrated Development Environment |
| IP | Internet Protocol |
| JTAG | Joint Test Action Group |
| LAN | Local Area Network |
| MAC | Media Access Control |
| MIPS | Million Instructions Per Second |
| MOST | Media Oriented Systems Transport |
| NDP | Neighbor Discovery Protocol |
| OS | Operating System |
| OSI | Open Systems Interconnection |
| OTG | On The Go |
| PCB | Printed Circuit Board |
| PLL | Phase Lock Loop |
| RTOS | Real-Time Operating System |
| SPI | System Packet Interface |
| SRAM | Static Random Access Memory |
| SWD | Serial Wire Debug |
| TCP | Transmission Control Protocol |

| | |
|---|---|
| TQFP | Thin Quad Flat Package |
| UART | Universal Asynchronous Receiver/Transmitter |
| UDP | User Datagram Protocol |
| USB | Universal Serial Bus |
| VHDL | Very high speed integrated circuit Hardware Description Language |

# 2 Background

What the modern automotive industry designs is a highly integrated and complex product with many subsystems that need to interact to a certain degree. For the multimedia and infotainment systems that are becoming more and more prevalent, the MOST standard was designed. The first generation MOST standard specifies a transmission speed of "circa 25 Mbit/s" (for the MOST25 standard). The more recent MOST50 standard specifies a speed of "circa 50 Mbit/s", and a third standard, MOST150 calls for speeds of 150Mbit/s (MOST, 2008, page 24, page 197).

FYI Communication already has a MOST analyzer on the market, called 'MCBuster', which is based around an FPGA analyzer core. This analyzer is mostly used when new products using the MOST bus are developed. This device translates from the MOST bus to CAN, RS-232 and USB 2.0, but only in the lower speed mode 'Full Speed' (limited to 12 Mbit/s). However, the available downlink speed from the MCBuster through these interfaces is lower than the MOST25 specification it was designed for. This is not a problem for the device itself as it filters messages and only delivers select parts of the MOST stream downstream.

Since the release of the MCBuster, the MOST50 specification has been developed. This, in addition to the desire to be able to analyze all of the data on the MOST bus, is what has lead to this thesis.

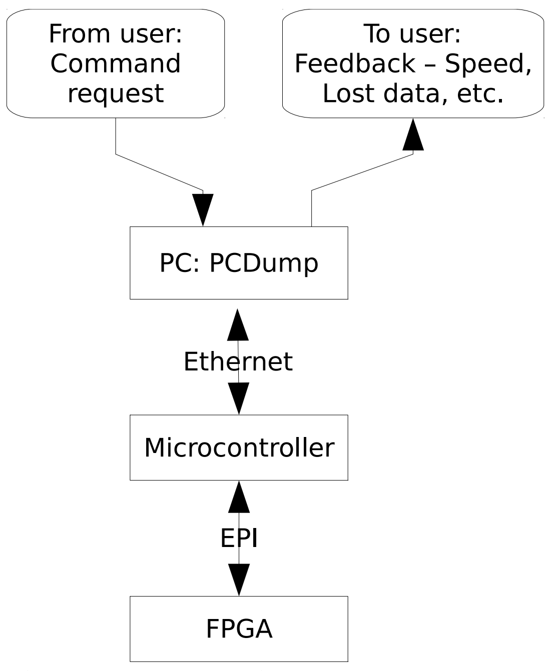
The company's desire is to be able to transmit the entire MOST50 datastream downlink through Ethernet. In addition to this, it is desired for the next-generation MCBuster to be able to actively send data on the MOST bus. Currently, there are no other products on the market operating at this speed. This, combined with the challenge of reaching the high performance needed, makes this project viable as a Master's thesis.

## 2.1 Purpose

The goal of this Master's thesis work is to develop a proof of concept for a transmission system from an FPGA to a PC operating at more than 53 Mbit/s, according to specification from FYI, as well as 500 kbit/s upstream for control messages.

An actual MCBuster will not be used as the source for the data but rather a simpler test source will be developed. This is partly due to the proprietary nature of the current MCBuster and partly due to ease of use - it was considered easier to design a tester from scratch rather than modify the MCBuster for MOST50 operation and connect it to a MOST50 test bed.

To accomplish this goal, FYI wanted to use a microcontroller for future extensibility and due to the availability of well-documented and thoroughly tested interfaces on the microcontroller. Implementing the design in a microcontroller is also considered to have a shorter development time compared to other options. Thus, the overall layout of our work will be as shown in Figure 1.


From user: Command request
To user: Feedback – Speed, Lost data, etc.
PC: PCDump
Ethernet
Microcontroller
EPI
FPGA
*Figure 1: System overview*

## 2.2 Delimitations

The main delimitation of this project has already been mentioned - not using the actual MCBuster as a source for the data, but instead using a much simpler data source.

Another delimitation is that we will only attempt to implement the snooping MOST50 part – actively sending MOST25 data is beyond the scope of this thesis.

Data will only be logged or discarded, not handled by the PC in any other way.

# 3 Theory

This chapter will give a brief overview of some of the topics covered in this report. It is assumed that the reader has some previous experience with microcontrollers and FPGA's.

## 3.1 Ethernet

Ethernet is a family of Local Area Network (LAN) technologies. The IEEE 802.3 standard defines Ethernet using the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) protocol (Leon-Garcia & Widjaja 2004, 384-387 and page 427-428). This standard was first issued in 1985 but is still being revised occasionally. The protocol covers the physical layer and the data link layer of the Open Systems Interconnection (OSI) model. It is placed in the link layer of the Internet Protocol Suite. CSMA/CD basically works as follows:

Carrier Sense: First listen until the channel is not busy.
Multiple Access: When the channel is not busy start sending the data.
Collision Detection: If a collision is detected the transfer is aborted. Wait a quasi-random time and try to send again.

If a collision is not detected after 2 times the longest propagation delay time of the network, the transfer continues until all data is sent.

The header consists of seven fields; preamble, start-of-frame delimiter, destination address, source address, length/type and data followed by frame check sequence. The header is 26 bytes excluding the payload data. The payload data may be a maximum of 1500 bytes in common implementations.

Different Ethernet standards support different speeds; 10Mbit/s Ethernet, Fast Ethernet (100Mbit/s), Gigabit Ethernet, 10-gigabit Ethernet and even 40 and 100 gigabit Ethernet (IEEE 802.3, last updated 2010).

## 3.2 IP

The Internet Protocol (IP) is the Internet layer protocol in the Internet Protocol Suite and is referred to as the network layer in the OSI reference model. The purpose of the protocol is to move datagrams through an interconnected set of networks (DOD Standard Internet Protocol, 1980, page 6). It carries, among other information, the Internet address and a fragment offset for large packets. The version that is mostly used is IPv4 but the use of IPv6 is increasing.

The IP header consists of at least 12 fields plus options, which comes to a minimum length of 20 bytes. The fields are: version, IP header length, type of service, total length, identification, flags, fragment offset, time to live, protocol, header checksum, source IP address, destination IP address, options and padding.

## 3.3 UDP

The User Datagram Protocol (UDP) or, alternatively, Universal Datagram Protocol, is a simple protocol in the Internet Protocol suite (Postel. J, 1980, page 1-3). UDP is part of the transport layer. The header consists of four fields: source port, destination port, length and checksum, each of 2 bytes size. The protocol does not perform any type of handshaking between the sender and receiver. This means that data may appear malformed, lost or out of order during transmission without being noticed. The protocol assumes that the error checking is performed by the application. Compared to TCP, UDP is used when it is preferable to loose data instead of resending it. This may be the case in applications that have higher demands on speed than on totally correct data packets, and where this loss can be tolerated, for example in streaming media.

## 3.4 ARP

The Address Resolution Protocol (ARP) is a link layer protocol of the Internet Protocol Suite. This protocol is used to determine a lower layer address from a higher layer address (Plummer. C, 1982, page 1-5). For example, ARP is used to retrieve the Ethernet MAC addresses when the IPv4 address is known. ARP is used for IPv4 while Neighbor Discovery Protocol (NDP) is used for IPv6.

The header consists of 9 fields with a total size of 28 bytes. The fields are: hardware address space, protocol address space, byte length of each hardware address, byte length of each protocol address, operation code (opcode), hardware address of sender of this packet, hardware address of target of this packet and protocol address of target.

When a MAC address is unknown a user may send a request (opcode = request) using ARP, asking a specific IP address what its MAC address is. The owner of the IP address – if it exists – will send a reply (opcode = reply) with the corresponding MAC address. The pairs of IP address and MAC address will be saved in a table used by the ARP protocol. If the pair already is in the table it is not necessary to send a request to get the MAC address. Once the MAC address is known it can be used for e. g. sending IP packets over Ethernet.

## 3.5 Direct Memory Access

Direct Memory Access (DMA) is a method used in processors to do reads and writes to memories and peripherals without using the CPU for every memory access. This has the advantage that a DMA transfer can run in the background while the processor is free for other operations. The CPU is used to set up a transfer and to start it. When it is finished it will generate an interrupt. DMA is useful in applications with high demands of performance such as data streaming (Null & Lobour, 2006, page 336).

In addition to simple copy transfer modes used with DMA, a more advanced mode called scatter-gather is available. When using DMA with normal transfer mode it is only possible to set up starting address and size of the transfer. With scatter-gather, it is possible to send data to and from several memory and/or peripherals' addresses.

## 3.6 Non-blocking Reads

The microcontroller LM3S9B96 that is used in the project supports non-blocking reads when using the external peripheral interface. A normal peripheral read will stall the processor until the memory access is finished. On the contrary a non-blocking will read run in the background until it is finished after it is configured and started (Luminary Datasheet, 2009, page 356-357).

## 3.7 Watchdog Timer

A Watchdog timer is a device used to reset the processor if the program hangs (Kamal. R, 2009, page 37-38). A counter running in the background of the program is reloaded at appropriate times when the program works normally. However, if the program gets stuck, for example in a while loop, the counter will not be reloaded. When the counter reaches 0 a first interrupt will occur. This can, for example, be used to softly (without reset) restore the system to a working state or to save information into non-volatile memory for storage through a reset. The counter is then reloaded with a value. If a second interrupt occurs it may generate a software reset, depending on if this feature is enabled or not.

## 3.8 MOST

The Media Oriented Systems Transport (MOST) is an automotive bus standard for media and infotainment data (MOST, 2008, page 23-25). The bus differs from other standards, e. g. CAN, in the high speed. MOST25 and MOST50 run at speeds of 25 Mbit/s and 50 Mbit/s. A new version called MOST150 will run at 150 Mbit/s. The standard defines all seven layers of the OSI model. The MOST cooperation sets the standard and consists of 16 car makers and about 60 suppliers. It is not an open standard.

## 3.9 Method

The project work was divided into five parts. The four first are centered around one communication direction between two modules, and the last is the complete system integration. Each of the phases are described briefly here. For a more complete description, please refer to the following sub-chapters.

### *3.9.1 Phase one*

In the first phase, the main development of the communication stack in the microcontroller was done. This included implementing Ethernet, ARP, IP, UDP and the proprietary protocol FYI CETP (Custom Ethernet Transmission Protocol). In addition to this, the microcontroller was able to stream dummy data to the PC application.

### *3.9.2 Phase two*

The second phase included adding control mechanisms that allowed the PC to control the microcontroller. For example, this can be used to stop and start the microcontroller's stream.

### 3.9.3 Phase three

In the third phase, the PCB for the FPGA was developed. After this was tested and fully functional, it was connected to the microcontroller. At the end of this phase, the microcontroller could read data from the FPGA. Because the same buffers were used for reading data into the microcontroller as those used for sending data to the PC, data could also be sent from the FPGA all the way to the PC in this phase.

### 3.9.4 Phase four

The task in the fourth phase was to send instructions from the microcontroller to the FPGA.

### 3.9.5 Phase five

The final phase saw the integration of all previous phases. The new functionality implemented was being able to send commands from the PC to control the FPGA. After this was accomplished, code optimizations were conducted to improve performance, and the entire system was put through a rigorous testing to nail down any remaining bugs.

## 3.10 Feasibility

Before any of the project work was started a feasibility study was conducted to ensure that the system was possible to design according to the specification.

All of the applicable microcontrollers had a clock frequency of approximately 80 MHz and could execute most instructions in a single cycle, or in some cases slightly more. For the calculations, it was assumed that the processor could execute 80 million instructions each second.

According to the specification, at least 50 Mbit of data need to be transmitted each second. This is equal to 6.25 MB of data per second. A single UDP datagram can contain 1468 bytes (after 4 bytes are removed for the proprietary FYI CETP header). Thus, one datagram needs to be sent every 224 µs. With 80 million instructions per second, this gives 17920 cycles per datagram. Broken down to cycles per sent byte, this gives 12.2 cycles per byte. This shows that the requirement is quite strict but possible to achieve.

# 4 System description

This chapter will describe how the system was implemented. The design choices that were made will be discussed in chapter 5.

## 4.1 Overview

The implemented system is at first glance quite simple. There are three major components, as can be seen in Figure 1 on page 4 – the PC with PCDump, our logging and control application; the microcontroller which requests data from the FPGA and encapsulates it for transport across Ethernet; and the FPGA which produces data. A user (be it a person or another application) sends requests that control what the microcontroller does. The microcontroller in its turn controls the functions of the FPGA.

## 4.2 PCB Design

The PCB that was designed is as simple as possible with the shortest possible trace lengths from microcontroller to FPGA in order to reduce noise. This desire resulted in a sandwich-like construction.

The PCB includes the FPGA and three different voltage regulators to provide the necessary voltages (1.2V, 2.6V and 3.3V). Power to these regulators is drawn from a 5V rail from the microcontroller PCB, eliminating the need for a separate power supply. There is a button for reset, LEDs for output indication, and a power supply monitor to provide FPGA resets on power dips and power-up. The PLL of the FPGA requires an extra stable power source, which necessitated an additional, well-filtered rail. In addition to this, there are plenty of decoupling capacitors. A crystal along with the same clock generator as on the original MCBuster is used.

A total of five pin headers of different sizes are provided: One is used to allow the connection of external power supplies and selective connection of the on-board ones to the rest of the circuitry. Two are used for configuration options, and the last two headers are provided for different programming connectors. Finally, there is a female pin header connector on the reverse side which connects to the microcontroller .

In addition to these basic functions, a DDR memory was included at the request of the thesis initiators. Using a DDR memory also requires matching resistors. These were chosen according to common design practice. Additional matching resistors were placed on the traces leading to the microcontroller connector.

The schematic and PCB layout was created using Eagle 4 Professional. To provide a stable, low-impedance power supply across the entire board, especially to the DDR memory, a four-layer PCB was used. This PCB has a split 3.3 / 2.6 V power plane and a GND plane as internal planes, with the outer layers used as signal routing layers.

## 4.3 FPGA

The FPGA used is a LFXP2-8E-5TN144C in a TQFP144 packet from Lattice Semiconductor, hereafter referred to as the XP2.

The PCB provides two separate clocks to the device, but the application is clocked with a third clock, the EPI clock from the microcontroller that is provided with the EPI bus, running at half the microcontroller core, thus 40 MHz. This setup has worked fine without any known issues.

### *4.3.1 The FPGA Code*

The FPGA code is used to simulate data flow to the microcontroller and some simple communication between the devices. Two modes of operation is supported: A continuous stream of data, and a peaks and valleys case when there is no data intermittently. The data output is simply the value of a counter that is incremented after each write.

The I/Os of the application is clock, reset, epi_data (16 bits), epi_adress (11 bits), epi_iRDY, epi_rd and epi_wr. See Figure 2. Five LEDs were also used during debugging. Among the interesting internal signals are: counter, counter_step and irdy_trig. The signal counter is 1 byte wide and holds the value of the counter (integer values 0-255). The signal counter_step is 1 byte wide and holds the increment value of the counter (0-255). At reset the counter is set to 0 and counter_step is set to 1. irdy_trig is used to trigger a pause in the data flow, simulating an out-of-data condition.

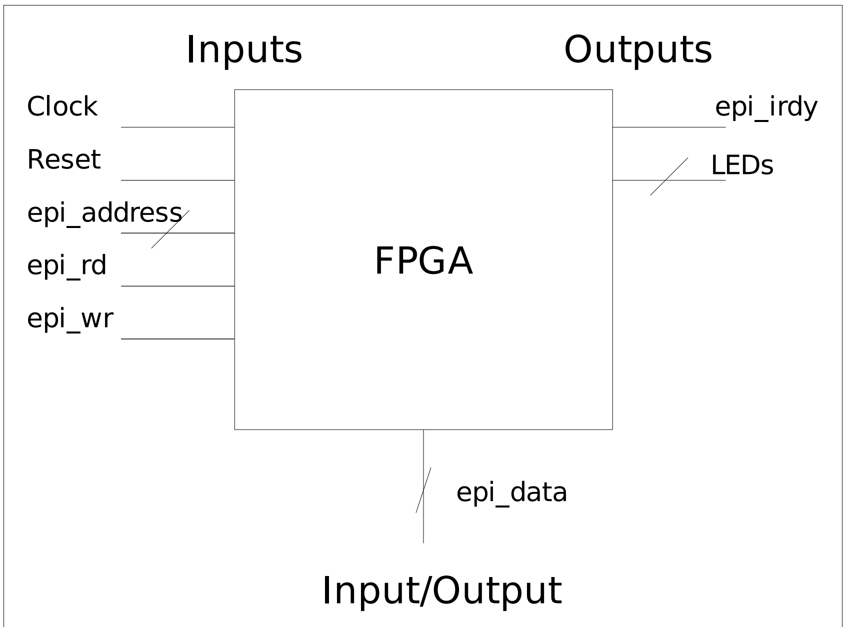


*Figure 2: The I/O's of the FPGA application*

Data requests and command messages to the FPGA are controlled by the epi_rd and epi_wr enable signals. See Figure 3.

Data is sent to the bus when the microcontroller sends a request by setting the epi_rd signal high. The data that is written to the data bus is the value of counter and counter plus counter_step. Since the data bus has a width of 2 bytes and counter is only 1 byte wide, two values are written simultaneously. After each write, the counter is increased with the chosen value of counter_step multiplied by 2. The counter simply wraps when it reaches a value above 255.

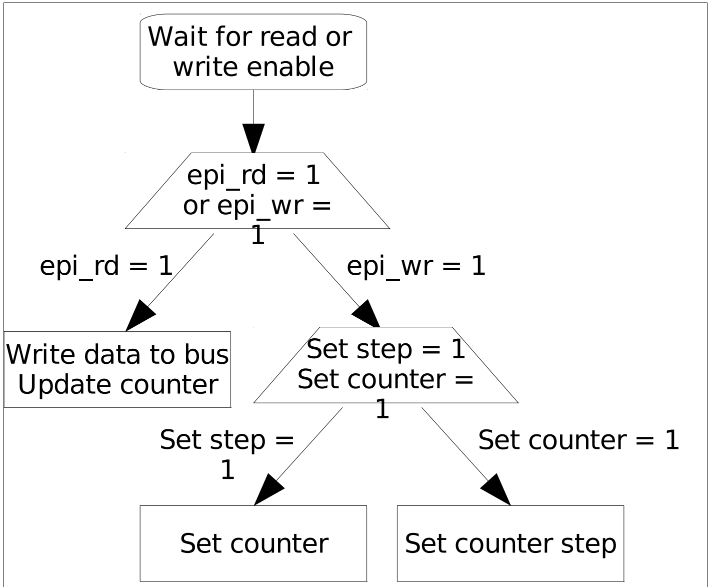


*Figure 3: The EPI read and write enable signals control the application. A read enable write data to the bus. A write enable read setup commands from the microcontroller*

Setup commands are sent to the FPGA when the microcontroller sets the epi_wr signal high. The command that was sent is read from the epi_address bus, and if applicable, a possible value of the command is read from the epi_data bus. The used commands are: Set counter and Set counter increment. Set counter sets the value of the counter signal. Set counter increment sets the value of the counter_step signal.

The epi_irdy output is an enable signal for the EPI controller in the microcontroller. The signal has to be high for the EPI controller to be allowed to read and write.

The code includes a state machine that runs in parallel with the write and read enable signals which sets epi_irdy low for a given time. See Figure 4. This pause is triggered when irdy_trig goes high. The length of the pause is set by the Timeout signal. The signal Pausecnt is used as a counter during the pause. This has been used during testing to simulate that the FPGA has run out of data which disables the read and write functionality of the EPI controller. In the end system used for maximum speed testing the pause is never triggered.

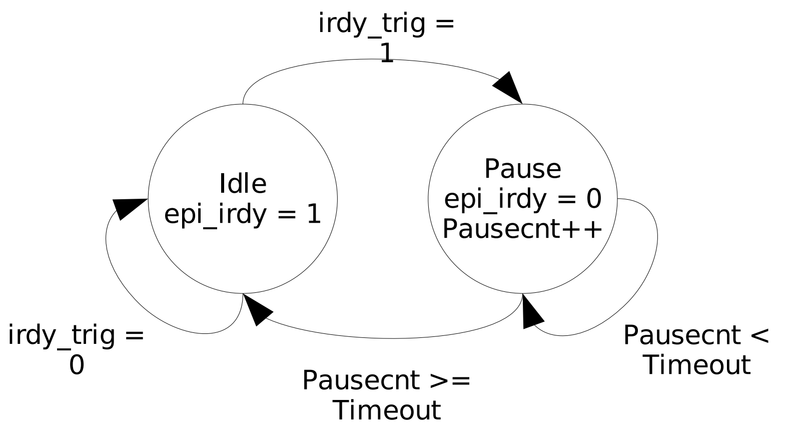


*Figure 4: The statemachine was just to simulate when the system is out of data. The irdy signal goes low after a given time during a given time*

## 4.4 Interface: FPGA to microcontroller

The interface between the FPGA and microcontroller consists of 32 pins, as seen in Figure 5. Among these are 16 data pins and 11 address pins. Two pins are enable signals from the EPI interface: one for read and one for write. One pin is an enable signal which is an input for the EPI controller. The EPI controller also has the EPI clock as an output. The last pin is not used but could be configured as a general purpose IO pin or as an EPI Frame pin.

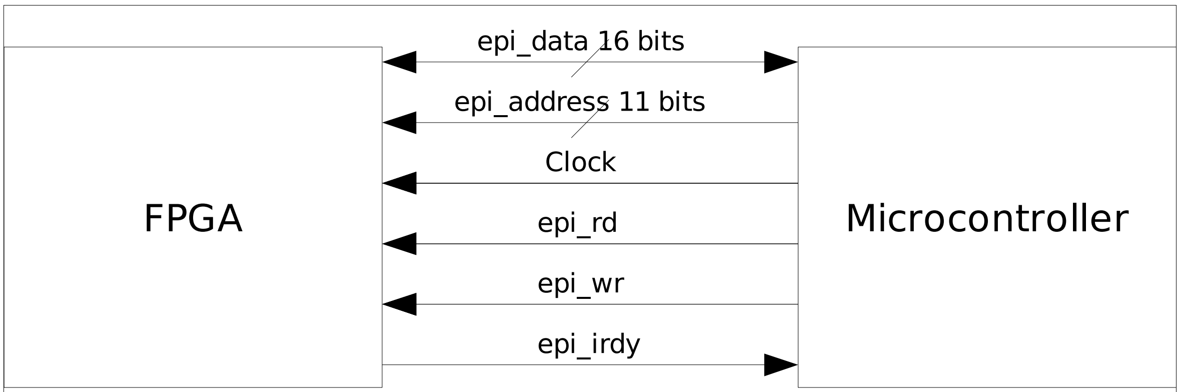


*Figure 5: The interface between the FPGA and microcontroller consists of data, address, epi clock, read and write enable and an enable signal for the EPI controller, the irdy signal*

## 4.5 Microcontroller

The microcontroller that is used is a Stellaris LM3S9B96 microcontroller from Luminary/Texas Instruments. Among its most important features are:

| | |
|---|---|
| Processor core: | ARM Cortex -M3 |
| Core frequency: | 80 MHz, resulting in approximately 100 DMIPs |
| External memory interface: | Up to 32-bit parallel bus |
| SRAM: | 96 KB |
| Flash Memory: | 256 KB |
| EMAC: | Yes, Ethernet MAC PHY |
| CAN: | 2 controllers |
| USB: | 2.0 OTG/Host/Device |
| UART: | 3 |
| SPI: | 2 |
| DMA: | Yes |
| DEBUG Interface: | JTAG and SWD |
| Package: | 100-pin LQFP |
| Temperature Range: | Industrial (-40° C to +85° C) |
| OS: | FreeRTOS supports ARM cores. |
| Sleep mode: | Yes |
| Availability: | The microcontroller was planned to be in production in March 2010, but has been delayed. A development board with a microcontroller of revision B was used. It has some bugs that should be removed in later revisions. |

The development board used is the Stellaris LM3S9B96 Development Kit. Its most important features are:

Dedicated connector to Ethernet MAC+PHY, USB, CAN, SPI and UART
1 MB Flash Memory
MicroSD card slot
USB cable for debugging (on-board JTAG controller)

## 4.6 Microcontroller C program

The C program reads data from the EPI interface into a buffer, writes it with DMA to the EMAC (Ethernet Media Access Controller) and sends it over Ethernet. The program also replies to ARP requests, reads messages from the PC and sends setup commands to the FPGA.

The data is handled in the size of full Ethernet frames, excluding the header information. This is 1468 bytes. A buffer with three slots of data is used to enable writing and reading simultaneously. Figure 6 Is a simplified flowchart showing how the main program works. The stream flag must be set for the program to keep a stream going. If a slot is empty – start an EPI read into this slot. If a slot is non-empty – start a Ethernet write from this slot. If a read was started, calculate the number of read bytes. See Appendix A.i for a more elaborated flowchart.

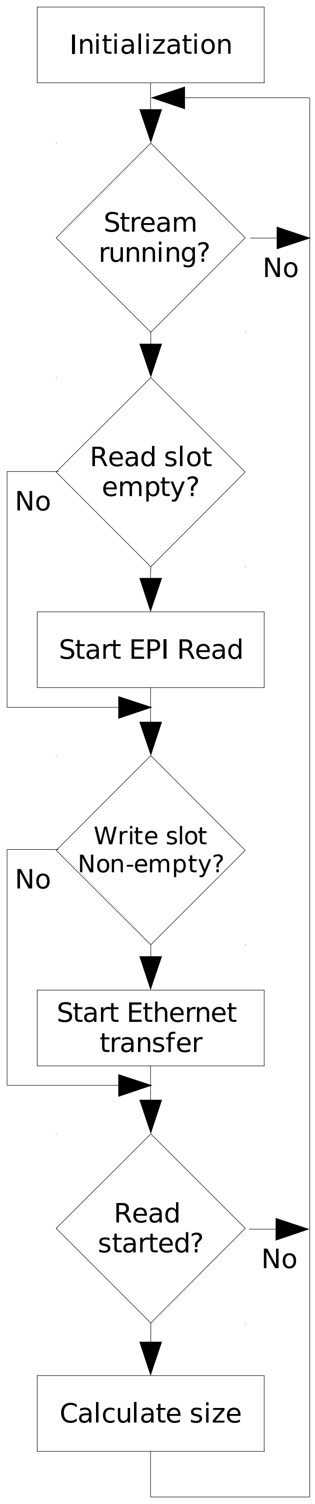


*Figure 6: A simplified flowchart how the main loop works. Data is read from FPGA and sent over Ethernet*

### *4.6.1 Initialization*

At startup, initialization of the microcontroller and all the peripherals units will occur. The microcontroller sends an ARP request to the target IP to get its MAC address. After the ARP response has been received, the programs main loop starts. See Figure 6.

### *4.6.2 EPI Read*

If the FPGA has any data the epi_irdy signal will be set high. If there is an empty slot, an EPI read is started. Non-blocking reads are used which run in the background while other processes are performed. One full frame of data is read unless a timeout occurs. This is tested in "Calculate size" in the above figure. If a timeout has occurred it means that the FPGA has run out of data (the epi_irdy signal has gone low). If this happens the running read stops and the number of read bytes will be calculated.

### *4.6.3 Ethernet Write*

After an EPI read has started, an Ethernet write starts if there is an non-empty slot in the buffer. If the slot is full, DMA will be used to write to the EMAC. During the DMA interrupt the packet is sent. DMA runs in the background while the program continues with other processes. If the packet is smaller than full it means that the FPGA has run out of data. These packets are sent without DMA because it is easier and maximizing performance is not necessary since there is no more data. After each write the packet counter in the FYI CETP header is updated.

### 4.6.4 Incoming Ethernet packets

When an Ethernet packet is received an interrupt is triggered. The program handles ARP requests and commands from the PC. See chapter 4.7 for more details of the communications between the microcontroller and the PC.

### 4.6.5 ARP request

During testing, it was found that the target PC seemed to forget the MAC address of the microcontroller, thus sending out a new ARP request. The program will identify this incoming packet and reply to it between “Start EPI read” and “Write slot non-empty?” in Figure 6.

### 4.6.6 PC commands

The program handles commands as specified in chapter 4.7. If the program receives a setup command from the PC it will be performed after “Calculate Size” in the above flowchart, Figure 6. The command is sent to the FPGA with the EPI controller.

### 4.6.7 Watchdog timers

The program has some while loops that are used as wait statements. For example, there is a wait for previous DMA transfer to finish and a wait for previous Ethernet Transfer to finish. The program should theoretically never get stuck but this cannot be guaranteed. Therefore, a watchdog timer is used to reset the program after a possible timeout. The timeout is set to 8 seconds which is the time it takes to fill a memory of 50 MB at the speed of 50 Mbit/s. Before reset, the stream flag and the packet counter will be saved in a non-volatile memory, in this case Flash. Furthermore, a watchdog flag is set and saved. At restart this information will be read from Flash. If the watchdog flag is set, the packet counter and stream flag are updated with the values stored in flash. An error message is sent to the PC before the main loop resumes.

This functionality implies that up to the three frames stored in the microcontroller may be lost.

## 4.7 Interface: PC to microcontroller

The interface between the microcontroller and the PC was specified to be Ethernet and to be able to run at 53 Mbit/s. The actual implementation was not specified. After finding both TCP and UDP unsuitable, a new custom protocol was developed, FYI CETP.

FYI CETP is used on top of UDP and provides two different modes: COMMAND and STREAM. COMMAND mode is used to control the microcontroller from the PC, while STREAM mode is used to signify a stream from the microcontroller. The STREAM mode allocates 16 bits as a counter, thus, 65536 packets can be sent before wrapping this counter. This counter is used to identify that no packets have been dropped, as well as to try to sort packets that have arrived out of order. One bit in the header is used as an inline flag to indicate that a COMMAND has been received with inline priority.

Basic transmission control is provided in COMMAND mode by using an ACK-ACK scheme. The source sends a command. When the target correctly interprets this command, it replies with an

ACK. The source receives this ACK and replies with its own ACK. Not until the target receives this ACK is the command actually performed. If, for some reason, either side is unable to understand the received command, it replies with a NACK, which prompts the other side to re-send its last command (up to a maximum of three times).

The three commands that can be sent to the microcontroller are COMMAND START which will start a stream, COMMAND STOP which will stop a running stream, and COMMAND SETUP which will forwarded setup data to the FPGA. The commands can be sent with four different priorities that indicate how the microcontroller should handle them.

The microcontroller will send the COMMAND ERROR to the PC if it has hanged and generated a watchdog interrupt.

For a more detailed description of the protocol, please refer to Appendix B.

## 4.8 PC Application

The PC application developed is as simple as possible, since it is not really in the scope of the project but rather a requirement for the testing of the system. The application, called PCDump, is developed using Java and relies on a module-based producer-consumer scheme with one worker thread performing a more or less complicated task in each module. Between the modules, FIFO queues are used to compensate for peaks in incoming data rate and operating system scheduling effects. For an overview of this system, please refer to Figure 7 below, and its explanation in chapter 4.8.1.

In short, PCDump communicates with the microcontroller according to the user's requests, checks the incoming data stream for out-of-order data and missing data, as well as measures the stream's speed.

PCDump is capable of sending all defined FYI CETP commands and also some unsupported ones in order to provide a debug tool for the microcontroller and the communication protocol that is as complete as possible. For more information, please refer to chapter 4.7.

### *4.8.1 Detailed description*

In Figure 7, each module is shown as a shape, and the buffers between them are arrows.

The UDP in module listens to the network interface for incoming datagrams to the specified port (30000). This data is stored in raw form together with an arrival time stamp. Due to the high speed, the UDP receive buffer size in the receiving computer has to be increased. The received data is interpreted according to the FYI Custom Ethernet Transmission Protocol (CETP) in the next module.

The data stream is then split and filtered to produce one stream of COMMAND datagrams and one stream of STREAM datagrams.

The STREAM stream is passed through the reorderer module, which attempts to organize the packets in their proper order according to the FYI CETP header. Any missing data is also detected here. The resulting stream is then split and sent to a file logger module and to the timing module. The timing module calculates an average speed and passes this on to the graphical user interface

(GUI).

The COMMAND stream is delivered to the command module, which is responsible for maintaining the communication standard specified by FYI CETP. This module also receives command requests from the GUI and reports communication status back to the GUI. The main output of the command module is the outgoing COMMAND datagrams, which are duplicated to on-screen logging and to the UDP out module, which is responsible for the actual transmission of the datagrams. The on-screen logging module also receives incoming COMMAND datagrams, and there is a file logger module to log all COMMAND packets to disk.

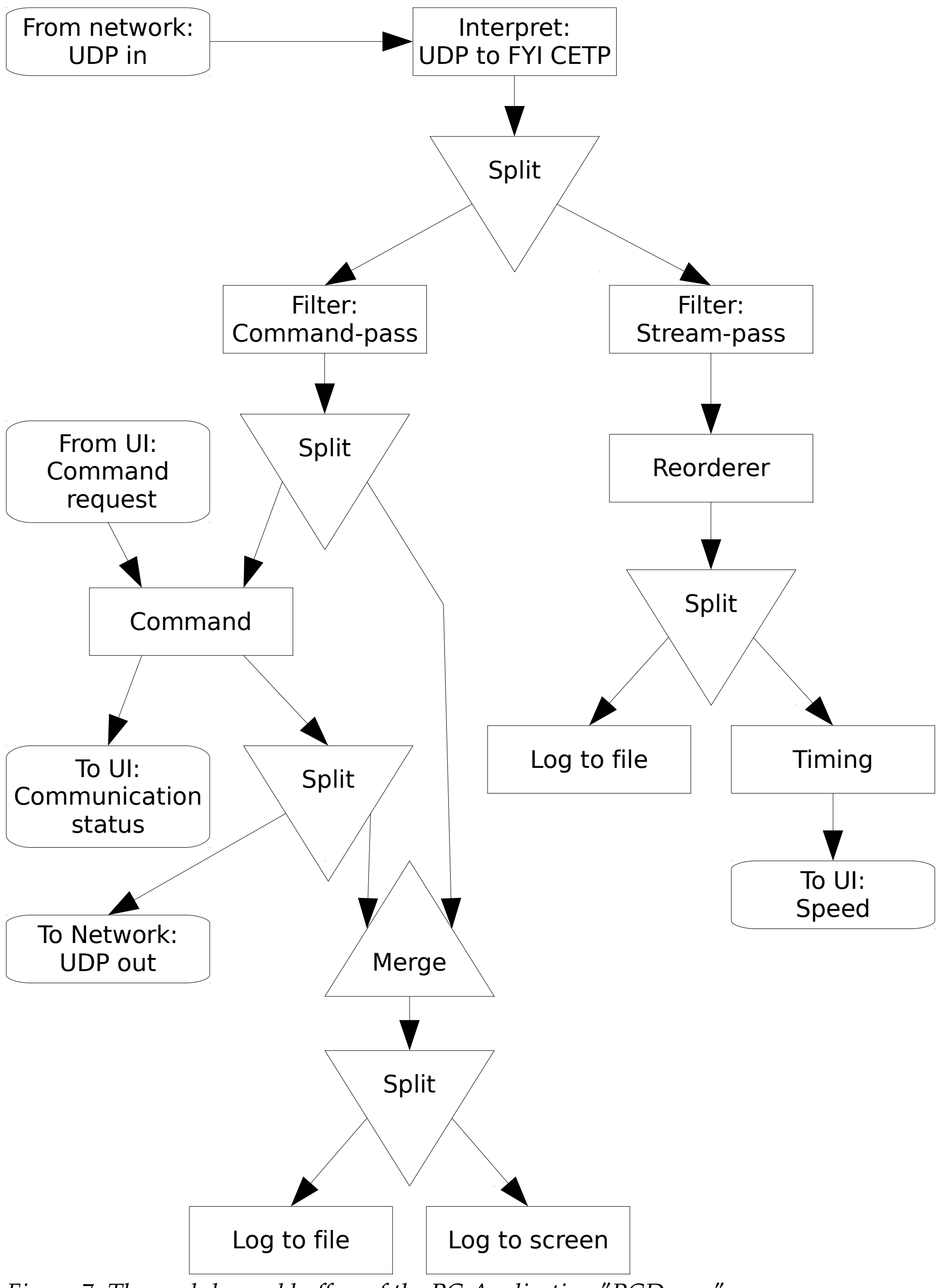


*Figure 7: The modules and buffers of the PC Application "PCDump"*

# 5 Design choices

There are not that many overall design choices in the project. Most were already outlined in the specification – such as using a microcontroller for the Ethernet encapsulation.

The other possible design choice envisioned is a more device-centered design where all the functionality in the FPGA, the microcontroller and the PC is developed separately and then interfaced, as opposed to the used communication-centric design. The following sub-chapters will further describe the choices that were made in each of the parts of the system.

## 5.1 PCB and FPGA

The already existing product MCBuster has an FPGA from Lattice and it is desirable from the thesis initiators to also use the same family of devices in the future. The XP2 series are non-volatile and is the sequel to the XP series that are used in the MCBuster. The most natural is to use one of the development kits for the XP2, but this was not done. Instead a custom FPGA PCB was developed. There were four major reasons for this: Signal integrity, available connections. development kit availability and learning desire.

### *5.1.1 Signal integrity*

For a safe, error-free communication between the FPGA and the microcontroller, shorter wires are always better. This will reduce both crosstalk and signal reflection. Using any of the FPGA development kits that were available would have meant using a ribbon cable of at least a couple of decimeters. Building a custom PCB would reduce the total signal path length to below one decimeter.(Johnson, M, 1993, pages 133)

### *5.1.2 Available connections*

The EPI interface on the microcontroller uses 32 signals in different configurations. To maximize performance and flexibility, as many as possible of these need to be connected to the FPGA. None of the Lattice XP2 FPGA development kits found had enough available outputs. That would have necessitated an FPGA provider change and made the final project product require customization before use by the project initiator. (Lattice Datasheet, 2009, page 2-3)

### *5.1.3 Development kit availability*

At the time of the project, a Xilinx Spartan3 development kit was freely available, as well as an

Altera Cyclone II. Not using one of these kits would have meant another economic investment from the project initiator.

### *5.1.4 Learning desire*

The education leading up to the Master's thesis is quite theoretical. Gaining hands-on experience with designing and using actual hardware is a clear benefit, not only due to the actual development but also in the insight as a user of the hardware.

These factors combined resulted in the development of the custom FPGA PCB. The cost of PCB manufacturing as well as the components was comparable to the cost of buying a new development kit.

The circuit schematic and the layout were designed using Eagle from Cadsoft. This was chosen because it was freely available for the project.

### *5.1.5 FPGA tools*

The tool used for designing the FPGA program was ispLEVER Starter from Lattice Semiconductor. This version is available for free and worked fine for the project. For synthesis, SynplifyPro was available as a free plugin to ispLEVER. ModelSim has been used as a simulation tool. This is one of the most widespread tool used for verification, and both the project members had used it before.

## 5.2 Interface: FPGA to microcontroller

The microcontroller provides 32 pins for the EPI interface. When deciding how to configure them the goal was to achieve high speed combined with an interface which is easy to use. The decision is mainly how many pins that should be dedicated to data or address. The specified transmission speed of the project is so high that the numbers of operations in the microcontroller should be minimized. therefore it is preferable to read as much data as possible during a bus access. The FPGA in the end product will always send whole bytes. In almost all cases will it also be an even number of bytes and it is easy to pad with one byte if necessary. This motivated the choice of a data width of 16 bytes.

It is fairly easy to change this to 8 bytes if the end product frequently needs to read an odd number of bytes. However, doing this will effectively halve the EPI bus bandwidth, probably resulting in the system  not reaching the required 53 Mbit/s speed.

## 5.3 Microcontroller and development kit

The most important features for the microcontroller is that it should be able to handle speeds of 53 Mbit/s downstream and 500 kbit/s upstream. It should also have some peripheral interfaces to Ethernet and the FPGA. Furthermore, the thesis initiators had several demands on the microcontroller features for later development of the product. The following specification was set up to fulfill this:

High performance in terms of speed.

DMA and external memory interface.
As much embedded SRAM as possible.
As much embedded Flash memory as possible.
EEPROM
EMAC unit
CAN unit
UART interface
SPI unit
USB 2.0 unit
Non-BGA (or equivalent) package
Operate in industry temperature grade (-40 to +85 degrees)
The microcontroller should also have a processor core that supports an open-source operating system. Source code, code examples and forums should be available. A widely used microcontroller is preferred.
The microcontroller should have some kind of sleep mode to save power.
The microcontroller should be in production or soon in production.
The amount of embedded SRAM and Flash memory is set to as much as possible since it is very difficult to know how much is necessary before the implementation starts.

### *5.3.1 Development kit*

The specification for the evaluation board is mostly concerned with the connectors and software environments that are available. The evaluation board should have connectors for at least the following:

External memory interface
EMAC
CAN

Other peripherals that are desirable but not necessary:

UART
SPI
USB
CPLD/FPGA
Memory card, such as Compact Flash or MicroSD
Common programming interface
Common debug interface

The cost of both microcontroller and development kit was discussed with FYI. A free or not too expensive IDE should be available for the development kit. A good support structure, like an active forum, must exist.

Approximately 30 manufacturers of microcontrollers were examined. This was narrowed down to three final candidates that fulfilled the specification, see Table 1 below for a matrix of their features. Their differences were fairly small in terms of speed and size of memory (Luminary Datasheet., 2009, page 44-46 ) (ST Datasheet, 2009, pages 11) (Freescale Datasheet, 2009, page 3).

*Table 1: A comparison between the final three microcontroller candidates*

| **Vendor** | Freescale | Luminary | ST Microelectronics |
|---|---|---|---|
| **Microprocessor** | MCF52259CAG80 | LM3S9B96 | STR912FAW44 |
| **Core** | Coldfire V2 | ARM Cortex-M3 | ARM-966E-S |
| **Clock frequency** | 80 MHz | 80 MHz | 96 MHz |
| **MIPS** | 76 MIPS | 100 DMIPS | 96 MIPS |
| **SRAM** | 64 kB | 96 kB | 96 kB |
| **Flash** | 512 kB | 256 kB | 512 kB |
| **EEPROM** | No | No | No |
| **Eth 100** | Yes | Yes | Yes |
| **CAN** | Yes | Yes | Yes |
| **UART** | 3 | 3 | 3 |
| **Ext. mem int.** | Yes | Yes | Yes |
| **DMA** | Yes | Yes | Yes |
| **SPI** | Yes | Yes | Yes |
| **USB** | Yes | Yes | Yes |
| **Package** | LQFP 144 | LQFP 100 | LQFP 128 |
| **Phy. size** | 20x20 mm | 16x16 mm | 14x14 mm |
| **Temp. range** | Industrial | Industrial | Industrial |

After looking at the development boards, the Stellaris LM3S9B96 microcontroller from Luminary/Texas Instruments was chosen. The disadvantages of the microcontroller is that it does not have an on-chip EEPROM, and the development kit does not have an integrated CPLD/FPGA which means that an external one must be used. These drawbacks were not considered as problems.

### 5.3.2 Buffer Size

The buffer in the microcontroller consists of three slots, each the size of the data of an Ethernet frame (1468 bytes). Since the FPGA runs at a higher frequency than the microcontroller it is not necessary to have a large buffer in the microcontroller. If the microcontroller runs out of data the FPGA will quickly be able to fill it up if there is any more data to be sent. Later projects will eventually design a large FIFO buffer for the FPGA to smooth out the inherent peaks and valleys of the MOST data flow.

However, a small buffer is needed to be able to read and write simultaneously. The buffer was chosen to contain three slots of data frames. One slot is used for reading from the EPI interface, one slot is used for writing to the EMAC. The third slot is used as an extra to prevent glitches; when a read or write is finished it is possible to start a new one without delay.

### 5.3.3 Microcontroller tools

The tools that were provided with the microcontroller development kit were from Keil, IAR, Code Sourcery and Code Red. Red Suite 2 from Code Red was selected since it was a full version locked to the board. The other tools had either a code size limit or a time limit. Red Suite is based on Eclipse which had been used by both project members before. For the preceding reasons Red Suite was used.

## 5.4 Interface: PC to microcontroller

The two major protocols used to transmit data across Ethernet are TCP and UDP, both operating on top of IP. After an initial calculation regarding the available processing power and available memory in the microcontroller, it was clear that TCP was out of the question (Kozierok, 2005, page 692).

Since TCP requires that packets be stored for retransmission, the available memory will be exhausted very fast: The data stream at 50 Mbit/s equals 6.25 MB/s. The microcontroller we selected has 96 KB of RAM, which would last for 15 ms. Considering the possible latencies in a network, possible TCP retransmission timeout values, the fact that all of the RAM can not be dedicated to TCP buffers and the processing power needed to handle TCP packets, we came to the conclusion that TCP was unsuitable.

Using UDP forgoes all communication reliability. This in itself is not an issue – dropping packets in a stream is acceptable. However, this condition must be detected, which UDP cannot do. Due to this limitation, UDP is also unusable.

A quick survey of some of the other available transport protocols showed that none had the functionality and low overhead that we needed. Therefore, we developed our own on top of UDP, called FYI CETP. This protocol includes a counter that is used to identify which packet it is. For more details, see Appendix B.

## 5.5 PC Application

The choice to develop the PC Application PCDump in Java was based on Java's integrated networking capabilities, its ease of creating a GUI and because the project members had previous experience with it. (Skansholm, J, 2005, page 641)

The development environment of choice was Eclipse due to its open nature, the multitude of plug-ins (for example a SVN client) and above all, previous experience.

Making the application heavily threaded with FIFO buffers was a given because of the performance requirements and fluctuating data rate. The first version of the application had a more rigid backbone which dispatched data to worker threads. This worked fine with a data rate of 50 Mbit/s, but as the performance of the microcontroller improved, the consumption rate of the main dispatcher thread became a bottleneck causing dropped data. This warranted the move to the current, completely module-based approach.

The application was developed on an computer running the Ubuntu operating system to provide easy access to the packet analyzer Wireshark, and it was apparent that Linux was more accommodating with the connect-disconnect behavior exhibited by the microcontroller during reprogramming. Where Windows would need seconds just to identify that the connection was down, in Ubuntu all it required was the push of a button. This eased development and debugging of the microcontroller.

One caveat of using Linux is that the default maximum UDP receive buffer size is too small and will cause datagrams to be dropped. This needs to be adjusted by writing a larger value – 4000000 was found to be a good size – to /proc/sys/net/core/rmem_max and requesting a larger buffer size in the Java application. 2 MB seems to be a good trade-off between memory usage and retaining zero data drops under normal conditions.

The application has never been tested on a Windows machine. Java is designed for portability which should mean that this should not be a problem, aside from the UDP receive problem mentioned above.

The research done seems to suggest that Windows does not have a limit to the receive buffer size and will honor any reasonable buffer size request.

# 6 Testing and verification

The testing was divided into several parts that were first tested separately, then the interfaces between each part were tested, and finally the entire system. Furthermore, the PCB was tested separately. ModelSim has been used during the project to simulate the FPGA code's functionality. The debugger in Red Suite 2 was used to test the C program. Eclipse's debugger was used to test PCDump. The packet analyzer Wireshark was used to debug the data sent over the interface between the microcontroller and the PC. It was also used to calculate the speed of a stream. A logic analyzer was used to analyze the data sent over the interface between the microcontroller and the FPGA. The PC that has been used when performing tests with Wireshark and PCDump is a FujitsuSiemens Amilo Si1520 with the operating system Ubuntu 9.10. Another PC with better or worse performance could produce other results.

## 6.1 PCB Testing

All voltage levels (1.2 V, 2.6 V and 3.3 V) at the PCB were tested after production to see that they were at correct levels. The pins on the card were manually checked to assure that no soldering bridges existed. Existing ones were removed using a soldering iron and scalpel.

## 6.2 System testing

A test to assure that all packets that are read from the FPGA reach the PC was done during long running streams. Several test cases between 15 minutes and 60 minutes were used. PCDump counts the number of received packets and sorts them to assure that none appears twice or is missing. The number of packets is compared to the value in the microcontroller debugger.

To simulate the FPGA running out of data, the FPGA code was modified. The code sets the EPI enable input (epi_irdy signal) low after different times and with varying durations. The EPI enable signal is connected to a dedicated pin in the EPI interface.

Verifying the watchdog functionality was done by introducing bugs in the code to trigger a watchdog reset.

To test that the functionality of the FYI CETP was working correctly a test protocol was set up to verify all functionality, available in its own document. All commands with all priorities were tested both during stream and not. Wireshark was used to assure that the header was as expected. In total, all of the approximately 75 test cases were successfully completed.

*Table 2: Test results for the test of PCDump's internal speed calculation, compared with the stated speed in Wireshark which was used as a reference.*

| | Speed | | Adjusted | Diff related to Wireshark | |
|---|---|---|---|---|---|
| **Delay value in µC** | **Wireshark** | **PCDump** | **Wireshark** | **Diff (abs)** | **Diff(%)** |
| 25000 | 10.04 Mbit/s | 9.77 Mbit/s | 9.73 Mbit/s | 0.04 Mbit/s | 0.39% |
| 30000 | 8.56 Mbit/s | 8.35 Mbit/s | 8.30 Mbit/s | 0.05 Mbit/s | 0.62% |
| 50000 | 5.39 Mbit/s | 5.23 Mbit/s | 5.22 Mbit/s | 0.01 Mbit/s | 0.13% |
| 80000 | 2.92 Mbit/s | 2.83 Mbit/s | 2.83 Mbit/s | 0.00 Mbit/s | -0.05% |
| 100000 | 2.35 Mbit/s | 2.28 Mbit/s | 2.28 Mbit/s | 0.00 Mbit/s | -0.07% |

During the project, PCDump was used to calculate the speed of an active stream. The correct speed was calculated using Wireshark by measuring the number of packets during a given time. This was done at different speeds by introducing a delay every three packets in the microcontroller. For the results, see Table 2. The tests ran for approximately 50 seconds. In PCDump the speed is calculated from 1468 bytes per frame. Wireshark uses the whole frame and therefore the speed was adjusted. Comparing these speeds gives PCDump an accuracy better than 0.65 %. This accuracy is extrapolated to all possible speeds.

An approximate reasonability test was performed to see that the values are not totally wrong. The number of packets received during 10 seconds was measured. The amount of data should be approximately the product of speed, time and the payload of a frame,

Received Data = Speed * time * Payload .

Testing this has shown that the speed results are reasonable.

The maximum speed of the system was measured using PCDump after everything had been implemented. The specification requires a download stream of 500 kbit/s for control messages. The result of the upstream speed is much higher than the requirements of 53 Mbit/s. This means that it is no problem to reach the download speed. Because of this and after discussions with the thesis initiators it was decided that a test that sends control messages at this speed was not needed.

# 7 Results

The maximum measured speed of the system was 73.5 Mbit/s. Optimizing the code for higher speeds by removing functionality made the DMA transfer the bottleneck of the system. Then, speeds of about 78 Mbit/s were reached. This is the maximum speed that could be reached using this solution.

PCDump is not able to log all the data during a stream at maximum data rate on the used PC. By introducing a delay in the microcontroller to simulate a speed of 50 Mbit/s it was possible to log the data.

The tests of the FYI CETP was done without finding any bugs. Overall, for the entire system, no bugs are known.

# 8 Discussion

This project is merely a part of a larger project involving the MCBuster, with the obvious goal of developing a new version of the MCBuster with additional functionality. We have demonstrated that it is possible to transmit the required amount of data across Ethernet and have provided an implementation of this that could be used with minimal adaptations.

It is possible that this solution, with its speed of 73 Mbit/s, could be used with the next standard, MOST150. This should work if the user does not stream too many MOST channels at the same time. If the performance is not satisfactory, a more powerful microcontroller is an option. Another option is to encapsulate the data to Ethernet packets within the FPGA either in hardware or by using a soft-core. A third option would be to have a pure FPGA solution, removing the microcontroller.

# 9 References

DOD Standard Internet Protocol (1980). *RFC 760* (Standard). URL:reference <http://tools.ietf.org/html/rfc760>

Freescale Datasheet (2009). *Freescale Semiconductor Document number: MCF52259*. URL: reference <http://freescale.com>

Luminary Datasheet. (2009). *Texas Instruments Stellaris LM3S9B96 Microcontroller Data Sheet*. URL:reference <http://www.luminarymicro.com/products/lm3s9b96.html>

IEEE 802.3. (last updated 2010). *Ethernet working group.* URL:reference <http://www.ieee802.org/3/>

Johnson and Graham, M. (1993). High Speed Digital Design: A Handbook of Black Magic. United States of America: Prentice Hall Ptr.

Kamal, R. (2009). *Microcontrollers: Architecture, Programming, Interfacing and System Design.* Third Edition. India: Dorling Kindersley.

Kozierok, M. C.(2005). *The TCP/IP guide: a comphrehensive, illustrated internet protocols reference.* United States of America: No Starch Press.

Lattice Datasheet (2009). *LatticeXP2 Advanced Evaluation Board*. URL: reference <http://www.latticesemi.com/>

Leon-Garcia, A. Widjaja, I (2004). *Communication Networks Fundamental Concepts and Key Architectures*. NY New York: McGraw-Hill.

MOST. (2008). *Most Specification Rev. 3.0.* URL:reference <http://www.most.de/home/index.html/>

Null and Lobur, J. (2006). *The essentials of computer organization and architecture.* Second Edition. Unites States of America: Jones and Barlett Publishers.

Plummer, C. D. (1982). *An Ethernet Address Resolution Protocol. RFC 826* (Standard). URL:reference <http://tools.ietf.org/html/rfc826>

POSTEL, J. (1980) *User Datagram Protocol. RFC 768* (Standard). URL:reference <http://tools.ietf.org/html/rfc768>

Skansholm, J. (2005). Java direkt: med Swing. Stad: Tryckeri.

ST Datasheet (2009). *STR91xFAxxx*. URL: reference <http://www.st.com/mcu/>

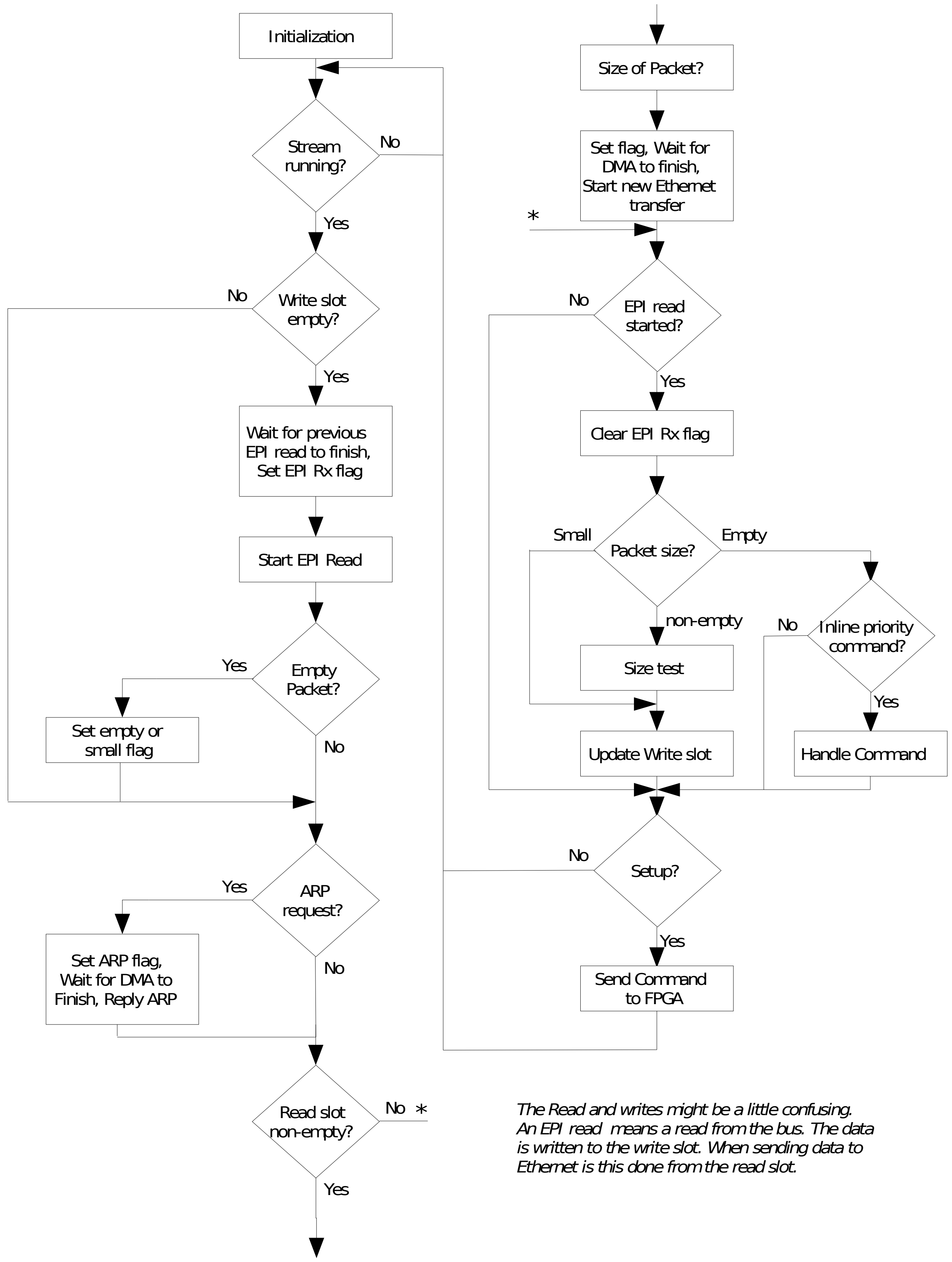
# 10 Appendices

## Table of Appendices

# Appendix A: Figures

This appendix contains figures that were deemed to large to contain in the main report.

## A.i: The C program



*The Read and writes might be a little confusing. An EPI read means a read from the bus. The data is written to the write slot. When sending data to Ethernet is this done from the read slot.*

# Appendix B: FYI CETP specification

This appendix contains the proprietary FYI CETP specification.

# Contents

## B.i: Background

We are designing an embedded system that shall be able to transmit and receive data fast, easily and with a specified and varying degree of handshaking and transmission reliability. This document describes a proposed protocol to be used on top of UDP to provide this variability.

Our system has two different kinds of transmissions: One part that is not time sensitive which is used to send control messages (hereafter called 'command'), and one part that needs as high throughput as possible (hereafter called 'stream').

For our data transmissions, we are using UDP over IP over Ethernet, to give maximum throughput with minimum overhead but still be standards-compliant. UDP, however, lacks any form of transmission control, which we need for the command parts of our communication. The primary alternative to UDP, TCP, has a more elaborate handshaking than we need with a requirement for retransmissions of dropped packets. This will severely limit the maximum possible speed of our system during the 'stream' phase, therefore TCP is too cumbersome. We don't need that much control during the 'stream' phase either, just a way to detect if there have been dropped packets. The 'command' phase could use TCP. However, it is too elaborate for us even then – we can accept a much simpler scheme. Thus the need for this protocol.

## B.ii: Definitions

This section describes definitions used in this document.

A *flag* is a one-bit value that can be used to indicate on or off.

A *field of flags* is several flags grouped together. In a field of flags, each value can be turned on or off. This is in contrast to the *field of options* defined below.

An *option* has no fundamental difference from a flag – they are both used to indicate on or off. In fact, a single option is identical to a flag.

A *field of options* is an encoding of mutually-exclusive flags used to save space and provide built-in sanity checking. Using a field of N bits, 2^N different options can be encoded.

## B.iii: Proposal

We propose to use a 16+16 bit header as described in this document, to be placed immediately after the UDP header. This has the clear benefit of maintaining the whole-word alignment present in all parent headers, and also maintains this alignment throughout the payload.

The header is divided into two 16-bit parts: The first (low order) 16 bits contain information regarding packet type and some general flags. The next 16 bits (the high order bits) are packet type-dependent, but required by this proposal.

### B.iii.i: Packet type

Of the 16 bits with low offset, the low 8 contain the packet type encoded as follows:

| Value | Name | Description |
|---|---|---|
| 0x00 | N/A | Unused – this value is not permitted and should generate a NACK response |
| 0x01 | STREAM | The packet contains streaming data |
| 0x02 | COMMAND | The packet contains a command |
| 0x03 → 0xFE | UNUSED | These values are not defined in the current specification |
| 0xFF | EXTENDED | This is reserved for future expansion and indicates that the 32 bits of this header are to be discarded and that another header follows (to be specified as needed at a later time) |

### B.iii.ii: Packet type options

The high 8 of the low 16 bits contain options to indicate modifications of the packet type

| Value | Name | Description |
|---|---|---|
| 0x00 | NONE | No options set |

| | | |
|---|---|---|
| 0x01 | PROCESSING | This flag can be set to acknowledge that a command packet type has been received during streaming, but that it has lower priority than the streaming and will be handled at the next possible time. Note that there are different priority levels for command packets received during streaming – see the COMMAND packet type section for more information. |
| 0x02 → 0xFF | UNUSED | These values are not defined in the current specification |

## B.iii.iii: Packet type: STREAM

When using packet type STREAM, the high 16 bits of the header are used as a counter to track dropped packets. The stepping of this counter (how often it increases) can be setup to applicable powers of two, by using the COMMAND packet type, SETUP subtype. This counter is modulo-2^16 with wrapping, and is send using host byte order – not network byte order. This is done to improve performance on processors with relatively limited performance, such as microcontrollers.

The reasoning behind not incrementing the counter each time is that we won't have to update the headers as often during a running process, thus improving performance. We are still guaranteed to know if there has been a dropped packet, since the target system will know the value of the stepping, and can count packets. What we lose is the ability to detect exactly which packet has been dropped. This is a tradeoff, naturally, and therefore it has been made available to for run-time setting and optimization.

The STREAM packet type has no handshaking – there shall be no retransmission mechanism.

Any packets received by a device while it is using STREAM to send packets will be handled based on their priority, see the COMMAND packet type section. Some shall cause an immediate halt, some an immediate response with subsequent restart, some shall be handled inline with the stream at a convenient time and some will not give any response at all.

## B.iii.iv: Packet type: COMMAND

When using the COMMAND packet type, the low 12 of the high 16 bits are defined as options with the following definitions:

| Value | Name | Payload | Description |
|---|---|---|---|
| 0x000 | N/A | -- | Unused – this value is not permitted and should generate a NACK response |

| | | | |
|---|---|---|---|
| 0x001 | START | Padding | Start streaming data according to a previously sent SETUP request, or resume from the last STOP, whichever is applicable |
| 0x002 | STOP | Padding | Stop streaming data |
| 0x003 | SETUP | Setup structure | Setup a streaming transfer according to the structure in the payload |
| 0x004 | ACK | Padding | Acknowledge the last received packet |
| 0x005 | NACK | Padding | Do not acknowledge the last received packet – i.e. message not understood. Shall cause a retransmission of the NACK'ed packet |
| 0x006 | UACK | Padding | 'ACK unavailable'. Can not retransmit packet requested (by incoming NACK). For example, if last packet sent was a STREAM packet. |
| 0x007 | ERROR | Error source | Indicate that an error has occurred. The payload may be used to indicate the error cause. |
| 0x008 → 0xFFE | N/A | – | These values are not defined in the current specification |
| 0xFFF | PANIC | Padding | This value is used to indicate critical failures or any other reason that needs to have the communication terminated as soon as possible. The behavior after a PANIC is, by nature, undefined. |

The COMMAND packet type uses a rudimentary form of transmission control and retransmissions, based on ACK's and NACK's. There shall be only one pending COMMAND message at all times, until it has been ACK'ed or NACK'ed, nothing more may be sent – until a timeout has elapsed, which should be treated as a NACK.

A COMMAND communication is finished when the initiator has received an ACK on its message and then replied with an ACK (creating the above mentioned ACK-ACK sequence), or alternatively with an UACK-ACK sequence (thus ending with the respondent's ACK of the initiator's UACK).

The Payload field indicates what the bytes following the header are used for. If it is set to 'Padding', there is no more information in the packet and the data is just padding to fulfill the Ethernet specification. Any other value shall be specified in this document.

### B.iii.iv.i: COMMAND priorities

When using the COMMAND packet type, the high 4 of the high 16 bits are defined as flags to indicate priority (during a STREAM transfer or other future defined , as follows:

| Value | Name | Description |
|---|---|---|
| 0x0 | N/A | Unused – this value is not permitted and should generate a NACK response unless sent during a running STREAM |
| 0x1 | LOW | This command will be ignored if sent during a running STREAM |
| 0x2 | INLINE | This command will be responded to at a convenient time during a running STREAM. This shall be when there is no more data to STREAM or when the STREAM counter is incremented, whichever occurs first |
| 0x4 | RESUME | This command will cause the STREAM to halt as soon as possible. This command will then be responded to, and once the final ACK-ACK (see communication flow below) has been reached, the STREAM will restart (same as sending a COMMAD START immediately after the ACK-ACK) |
| 0x8 | STOP | This command will cause the STREAM to halt as soon as possible. This command will then be responded to, and the STREAM will be left in a halted state. This can later be resumed by sending a COMMAND START |

If more than one of these bits are set, the highest one shall count. This enables setting the high 16 bits to 0xFFFF for a COMMAND PANIC. However, the default behavior is that only one bit should be set.

Please note that this priority scheme will make some commands non-functional if the required priority is not set – a COMMAND PANIC with LOW priority will have no impact during a STREAM! On the other hand it gives flexibility – you can request that the current STREAM be stopped immediately, or at a convenient time, for sending other commands, for example.

### B.iii.iv.ii: COMMAND SETUP structure

When using the COMMAND SETUP command, the payload data shall contain these options according to the following format: The first 32 bits is the option name, and the next 32 bits is the option value. Multiple options may be specified in a packet, each after the other.

In this specification, the following options are specified. All others are for future use.

| Value | Name | Description |
|---|---|---|

| | | |
|---|---|---|
| 0x00000000 | PAD | This value may be used to pad the packet to the length required by the Ethernet specification |
| 0x00000001 | SET | Send SET to the FPGA |
| 0x00000002 | INC | Send INC to the FPGA |
| 0x00000003 → 0xFFFFFFFF | NA | These values are not defined in the current specification |

## B.iii.v: Transmission flow examples

| Simple successful COMMAND | | COMMAND with loss (message not understood) | | COMMAND with unrecoverable loss (UACK) | |
|---|---|---|---|---|---|
| Source | Target | Source | Target | Source | Target |
| COMMAND START | | COMMAND START | (Message has been jumbled in transmission) | COMMAND START | (Message has been jumbled in transmission) |
| | ACK | | NACK | | NACK |
| ACK | | COMMAND START | | UACK | |
| Done (command will be executed) | | | ACK | | ACK |
| | | ACK | | Done (command will **NOT** be executed) | |
| | | Done (command will be executed) | | | |

| COMMAND with ACK loss | | COMMAND with S → T loss | | COMMAND with much loss | |
|---|---|---|---|---|---|
| Source | Target | Source | Target | Source | Target |
| COMMAND START | | COMMAND START | | COMMAND START | (Message has been jumbled in transmission) |
| (ACK jumbled in transmission, not understood) | ACK | (timeout delay – no response to message) | | (NACK jumbled in transmission, not understood) | NACK |
| NACK | | COMMAND START | | NACK | (NACK received - Resend request of target's NACK) |
| | ACK | | ACK | | NACK |
| ACK | | ACK | | COMMAND START | |
| Done (command will be executed) | | Done (command will be executed) | | | ACK |
| | | | | ACK | |
| | | | | Done (command will be executed) | |

| STREAM | | STREAM with INLINE | | STREAM with INLINE, loss | |
|---|---|---|---|---|---|
| Source | Target | Source | Target | Source | Target |
| COMMAND START | | COMMAND START | | Assuming already started transfer | |
| | ACK | | ACK | | STREAM |
| ACK | | ACK | | | STREAM |
| | STREAM | | STREAM | COMMAND STOP (INLINE) | (Cmd jumbled in transmission, not understood) |
| | STREAM | | STREAM | | STREAM |
| | STREAM | COMMAND STOP (INLINE) | | | Zero or more STREAM |
| | etc. | | STREAM | | STREAM with PROCESSING |
| | | | Zero or more STREAM | | STREAM with PROCESSING until next command slot |
| | | | STREAM with PROCESSING | | NACK |
| | | | STREAM with PROCESSING until next command slot | | STREAM |
| | | | ACK | COMMAND STOP (INLINE) | |

| | | | | | |
|---|---|---|---|---|---|
| | | | STREAM | | |
| | | ACK | | | |

## B.iv: Changes

This section describes the changes made in this document between different versions

### B.iv.i: Version 0.5 (2009-11-26)

Removed the 'undersized' flag and associated functions due to redundancy – this same behaviour is achieved using the 'length' field in UDP.

Transmission flow examples have been updated.

Changed date format in header.

### B.iv.ii: Version 0.6 (2010-02-26)

Added the ERROR packet type

Fixed 3.2, packet options, value 0x00 is allowed and means no options