ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Marco Kuster

Firmware for Reconfigurable Hardware OS Platform

Student Thesis SA-2004-09
October 2003 to December 2003

Supervisor: Herbert Walder
Co-Supervisor: Ch. Plessl, M. Dyer
Professor: Prof. Dr. L.Thiele

Contents

Chapter 1

Introduction

1.1 Reconfigurable Hardware Operatin System

Reconfigurable systems have a great potential to speed up applications because of performing calculations in hardware. Thus an increase of performance linked with the flexibility of a software system make reconfigurable systems very interesting. One important prediction, of course, is the amazingly fast development of the speed and size of FPGA's.
The main goal of reconfigurable systems is to use small FPGA's and load code and programs on demand from an external memory and the dynamic administration of resources. Furthermore an increase of reusability and extensibility of hardware systems is aimed.

1.2 XFBoard Platform

These reasons have lead to a project called X-Forces at the Computer Engineering Lab of the Federal Institute of Technology and ended in a prototype board as a base for a **R**econfigurable **H**ardware **O**perating **S**ystem (RHWOS).
What are the goals of this project?
The project is supposed to show the possibilities and present limits of such RHWOS and to collect experiences in how much overhead they have. The reconfiguration of the used FPGA's shall get as fast as possible because of the wish creating a highly flexible and efficient system. Of course acquire knowledge about hard- and software debugging of these complex systems is another very important thing to shorten the development cycle.
Picture 1.1 on page 2 shows an abstract overview of such a RHWOS platform.

The idea is that a CPU (C-FPGA) is administrating the resources and the tasks running on a second FPGA, called R-FPGA. The components (C_n in the picture) are I/O devices, memory and other subsystems. Actually the tasks don't run only on the R-FPGA but on the CPU too. All the administration and flow steering is running on the CPU while the calculations (i.e. video decoding algorithms) are executed on the R-FPGA.

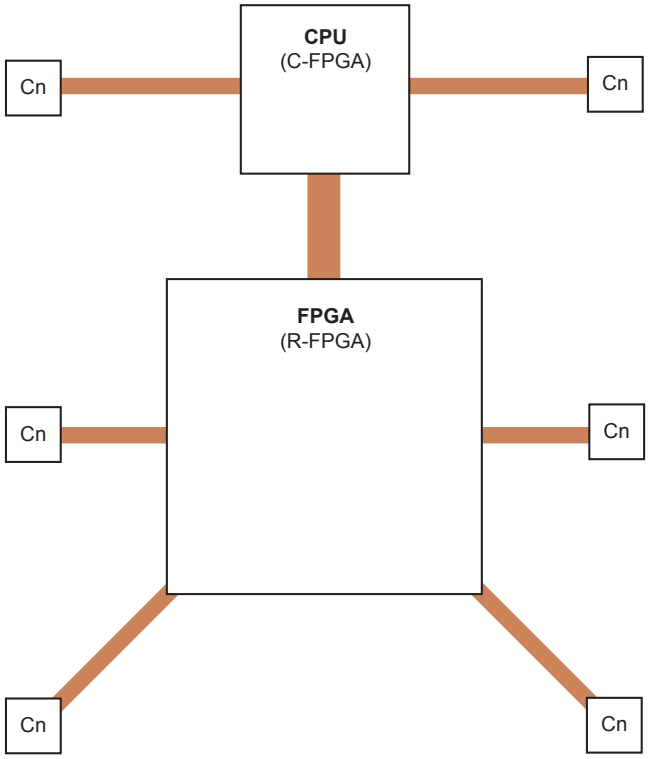

Figure 1.1: RWHOS platform

1.3 Conceptual formulation

One of the first steps when possessing such a prototype board is to implement something like a bootloader for making your life easier. When evaluating the most important parts of a minimal running OS, the network capabilities have a very high priority.
Main goals of this student thesis were:
1. Evaluate already existing cores (hardware) and libraries (software) from Xilinx
2. Create a concept for a PC administration software called *XFBoard PC Suite*
3. Implement network functionality on the FPGA and test programs on PC side written in C respectively Visual C++ with MFC
4. Write an easy-to-use program for the PC to configure the R-FPGA via ethernet

Chapter 2

Protocols

2.1 Introduction

Programming a network transceiver does not mean only being experienced in a programming language like C, Assembler or VHDL, but also having detailed knowledge of the different protocols respectively the build up of the different layers.
For more detailed information about the different protocols go to page www.faqs.org/faqs/ where all RFC's are available.

2.2 Hardware environment

The firmware based on the hardware environment of the XFBoard, see [1] for further information. Picture 2.1 on page 3 gives an overview of the network environment of the XFBoard. My work was implementing the network stuff, thus there is only the network physical scheme showed.

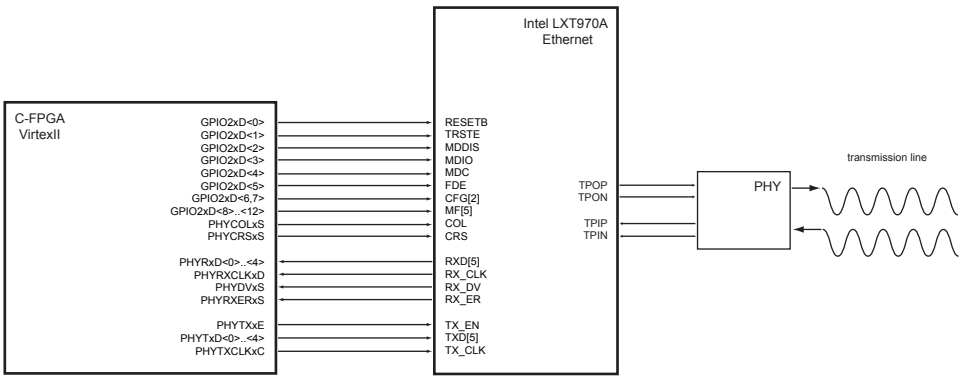

Figure 2.1: Physical Scheme

2.2.1 General information

Pay attention to the fact that the transmit and receive buffer need to be filled/read in *big-endian* order!
All byte offsets in the pictures in this chapter are relative to the beginning of the TX respectively RX buffer!

2.2.2 Receive Diagram Example

The *Intel LXT970A Ethernet* receives the bytes in *little-endian* order. Notice that first the lower nibble and after the higher nibble arrives. (picture 2.2 on page 4)

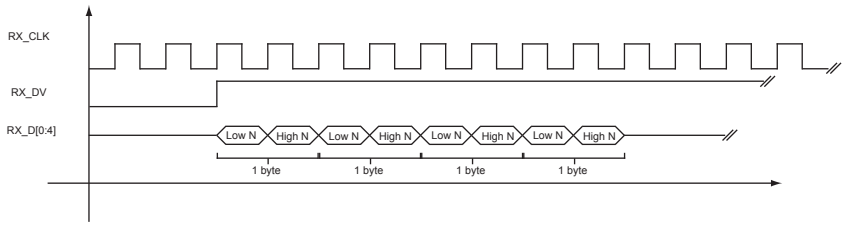

Figure 2.2: Receive Diagram Example

2.2.3 Layers

A short overview of the different protocols and their build up in five layers. This model does not correspond to the *OSI layer model*, but a simpler form of it. The figure 2.3 on page 5 should be easy to understand without any accompanying text explanations.

2.3 Ethernet

The lowest level protocol you will get in contact with is the ethernet protocol (see IEEE 802.3 for details). See picture 2.4 on page 5 for the exact size of the several parts.
The destination and the source address are MAC addresses, which is short for **M**edia **A**ccess **C**ontrol address, a hardware address that uniquely identifies each node of a network. An example for a MAC address is *00:01:02:90:F1:97*. This address is worldwide unique and is built of the maufactures code. Further information about the ethernet standard you will find on page grouper.ieee.org/groups/802/3/ from IEEE.

Fields

Destination Address (6 bytes): MAC address of the destination node or NIC.
Source Address (6 bytes): MAC address of the source node or NIC.

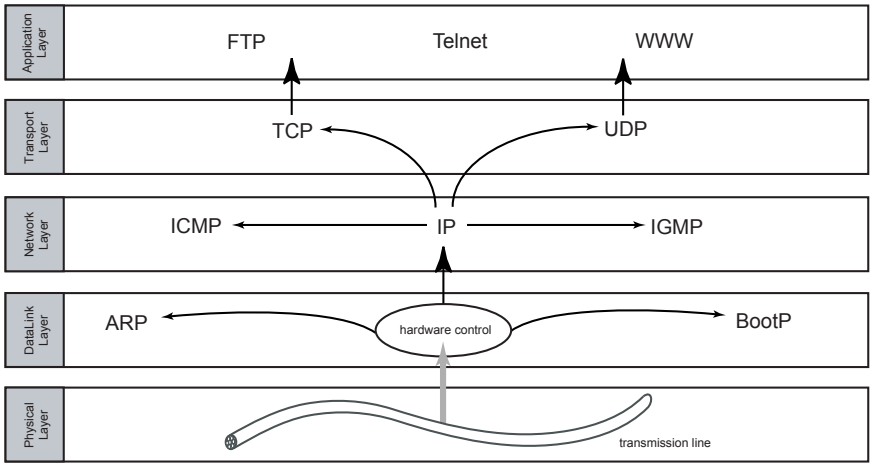

Figure 2.3: Network Layers

Type/Length Field (2 bytes): in the network codec it is only used as a type field. Value *0x0600* indicates that the field is a type field and is the standard value for an ethernet packet. Value *0x0806* indicates that the packet is of type ARP.

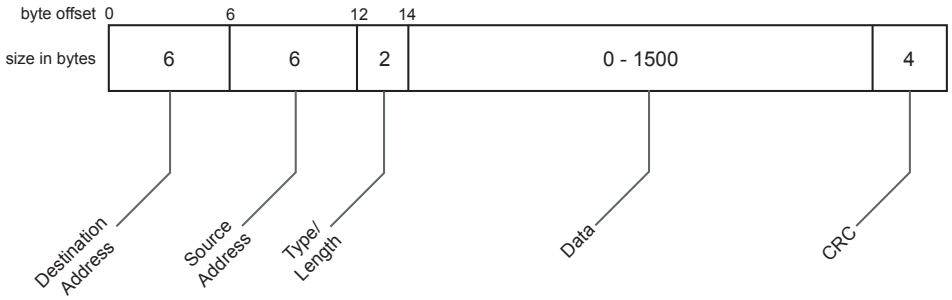

Figure 2.4: Ethernet protocol

2.4 IP

This section shows the basic build up of an IP packet header

Fields

Version (4 bit): Used version of the IP protocol, mostly 4.
HLEN (4 bit): Length of the header in DWORD's.
Service Type (8 bit): Priority of a packet and way of transmission (Type of Service).
Total Length (16 bit): Total Length of the whole IP packet.
Identification (16 bit): A value of the sender which helps to identificate the several fragments of a packet *(This field is not used in the network codec of the XF Board).*
Flags (3 bit): Actually only two of the three bit's are used. They refer to the fragmentation of the packet *(This field is not used in the network codec of the XF Board).*
Fragment Offset (13 bit): Byte offset to the beginning of the data of a packet.
Time To Live (8 bit): Live counter of a packet. Will be decreased every time a packet is passing a router. If 0, packet will terminated.
Protocol (8 bit): Describes the type of the packet. *1=ICMP*, *2=IGMP*, *3=GGP*, *6=TCP*, *8=EGP*, *17=UDP*
Header Checksum (16 bit): Checksum over the IP header to guarantee the authenticity of the packet.
Source IP Address (32 bit): IP address of the sender.
Destination IP Address (32 bit): IP address of the destination.
Options (variable): IP options, if needed at all *(This field is not used in the network codec of the XF Board).*
Padding (n*8 bit): If options are used and they don't finish at a DWORD border.

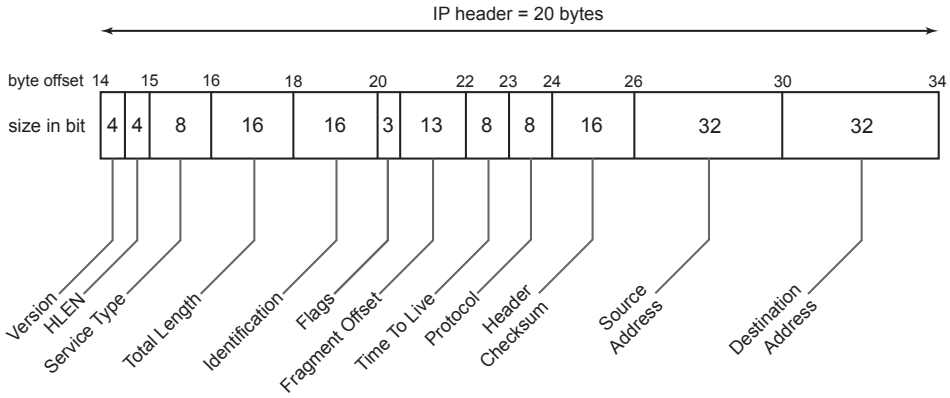

Figure 2.5: IP protocol

2.5 ICMP

ICMP is short for **I**nternet **C**ontrol **M**essage **P**rotocol and as the name implies it is only for control functions used and not for data transport.

Fields

Type (8 bit): Identifies the message. Look up the meanings of the different values directly in the [2]RFC 792, because only the *Echo Request* and the *Echo Reply* are important for us.
Values: *0=Echo Reply*, *3=Destination unreachable*, *4=Source Quench*, *5=Redirect*, *8=Echo Request*, *9=Router Advertisement*, *10=Router Solicitation*, *11=Time Exceeded*, *12=Parameter Problem*, *13=Timestamp*, *14=Timestamp reply*, *15=Information Request*, *16=Information Reply*, *17=Address Mask Request*, *18=Address Mask Reply*
Code (8 bit): Acts as a sub-code of the type field. The values are all listed, but only the value 0 will be used when doing a 'ping'.
Values: *0=Net unreachable*, *1=Host unreachable*, *2=Protocol unreachable*, *3=Port unreachable*, *4=Fragmentation needed and don't fragment was set*, *5=Source route failed*, *6=Destination network unknown*, *7=Destination host unknown*, *8=Source host isolated*, *9=Communication with destination network is administratively prohibited*, *10=Communication with destination host is administratively prohibited*, *11=Destination network unreachable for Type of Service*, *12=Destination host unreachable for Type of Service*
Checksum (16 bit): Checksum of the whole ICMP packet.

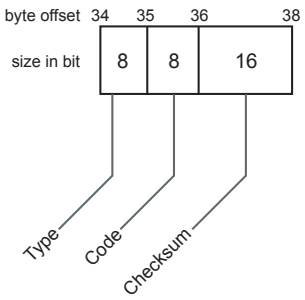

Figure 2.6: ICMP protocol

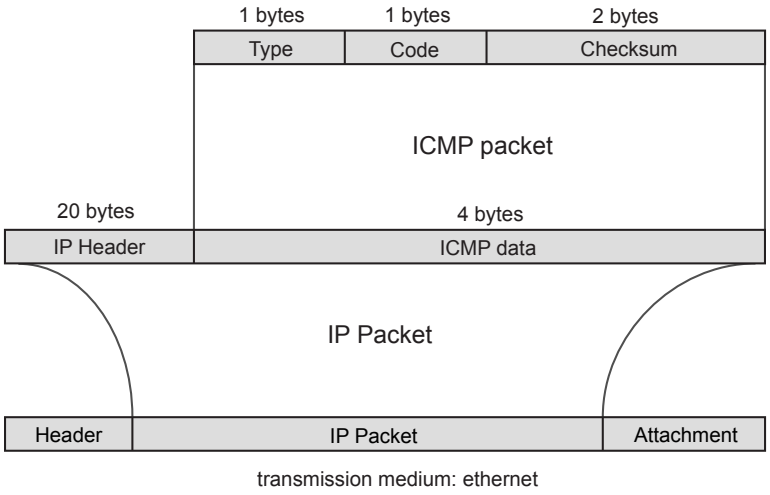

Figure 2.7: ICMP encapsulation

2.6 ARP

The description of the fields and lengths are only valid for ethernet and IP protocol as we use. For any other combinations values may change.

Fields

Hardware Address Space (16 bit): The type of hardware address, 0x0001 stands for ethernet addresses.
Protocol Address Space (16 bit): The type of protocol address, 0x0800 stands for IP addresses.
Length of Hardware Address (8 bit): Number of bytes of the physical address (=6 bytes for MAC addresses).
Length of Protocol Address (8 bit): Number of bytes of the protocol address (=4 bytes for IP addresses).
Command (16 bit): Code which specifies the ARP message: *ARP_REQUEST=1 (question)*, *ARP_REPLY=2 (answer)*
Sender Hardware Address (48 bit): Physical address of the sender.
Sender Protocol Address (32 bit): Protocol address of the sender.
Target Hardware Address (48 bit): Physical address of the target.
Target Protocol Address (32 bit): Protocol address of the target.

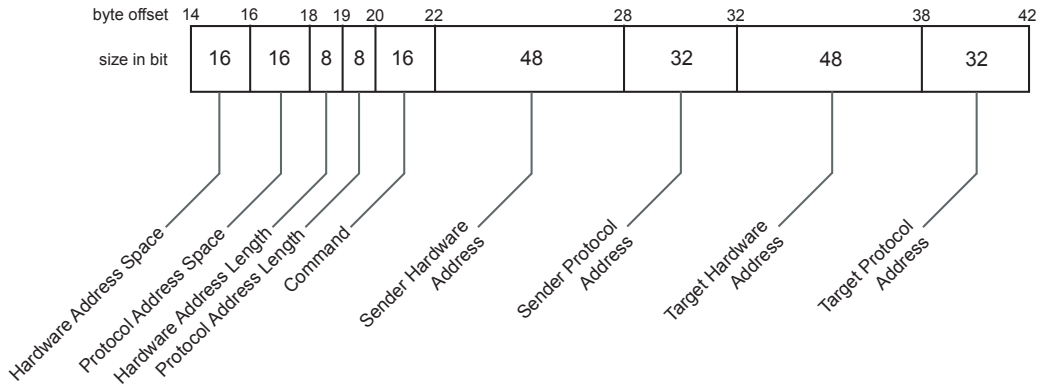

Figure 2.8: ARP protocol

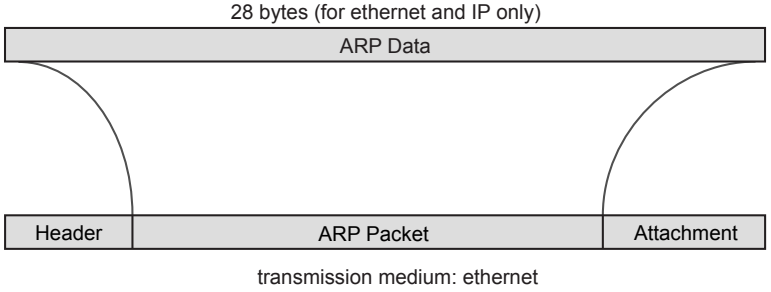

Figure 2.9: ARP encapsulation

2.7 UDP

The format of the UDP packet is quite simple and easy to understand. The UDP packets are the base for sending data from a PC to the XFBoard. TCP is more popular in the internet because it is much more secure referred to the packet loss. UDP has no packet control mechanism, but on the other side it is faster than TCP. After reflecting these facts we decided to implement UDP, because there is no need to send packets over a couple of routers, but only a back-to-back solution.

Fields

Source Port (16 bit): Port from which the packet was sent.
Destination Port (16 bit): Port to which the packet is sent.
Datagram Length (16 bit): The length of the whole packet, header & data.
Checksum (16 bit): Checksum over a pseudo header, the right header and the data.
Data (up to 65'527 bytes): User data.

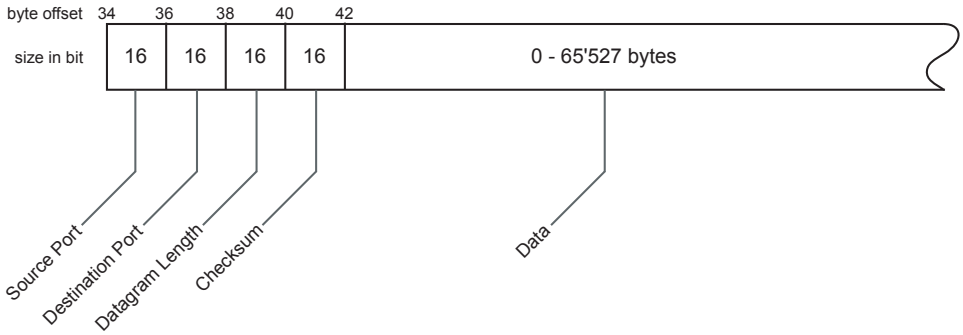

Figure 2.10: UDP protocol

The next picture 2.11 on page 10 shows the encapsulation of a UDP packet

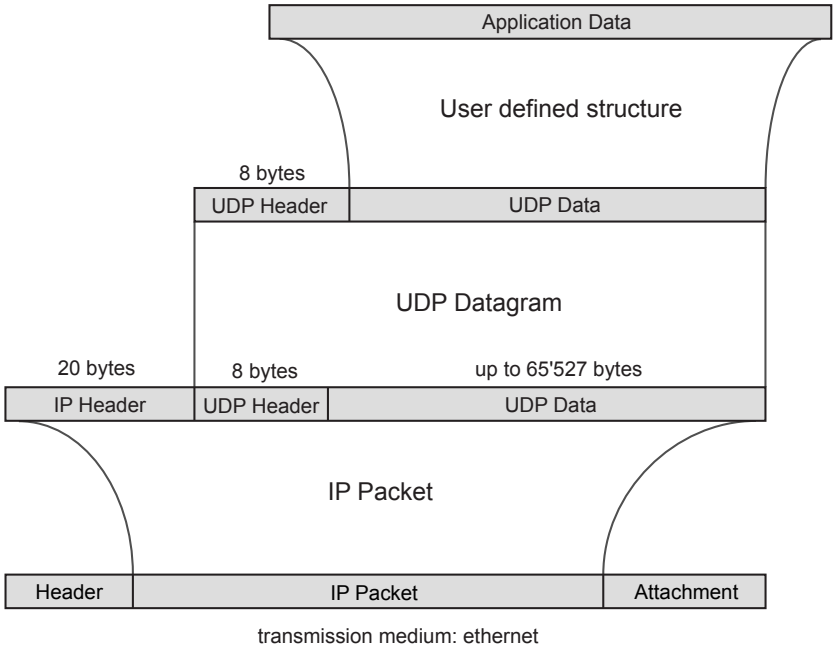

Figure 2.11: UDP encapsulation

2.7.1 Checksum calculation

Something special about the checksum calculation of a UDP packet is, that you don't have to calculate it over the whole packet but first insert a pseudo header as well at the beginning of the packet and calculate then the checksum over pseudo header, header and data. After the calculation remove the pseudo

header and send *only* the header and the data!
Picture 2.12 on page 11 shows the build up of the pseudo header.

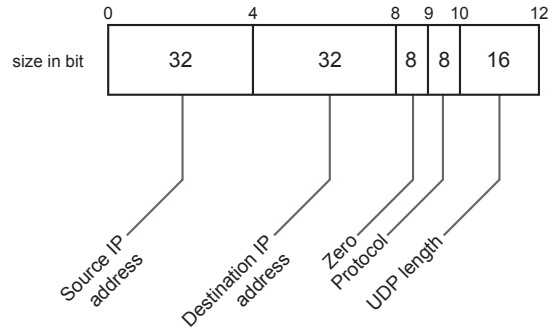

Figure 2.12: UDP pseudoheader

2.8 Examples of a RX and TX buffer

These two pictures are thought to show a developer how the transmit (TX) and receive (RX) buffer have to filled in.

RX buffer
- the software monitors the receive Status bit ('S' in the control byte) until it is set to '1' by Ethernet Lite
- once the Status is set to '1', the software reads the entire receive data out of the dual port memory
- the software writes a '0' to the receive Status bit enabling the Ethernet Lite to resume receive processing

TX buffer

- the software stores the transmit data in the dual port memory starting at address offset '0x0'
- the software writes the length data in the dual port memory at address offset '0x1FF4' and offset '0x1FF8'
- the software writes a '1' to the Status bit (bit 7 at address offset '0x1FFC')
- the software monitors the Status bit and waits until it is set to '0' by the Ethernet Lite before initiating another transmit

'Software' in the text above means the network codec described in chapter 4

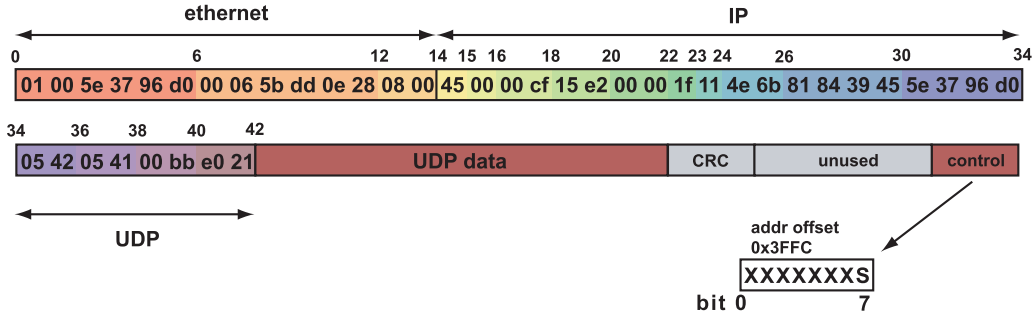

Figure 2.13: RX buffer example

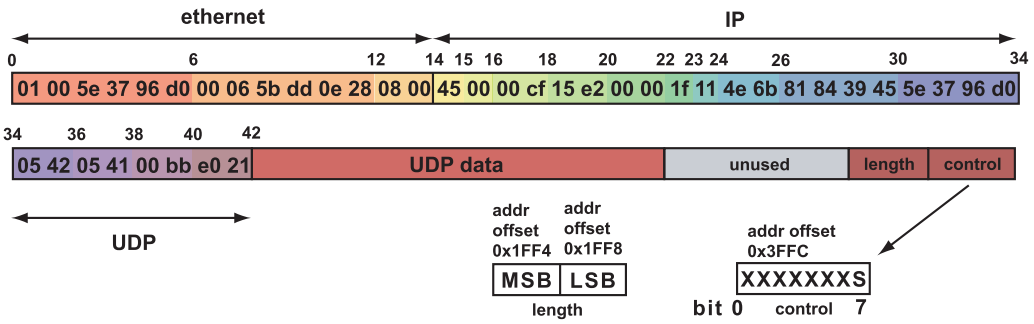

Figure 2.14: TX buffer example

Chapter 3

Concept XFBoard PC Suite

3.1 Overview

3.1.1 Purpose

Administration and installation tool for the XFBoard, executed on a PC which is connected to the XFBoard.

3.1.2 Ideas and basic principles

1. Use and handling have to be as simple as possible with the necessary functionality.
2. Modular structure - new parts can easily added to the already existing main part.
3. Programmed in Visual C++ with MFC.

3.2 Main menu or start screen

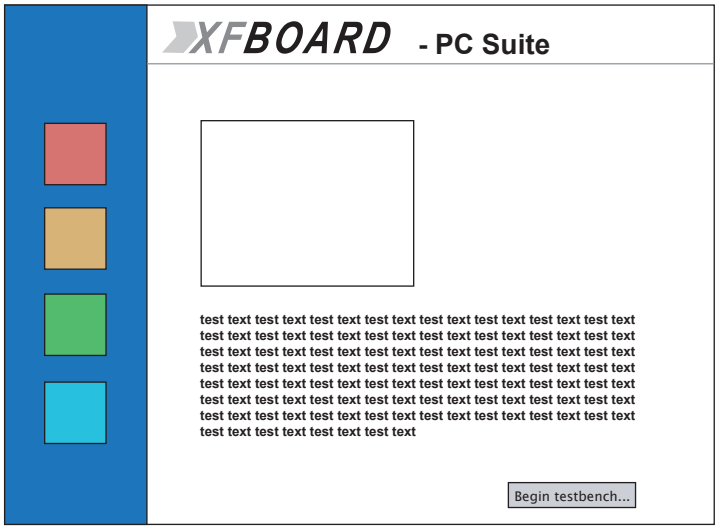

Figure 3.1: PC Suite main menu

The main menu is divided into two parts:
Left: Some kind of Icon- or Buttonview list to see all the different possibilities and modules at a glance.
Right: When an Icon/Button is pressed on the left side, all information and settings for this module will showed on the right side. A button is the link to a dialog of the module where the user can execute testbenches, download bitstreams, control FIFO buffers, memory dumps and many more.

3.3 Modules and module interface

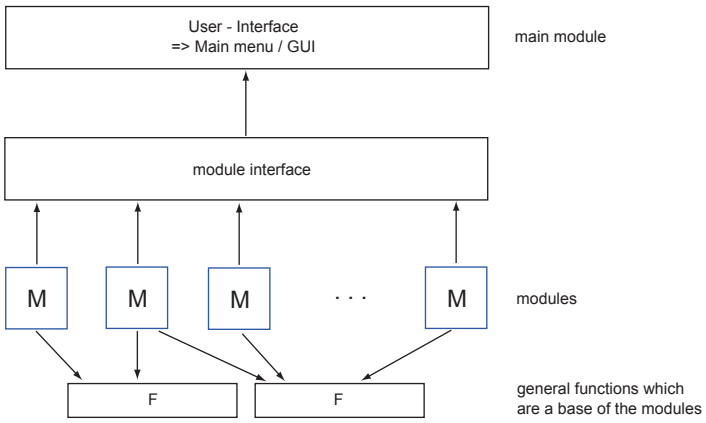

Figure 3.2: Vertical structure of the PC Suite

As showed in picture 3.2 on page 15 there is a module interface layer below the GUI layer. The purpose of this layer is to feature an interface between the modules and the main program/GUI. The bottom layer is built of functions and objects which can be used of all the different modules so that every function only once has to be written.
For the implementation there are two possibilities:
1. The several modules are realised as dll's
- modules are copied in a directory of the main program, recognised when starting next time the main program and added to the Iconview automatically
- thus the modules have to be dialog based windows and a certain specification for the graphical user interface of these dialogs must exist

2. All object/classes and the whole framework are already realised and ready to be enhanced and added
- the code of a new module only has to inserted at the right place
- proper naming conventions and specifications of the interfaces have to be given
- whenever a module is added, a completely new compilation of the whole project is necessary
- the GUI is for all the modules exactly the same

Chapter 4

Network Codec

4.1 General Information

The following functions described in this documentation provide some basic network functionality such as *Ping Reply*, *ARP Reply*, *ARP Request* and sending *UDP packets*. This short documentation shows how they work and how they can be used.

4.1.1 Included Files

These files need to be copied in your *data* directory of your Xilinx Project Studio project:

```
network.c, network.h, network_functions.c, arp.c
```

Perhaps you want to have an example file for using the network code, so copy then the

```
system.c
```

file in your directory too.

4.2 Functions

List of functions and their use

void Init()
Initialises the IP and MAC address of the XF Board and some other used variables. Furthermore the mode of the ethernet chip will be set (i.e. 10Mbit full-duplex mode).

void PacketAnalyser()
Does what you expect of its name, analysing a packet of which type it is. Following types can be determined: ARP, ICMP, IGMP, GGP, TCP, EGP, UDP, DHCP. But only for ARP, ICMP and UDP exist methods to process the according packets. Fortunately you don't have to care about any ARP or ping replies, just focus on processing the UDP data stored in the *ReceivedUDPData* struct.

void PingReply(Xuint32* ptrICMPPacket)
Sends back a ping reply to an incoming ping request. The argument pointer points at

the first byte of the ICMP header.

UDP functions

void RecvUDP(Xuint32* ptrUDPPacket)
Expects a pointer to the beginning of the IP Header. It reads then out the important information of the UDP packet and writes them in a struct called *ReceivedUDPPacketInfo* and for further use in other programs *ReceivedUDPData*. All structs are described in another section.

Xboolean UDPReply(Xuint16 DestinationPort, Xuint32* ptrDataBuffer)
Expects that the board already has received a UDP packet and sends back a UDP packet to the destination which informations about are found in the *ReceivedPacketInfo* struct.

Xboolean SendUDPTo(Xuint8 bytDestinationIP[4], Xuint8 bytDestinationMAC[6], Xuint16 intDestinationPort, Xuint8* ptrDataBuffer, Xuint16 DataLength)
Sends a UDP packet IP, MAC address and the wished port.
lngDestinationAddress[4] array of bytes which holds IP address, *bytDestinationMAC[6]* array of bytes representing the MAC address, *DestinationPort* to which port has the data to be sent, *ptrDataBuffer* a byte-pointer to the data you want to send, *BufferLength* length of the data buffer. The function returns *XTRUE* if it succeeds to send the data, *XFALSE* if it fails.

Xboolean SendUDP(Xuint8 bytDestinationIP[4], Xuint16 intDestinationPort, Xuint8* ptrDataBuffer, Xuint16 DataLength)
The third of these UDP sending functions is thought to make life easier for applications. It's the same function as *SendUDPTo()* but you don't have to know the MAC address of the destination. Resolving the MAC address will be executed by an ARP request mechanism and then saved in an ARP table. (array *ARPTable*)

ARP functions

Unfortunately the *SendARPRequest()* and the *UpdateARPTable()* functions couldn't be tested enough, their still in evaluation mode. Thus you should prefer *SendUDPTo()* and *UDPReply()* to the *SendUDP()* function, which needs the *ARP Request* mechanism.

void ARPReply(Xuint32* ptrARPData)
Replies an ARP request. Normally you won't get in contact with this function because it has no use for other applications or drivers. For completeness: *ptrARPData* has to be a pointer to the received ARP data buffer and the function will then process the data and send back the answer.

void SendARPRequest(Xuint8* IPAddress)
When an application uses *SendUDP()* and the destination has no entry in the ARP table, *SendUDP()* will call *SendARPRequest()* first before sending. Because of the non-blocking feature a flag is set *(bolAwaitingARPReply = XTRUE)* and the function is left to wait for an ARP reply or to process some other packets first.

void UpdateARPTable(Xuint32* ptrAddress)
After arriving and detecting an ARP reply, *UpdateARPTable()* is called to extract all information out of the buffer which *ptrAddress* points to.

Checksum calculations

IP Checksum definition: The IP checksum is the 16 bit 1's complement of the 1's complement sum of all 16 bit words in the header (or even of the whole packet for UDP). The 1's complement sum is done by summing the numbers and adding the carry (or carries) to the result.
As shown in RFC 1071, the checksum calculation is done in the following way:
1. Adjacent octets to be checksummed are paired to form 16-bit integers, and the 1's complement sum of these 16-bit integers is formed.
2. To generate a checksum, the checksum field itself is cleared, the 16-bit 1's complement sum is computed over the octets concerned, and the 1's complement of this sum is placed in the checksum field.
3. To check a checksum, the 1's complement sum is computed over the same set of octets, including the checksum field. If the result is all 1 bits (-0 in 1's complement arithmetic), the check succeeds.

Example
Packet
00 00 5E 00 FA CE A9 B8 00 00
(00 00 at the end is the checksum field)
Form the 16-bit words
0000 5E00 FACE A9B8
Calculate 2's complement sum
0000 + 5E00 + FACE + A9B8 = 0002 0286 (store the sum in a 32-bit word)
Add the carries (0002) to get the 16-bit 1's complement sum
0286 + 002 = 0288
Calculate 1's complement of the 1's complement sum
0288 = FD77
We send the packet including the checksum FD 77
01 00 F2 03 F4 F5 F6 F7 FD 77
At the receiving
0000 + 5E00 + FACE + A9B8 + FD77 = 0002 FFFD
FFFD + 0002 = FFFF → which checks OK

Xuint16 calc_chksum(Xuint8* buf, int len) and
Xuint16 calc_checksum32(Xuint32* buf, Xuint16 len)
Actually identical functions, the only difference is the buffer pointer. It is needed only as supporting act. The calculation of the IP header checksum is the job these functions have to do, returning a 16bit value of the checksum.

Xuint16 calc_UDPchecksum(Xuint32* buf, Xuint16* PseudoHeader, Xuint16 DataLength)
The checksum calculation of a UDP packet is more special, because we need additional information, the pseudo header. Thus there is a third argument to give the function.

Ethernet chip settings

The following six functions are used for setting the right values of the ethernet chip on the board, i.e. speed or duplex mode. Read the datasheet of the Intel LXT970A for further informations on these values and the different registers.

void WriteMDIO(Xuint8 argReg, Xuint16 argValue)
Writes the value *argValue* into the argument register *argValue*.

Xuint16 ReadMDIO(Xuint8 reg)
Reads out the argument register and returns the value as a short.

void DoMDIOClk()

Creates one clock period on the MDC (clock line), basically used after a *SetMDIO()* call.

void SetMDIO(Xboolean bit)
Writes a 'high' (=XTRUE) or a 'low' (=XFALSE) out on the MDIO (data line).

void SetEthernetMode()
Calls all needed functions to set the 10Mbit full duplex mode. Called from the *Init()* function.

void SetAutoNegotiationMode()
Calls all needed functions to set the auto-negotiation mode. The auto-negotiation mode choose the right speed and all other values by communicating and exchanging 'skills' with the other NIC (in the back-to-back installation). Called from the *Init()* function.

4.3 Structures & Variables

To get an overview of the common used variables and typedefined structures, here is the complete network.h file, including the comments and after the listing some hopefully useful explanations.

network.h

```
#ifndef __NETWORK__
#define __NETWORK__

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

//different modes
//#define BENCHMARK
#define NORMAL_UDP
//#define DEBUG_MODE

#define TX_BASEADDR XPAR_OPB_ETHERNETLITE_0_BASEADDR
#define RX_BASEADDR XPAR_OPB_ETHERNETLITE_0_BASEADDR+0x2000
//address of receive flag
#define RX_CTL_ADDR XPAR_OPB_ETHERNETLITE_0_BASEADDR+0x3FFC

#define TX_LEN_ADDR TX_BASEADDR+0x1FF4
#define TX_LEN_ADDR_HIGHER TX_BASEADDR+0x1FF4
//address of transmit flag
#define TX_CTL_ADDR TX_BASEADDR+0x1FFC

#define XFBoard_CommandPort 0x22B8  //not used yet
#define XFBoard_DataPort 0x22B9 //not used yet
#define XFBoard_UDPPort 0x22    //Port 34 is free
#define PC_SOURCEPORT 0xF0CE    //Port 61646


typedef struct {
        Xboolean bolARPRequest; //if a request has been detected
        Xboolean bolARPReply;   //true if board has to reply
        Xboolean bolDHCPPacket;
```

```
        Xboolean bolDHCPReply;  //true if board has to reply
        Xboolean bolICMPPacket;
        Xboolean bolICMPReply;  //true if board has to reply
        Xboolean bolUDPPacket;
} PacketDecisionStruct;

typedef struct {
        Xuint8 bytMACAddress[6];    //for saving the source address
        Xuint8 bytIPHeader[20];         //for saving the IP header
        Xuint8 bytIPAddress[4];         //address of source
} XPacketInfo;  //header of a packet

typedef struct {
        Xboolean bolDataAvailable;  //XTRUE if data is stored
        Xuint16 intSourcePort;
        Xuint32 *ptrData;
        Xuint16 intDataLength;  //only the length of the data
        Xuint16 volatile intDestinationPort;
} XPacketData;  //holds data of UDP packets

typedef struct {
        Xboolean bolDataAvailable;
        Xuint16 intSourcePort;
        Xuint8 *ptrData;
        Xuint16 intDataLength;
        Xuint16 volatile intDestinationPort;
} XPacketData8; //same struct as above
                //but with byte pointer to data

typedef struct {
        Xuint8 bytProtocolAddress[4];
        Xuint8 bytHardwareAddress[6];
} ARPEntry; //a MAC address for the IP address

//----------------
//general variables
//----------------
// 32bit pointers to the receive & transmit buffers
// ptrRXDest -> Ethernet destination address
// ptrRXSource -> Ethernet source address
// ptrRXType -> Type field, 0x0800=ethernet packet,  0x0806=ARP packet
// ptrRXData -> Data, normally begin of IP header
extern Xuint32 *ptrRXDest, *ptrRXSource, *ptrRXType, *ptrRXData;
extern Xuint32 *ptrTXDest, *ptrTXSource, *ptrTXType, *ptrTXData;
extern Xuint8 MACBoardAddress[6];   //the board's own MAC address
extern Xuint8 IPBoardAddress[4];    //the board's own IP address
extern Xuint8 BenchBuf[264];    //only for benchmarks
extern Xuint32 *ptrBenchBuf;    //only for benchmarks
extern Xuint32 volatile *rxflag;    //pointer to receive flag
extern ARPEntry ARPTable[5];    //the XFBoard's own ARP table
extern Xuint8 bytNumberOfARPEntries;
extern Xboolean bolAwaitingARPReply;    //indicates whether the board
                                        //is waiting for a reply
extern Xboolean bolAwaitingICMPReply;
// for storing a packet while waiting for an ARP reply
extern XPacketInfo QueuePacketInfo;
extern XPacketData8 QueuePacketData;
extern Xuint8 tempBuffer[1400];
// different packet storing structures
extern XPacketInfo ReceivedPacketInfo;
extern PacketDecisionStruct DecisionInfo;
extern XPacketData ReceivedUDPData;
```

```
//functions
extern void Init();
extern void PacketAnalyser();
extern void PingReply(Xuint32* ptrICMPPacket);

extern void ARPReply(Xuint32* ptrARPData);
extern void SendARPRequest(Xuint8* IPAddress);
extern void UpdateARPTable(Xuint32* ptrAddress);

extern Xuint16 calc_chksum(Xuint8* buf, int len);
extern Xuint16 calc_checksum32(Xuint32* buf, Xuint16 len);
extern Xuint16 calc_UDPchecksum(Xuint32* buf, Xuint16* PseudoHeader, Xuint16
    DataLength);

extern void RecvUDP(Xuint32* ptrUDPPacket);
extern Xboolean UDPReply(Xuint16 DestinationPort, Xuint32* ptrDataBuffer);
extern Xboolean SendUDPTo(Xuint8 bytDestinationIP[4], Xuint8 bytDestinationMAC[6],
    Xuint16 intDestinationPort, Xuint8* ptrDataBuffer, Xuint16 DataLength);
extern Xboolean SendUDP(Xuint8 bytDestinationIP[4], Xuint16 intDestinationPort,
    Xuint8* ptrDataBuffer, Xuint16 DataLength);

extern void WriteMDIO(Xuint8 argReg, Xuint16 argValue);
extern Xuint16 ReadMDIO(Xuint8 reg);
extern void DoMDIOClk();
extern void SetMDIO(Xboolean bit);
extern void SetEthernetMode();
extern void SetAutoNegotiationMode();

#endif
```

4.3.1 More details about structures & variables

PacketDecisionStruct is used to store whether you would prefer to let *PacketAnalyser()* reply the ARP and ICMP requests automatically or not. Default is XTRUE for all booleans.
XPacketInfo contains all the header information of the arrived and already processed packet.
XPacketData is useful if you have a program which is based on the support of UDP sending/receiving.
ARPTable[5] is the board own 5-entry ARP table.
MACBoardAddress[6] is the array which is filled with the default address in the *Init()* function.
IPBoardAddress[4] is the array of the board's IP address, initialised in *Init()* too.
QueuePacketInfo & QueuePacketData are only used when *SendUDP()* wants to send a packet to an unknown destination. Then it will send first an ARP request and saves in the meantime the values into these 'Queue' structs.

4.4 How to use & code snippets

After all the useful theory now a short example how to use the 'library' and an overview of the program flow for a better understanding of the operation mode. There are two example system.c files placed in this document to show how to integrate functions or programs in the program flow. The *PacketAnalyser()* function responds to ARP and ICMP requests, you only have to care about the UDP packets stored in the *ReceivedUDPData* struct.

Normal version

There are three different modes to choose among: a normal mode, the debug mode and a benchmark mode.
Normal mode: just sends back the received UDP packet.
Debug mode: sends back the received UDP packet and prints out some packet info.
Benchmark mode: After receiving one UDP packet, sending UDP packets to origin breakless.

system.c

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

//---------------------------------------------------------------
// A system.c file to show how to handle the network functions
// contained in the other c-files.
//
// Last updatet: 1.12.03
//
// Author: Marco Kuster
//     Reason: Student Thesis - Firmware HWOS
//---------------------------------------------------------------

//Xboolean bolBenchmarkStop;


// system is set to 'normal' mode
#ifdef NORMALUDP
//---------------------------------------
// function main
//---------------------------------------
int main(){

    Xuint32 i, j;
    Xuint8 IPAddress[4],MACAddress[6], UDPData[20];
    Xuint16 UDPDataLength;


    xil_printf("\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n
        \r\n");
    xil_printf("Ethernet_Package_XFBoard_ready...\r\n");
    xil_printf(">\r\n");

    // initialise network environment
    Init();
```

```
    //to this address we want to send a UDP
    IPAddress[0] = 0xA9; IPAddress[1] = 0xFE; IPAddress[2] = 0x2D; IPAddress[3] = 0
        x31;
    MACAddress[0] = 0x00; MACAddress[1] = 0x01; MACAddress[2] = 0x02; MACAddress
        [3] = 0x90;
    MACAddress[4] = 0xF1; MACAddress[5] = 0x97;

    strcpy(UDPData, "Hello world!\n");

  // polling mode
    while(1) {

        //do nothing when nothing arrives
        while(*rxflag==0);

        //analyse the new packet
        PacketAnalyser();

        //an example what to do with the extracted UDP packet data
        if (ReceivedUDPData.bolDataAvailable == XTRUE) {

            //
            // do now whatever you would like to do with your data
            //

            UDPReply(ReceivedUDPData.intSourcePort, ReceivedUDPData.ptrData);

            #ifdef DEBUG_MODE
                xil_printf("\r\nUDP: Sourceport=");
                xil_printf("%X", ReceivedUDPData.intSourcePort);
                xil_printf("UDP: Destport=");
                xil_printf("%X", ReceivedUDPData.intDestinationPort);
                xil_printf("UDP: Length=");
                xil_printf("%X", ReceivedUDPData.intDataLength);
                xil_printf("Data=");
                for (i=0;i<ReceivedUDPData.intDataLength;i++) {
                    xil_printf("%c", *(ReceivedUDPData.ptrData+i));
                }
            #endif

            //data processed
            ReceivedUDPData.bolDataAvailable = XFALSE;
        }
         // new data can be read out now
        *rxflag=0;
    }

    return 0;
}//end of main()
//---------------------------------------

#endif


// the benchmark mode is chosen
#ifdef BENCHMARK

int main() {

    Xuint32 i, lngCounter;
    Xboolean bolBenchmarkModeEnabled;
```

```
    Xuint8 BenchBuf[58];
    Xuint32 *ptrBenchBuf;

    BenchBuf[58] = "abcdefghijklmnopqrstvwabcdefghijklmnopqrstvwabcdefghijklm";
    *ptrBenchBuf = (Xuint32*)BenchBuf;

    for (i=0;i<58;i++) {
        *(ptrBenchBuf+i) = BenchBuf[i];
    }

    //flag=(Xuint32*)(RXCTLADDR);  //flag set when packet arrived
    //*flag=0;

    xil_printf("\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n
        \r\n");
    xil_printf("Ethernet Package XFBoard ready...\r\n");
    xil_printf("In benchmark mode\r\n");
    xil_printf(">\r\n");

    // initialise network environment
    Init();

    while(!ReceivedUDPData.bolDataAvailable) {
        while(*rxflag==0);
        //xil_printf("Packet received...\r\n");
        PacketAnalyser();
        *rxflag=0;
    }

    //send UDP packets without breaks
    while(1) {
       SendUDP(ReceivedPacketInfo.bytIPAddress, ReceivedUDPData.intSourcePort,
            ReceivedUDPData.ptrData, ReceivedUDPData.intDataLength);
    }

    return 0;
}//end of main()
//---------------------------------------

#endif
```

Minimal version

The second *system* listing is a minimal version. The places where you have to put your code or functions in are marked with a comment */**** ... ****.*

system-minimal.c

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

//---------------------------------------------------------------
// A system.c file with network functionality and minimal code
//
// Last updatet: 10.12.03
//
// Author: Marco Kuster
```

```
//     Reason: Student Thesis - Firmware HWOS
//-------------------------------------------------------------------


//-----------------------------------------
// function main
//-----------------------------------------
int main(){

   //Xuint32 i, j;
   Xuint8 IPAddress[4],MACAddress[6], UDPData[20];
   Xuint16 UDPDataLength;


   xil_printf("\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n
       \r\n");
   xil_printf("Ethernet Package XFBoard ready...\r\n");
   xil_printf(">\r\n");

   // initialise network environment
   Init();

   //to this address we want to send a UDP
    //**** fill in here your own addresses and data ****
   IPAddress[0] = 0xA9; IPAddress[1] = 0xFE; IPAddress[2] = 0x2D; IPAddress[3] = 0
       x31;
   MACAddress[0] = 0x00; MACAddress[1] = 0x01; MACAddress[2] = 0x02; MACAddress
       [3] = 0x90;
   MACAddress[4] = 0xF1; MACAddress[5] = 0x97;

   strcpy(UDPData, "Hello world!\n");

 // polling mode
   while(1) {

      //do nothing when nothing arrives
      while(*rxflag==0);

      //analyse the new packet
      PacketAnalyser();

        //**** either write your code down here.. ****

      //arrived packet is a UDP packet
      if (ReceivedUDPData.bolDataAvailable == XTRUE) {

         //
         // **** ..or when processing UDP data, here ****
         //

         //just send back the data to the origin
         UDPReply(ReceivedUDPData.intSourcePort, ReceivedUDPData.ptrData);

         //data processed
         ReceivedUDPData.bolDataAvailable = XFALSE;
      }
       // new data can be read out now
      *rxflag=0;
   }

   return 0;
}//end of main()
```

```
//-------------------------------------------
```

'Where to put my code?' may be your question after reading the listing. As the Ethernet Lite Core from Xilinx doesn't throw interrupts you have to poll the input queue. Thus you have to wait first for arriving packets. As soon as the pointer **rxflag* is set to 1, the function *PacketAnalyser()* is called to process the packet and response to it atuomatically, respectively fill in the UDP packet structure *ReceivedUDPData*.
And now it's your turn to fill in the code! When in benchmark mode in the *system.c* listing, the data of the UDP packet is given out on the RS232 interface.

4.4.1 Program flow

The last part of this section consists of a drawing (picture 4.1 on page 26)of the program flow for better understanding and illustration of the functionality.

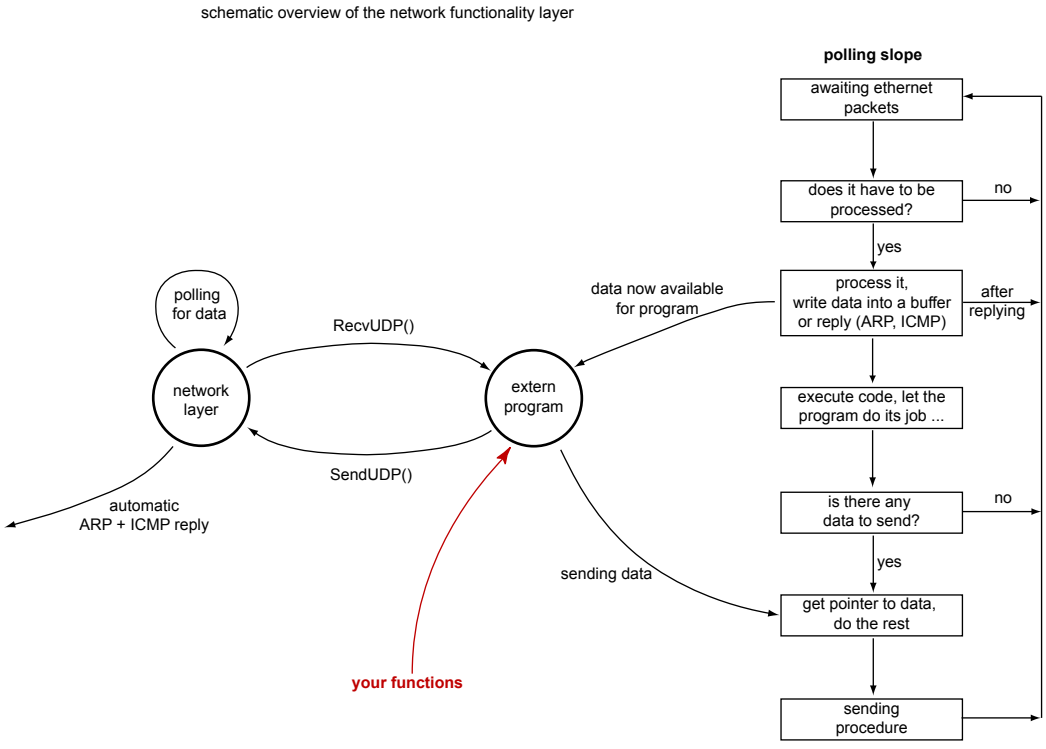

Figure 4.1: Functionality Scheme

4.5 XF Boardloader

In addition to the basic functions, the code of the bitstream loader is mentioned and integrated in this chapter as well. The network functions are exactly the same only the *system.c* was rewritten and adapted for this special job. More about the PC program for downloading the bitstream you find in chapter 5.

4.5.1 Functions

List of special functions for the programming of the R-FPGA and their use

Xboolean Prepare2SendFull()
Sets up a download connection over the SelectMap Port from C-FPGA to R-FPGA. As soon as the function gets the 'Init' bti from the R-FPGA, the download of a full configuration bitstream file can be started.

Xboolean Prepare2SendPartial()
Same function as *Prepare2SendFull()*, but for partial bitstream configuration files.

Xboolean PrepareFinishSequence()
After having sent the last bit the program has to wait for the 'done' bit returned by the R-FPGA when setting up all connections was successful.

SendBitstream(Xuint32 *argData, Xuint16 LenData)
Sends *LenData* bytes to the R-FPGA, the data starts at *argData*.

void WriteByte(Xuint8 argByte)
Writes one byte to the R-FPGA.

All the other functions in the *system.c* file, which are used for setting signals on the SelectMap Port, are self-explaining.

4.5.2 Code

system-xfloader.c

```
while(1) {

    while(*rxflag==0);

    //analyse the new packet
    PacketAnalyser();

      //if it is a UDP packet, enter the condition
    if (ReceivedUDPData.bolDataAvailable ==XTRUE) {
      //is already a download running?
      if (!bolBitstreamToDownload) {
        //if not, continue here
        //is it an initialising packet for a data transfer to the FPGA?
        //compare with 'Full'
        if( *(ReceivedUDPData.ptrData) == 0x46 &&
            *(ReceivedUDPData.ptrData+1) == 0x75 &&
            *(ReceivedUDPData.ptrData+2) == 0x6C &&
            *(ReceivedUDPData.ptrData+3) == 0x6C )
            {
```

```
            //=> full configuration
            //now we're ready to initialise the interface to the R-FPGA
            if (Prepare2SendFull()) {
                xil_printf("\r\nInit seqence done, R-FPGA is ready to receive
                    bitstream data\r\n");
            }
            UDPData[0]=0x46;
            UDPData[1]=0x75;
            UDPData[2]=0x6C;
            UDPData[3]=0x6C;
            //send back 'Full' to start the download
            SendUDPTo(DestIP,DestMAC,61646,UDPData,10);
        }
        //compare with 'Part'
        else if( *(ReceivedUDPData.ptrData) == 0x50 &&
            *(ReceivedUDPData.ptrData+1) == 0x61 &&
            *(ReceivedUDPData.ptrData+2) == 0x72 &&
            *(ReceivedUDPData.ptrData+3) == 0x74 )
            {
                //=> partial configuration
                //now we're ready to initialise the interface to the R-FPGA
                if (Prepare2SendPartial()) {
                    xil_printf("\r\nInit seqence done, R-FPGA is ready to receive
                        bitstream data\r\n");
                }
                UDPData[0]=0x50;
                UDPData[1]=0x61;
                UDPData[2]=0x72;
                UDPData[3]=0x74;
                //send back 'Part' to start the download
                SendUDPTo(DestIP,DestMAC,61646,UDPData,10);
        }
        //the first packet has the sequence number '00'
        else if(*(ReceivedUDPData.ptrData) == 0x00 &&
                *(ReceivedUDPData.ptrData+1) == 0x00)
                {
                    //the bitstream download has started
                    bolBitstreamToDownload = XTRUE;
                    exseqnum=1;
                    //write data to the R-FPGA
                    SendBitstream((ReceivedUDPData.ptrData+2), (ReceivedUDPData.
                        intDataLength-2));
                    xil_printf("Begin to send data...\r\n");

                    UDPData[0]=0;
                    UDPData[1]=0;
                    SendUDPTo(DestIP,DestMAC,61646,UDPData,10);
        }
    }
    //it's not an initial packet..
    else {
        //is it the last packet?
        //=> indicated by 0xFFFF value in the sequence number field
        if(*(ReceivedUDPData.ptrData) == 0xFF && *(ReceivedUDPData.ptrData+1)
            == 0xFF) {
            //write data to the R-FPGA
            xil_printf("Last packet sent to FPGA, download finished!\r\n");
            bolBitstreamToDownload = XFALSE;
            if (PrepareFinishSequence()) {
                //done bit set => send it to PC program
                UDPData[0]=0x44;
                UDPData[1]=0x4F;
```

```
                    UDPData[2]=0x4E;
                    UDPData[3]=0x45;
                    xil_printf("send done acknowledge\r\n");
                    SendUDPTo(DestIP,DestMAC,61646,UDPData,4);
                }
                //break;
            }
            else {
                //write data to the R-FPGA
                seqnum=256*(*(ReceivedUDPData.ptrData))+(*(ReceivedUDPData.ptrData
                    +1));
                if(seqnum!=exseqnum) {
                    xil_printf("###SEQUENCE ERROR ### (Expected=%d, Received=%d)\r\
                        n",exseqnum,seqnum);
                }
                SendBitstream(ReceivedUDPData.ptrData+2, ReceivedUDPData.
                    intDataLength-2);
                //xil_printf("\r\n data sent to FPGA\r\n");
                //xil_printf("%dth packet sent to FPGA\r\n",seqnum);

                //UDP packet was processed, thus send packet with sequence number
                //back to source
                UDPData[0]=seqnum/256;
                UDPData[1]=seqnum%256;
                SendUDPTo(DestIP,DestMAC,61646,UDPData,10);
            }
            //increase expected sequence number
            exseqnum++;
        }

        //data processed
        ReceivedUDPData.bolDataAvailable = XFALSE;
    }
     // receive buffer is ready for new data
    *rxflag=0;
  }

  return 0;
}//end of main()
//---------------------------------------
```

4.5.3 Flow Diagram

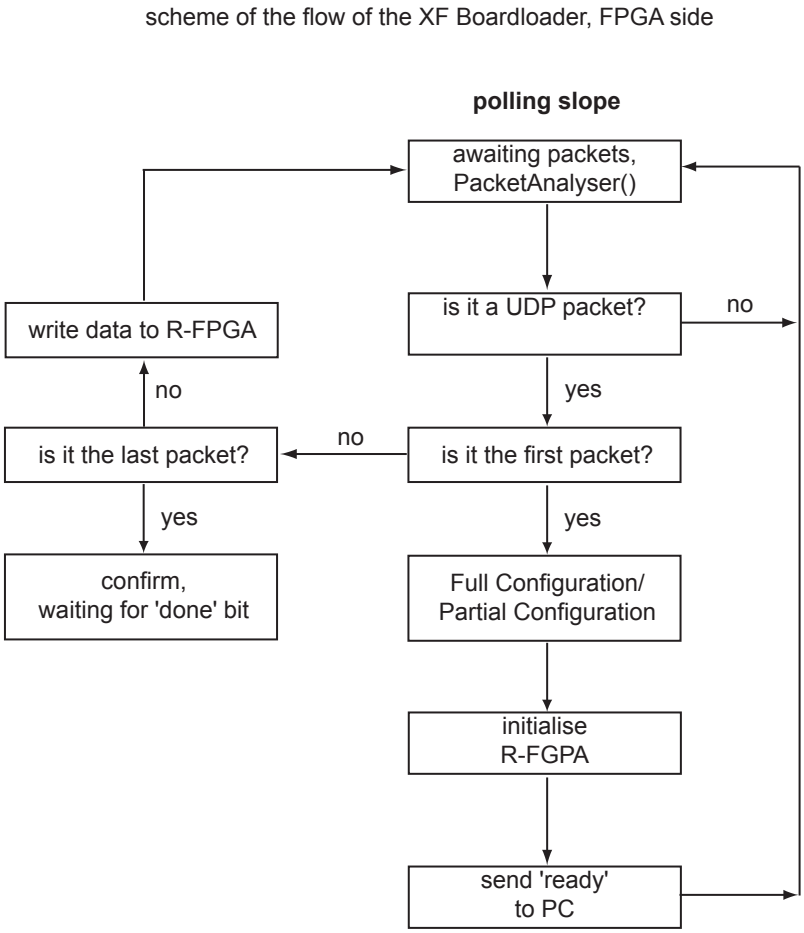

Figure 4.2: XF Loader Scheme

Chapter 5

PC Programs

5.1 Visual UDP

Two small Windows based programs were written in Visual C++ with MFC. The first one - Visual UDP - is able to send UDP packets to any host and list the answer in a list box. As you can see on the picture it is very simple to use, therefore you don't find a lot of text on how to use it. To mention is only, when you intend to start sending packets in intervals, just set up the interval time and press then the button 'send'. And obviously to stop, the button stop within the interval frame.

The code itself is not documented in this manual, because it would be too much to explain all the MFC functions and their use. Presumed that you are experienced in programming MFC, it should be no problem to understand the code, because it is well documented.

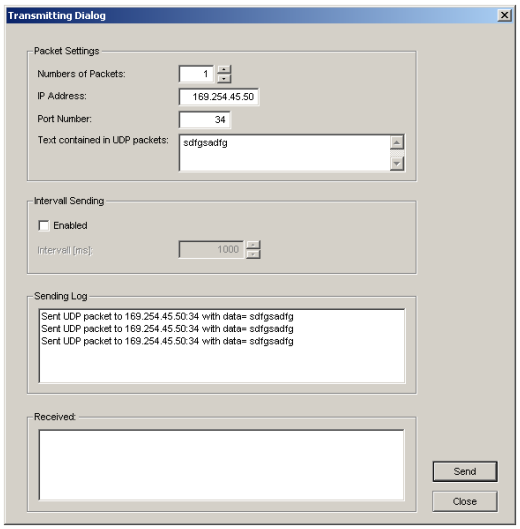

Figure 5.1: Screenshot of Visual UDP

5.2 XFBoardloader

The second program, the XFBoardloader, is as simple to use as it looks. Choose the bitstream file you want to download on the R-FPGA by pressing the button 'Browse'. To start the download press 'Full Configuration FPGA' or 'Partial Configuration FPGA' depending on what kind of configuration file you have created and intend to install. Don't forget to type in the right IP address and port number of the board *before* downloading.

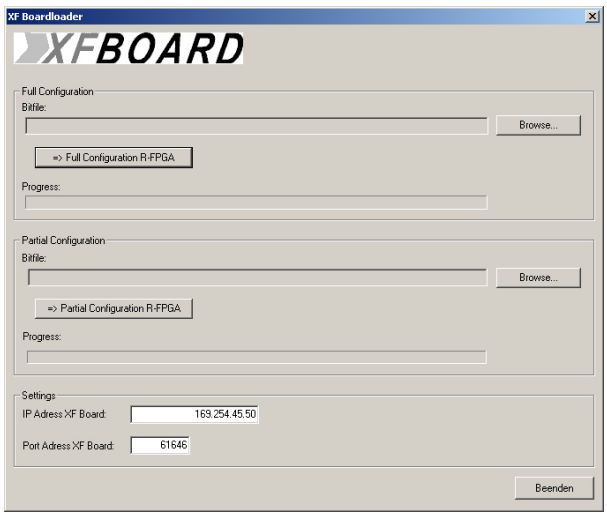

Figure 5.2: Screenshot of the XF Boardloader

Chapter 6

Measurements

Over the most important functions and implementations were measurements made to determine the potential of the transmission.

Environment:
Time measurement program: Ethereal - Network Protocol Analyzer
Mode: 10Mbit full-duplex
Delay: Difference between leaving time of a packet from the PC to the incoming time of the response packet
Programs used: Visual UDP for sending UDP packets, XFBoardloader for sending a bitstream file to the R-FPGA
Packet sizes: ARP packet = 60 bytes, ping request & reply = 32 bytes, UDP packet = 50 bytes, data packet for bitstream download = 1302 bytes

What measured	Time
ARP replies (ARP table empty)	delays: 1. 351μs 2. 351μs 3. 364μs
Ping replies	delays: 1. 405μs 2. 426μs 3. 434μs 4. 547μs 5. 426μs 6. 428μs 7. 417μs 8. 428μs
UDP reflect	delays: 1. 399μs 2. 489μs 3. 424μs 4. 420μs 5. 422μs 6. 426μs

	...
UDP Timer (interval sending)	delays: the values measured were in the same range as the values from *UDP reflect*
XFBoardloader	
Initialise Writing data	51ms 1. 23ms 2. 5ms 3. 5.6ms 4. 5.55ms ...
Time for whole bitstream download:	1. 6.09s 2. 6.13s

Bibliography

[1] Nobs Samuel. Prototype board for reconfigurable os. Manual of the prototype XFBoard, July 2003.

[2] Postel Jon. *RFC 792 - Internet Control Message Protocol*, September 1981.

[3] Prosise Jeff. *Windows-Programmierung mit MFC.* Microsoft Press Redmond, 2nd edition, 1999.

[4] Stroustrup Bjarne. *Die C++ Programmiersprache.* Addison-Wesley, 3rd edition, 1998.

[5] Jennrich Tischer. *Internet intern.* Data Becker, 1st edition, 1997.

[6] Xilinx. *Embedded System Tools Guide*, May 2003.

[7] Xilinx. *OPB Ethernet Lite Media Access Controller*, November 2002.

[8] Intel. *Intel LXT970A, Dual-Speed Fast Ethernet Transceiver*, January 2001.

[9] Postel Jon. *RFC 791 - Internet Protocol*, September 1981.

[10] Plummer David C. *RFC 826 - Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware*, November 1982.

[11] Postel Jon. *RFC 768 - User Datagram Protocol*, August 1980.

[12] Heinz Carsten. *The Listings Package*, April 2002.

[13] Irene Hyna Oetiker Tobias, Hubert Partl and Elisabeth Schlegl. *The not so short introduction to Latex*, September 2003.